nameko

Release 2.12.0

October 19, 2019

Contents

[nah-meh-koh]

> A microservices framework for Python that lets service developers concentrate on application logic and encourages testability.

A nameko service is just a class:

```
# helloworld.py

from nameko.rpc import rpc

class GreetingService:
    name = "greeting_service"

    @rpc
    def hello(self, name):
        return "Hello, {}!".format(name)
```

Note: The example above requires RabbitMQ, because it's using the built-in AMQP RPC features. RabbitMQ installation guidelines offer several installation options, but you can quickly install and run it using Docker.

To install and run RabbitMQ using docker:

```
$ docker run -d -p 5672:5672 rabbitmq:3
```

You might need to use sudo to do that.

You can run it in a shell:

```
$ nameko run helloworld
starting services: greeting_service
...
```

And play with it from another:

```
$ nameko shell
>>> n.rpc.greeting_service.hello(name="")
'Hello, !'
```

CHAPTER 1

User Guide

This section covers most things you need to know to create and run your own Nameko services.

1.1 What is Nameko?

Nameko is a framework for building microservices in Python.

It comes with built-in support for:

- RPC over AMQP
- Asynchronous events (pub-sub) over AMQP
- Simple HTTP GET and POST
- Websocket RPC and subscriptions (experimental)

Out of the box you can build a service that can respond to RPC messages, dispatch events on certain actions, and listen to events from other services. It could also have HTTP interfaces for clients that can't speak AMQP, and a websocket interface for, say, Javascript clients.

Nameko is also extensible. You can define your own transport mechanisms and service dependencies to mix and match as desired.

Nameko strongly encourages the *dependency injection* pattern, which makes building and testing services clean and simple.

Nameko takes its name from the Japanese mushroom, which grows in clusters.

1.1.1 When should I use Nameko?

Nameko is designed to help you create, run and test microservices. You should use Nameko if:

- You want to write your backend as microservices, or
- You want to add microservices to an existing system, and
- You want to do it in Python.

Nameko scales from a single instance of a single service, to a cluster with many instances of many different services.

The library also ships with tools for clients, which you can use if you want to write Python code to communicate with an existing Nameko cluster.

1.1.2 When shouldn't I use Nameko?

Nameko is not a web framework. It has built-in HTTP support but it's limited to what is useful in the realm of microservices. If you want to build a webapp for consumption by humans you should use something like flask.

1.2 Key Concepts

This section introduces Nameko's central concepts.

1.2.1 Anatomy of a Service

A Nameko service is just a Python class. The class encapsulates the application logic in its methods and declares any *dependencies* as attributes.

Methods are exposed to the outside world with *entrypoint* decorators.

```
from nameko.rpc import rpc, RpcProxy

class Service:
    name = "service"

    # we depend on the RPC interface of "another_service"
    other_rpc = RpcProxy("another_service")

    @rpc  # `method` is exposed over RPC
    def method(self):
        # application logic goes here
        pass
```

Entrypoints

Entrypoints are gateways into the service methods they decorate. They normally monitor an external entity, for example a message queue. On a relevant event, the entrypoint may "fire" and the decorated method would be executed by a service *worker*.

Dependencies

Most services depend on something other than themselves. Nameko encourages these things to be implemented as dependencies.

A dependency is an opportunity to hide code that isn't part of the core service logic. The dependency's interface to the service should be as simple as possible.

Declaring dependencies in your service is a good idea for *lots of reasons*, and you should think of them as the gateway between service code and everything else. That includes other services, external APIs, and even databases.

Workers

Workers are created when an entrypoint fires. A worker is just an instance of the service class, but with the dependency declarations replaced with instances of those dependencies (see *dependency injection*).

Note that a worker only lives for the execution of one method – services are stateless from one call to the next, which encourages the use of dependencies.

A service can run multiple workers at the same time, up to a user-defined limit. See *concurrency* for details.

1.2.2 Dependency Injection

Adding a dependency to a service class is declarative. That is, the attribute on the class is a declaration, rather than the interface that workers can actually use.

The class attribute is a `DependencyProvider`. It is responsible for providing an object that is injected into service workers.

Dependency providers implement a `get_dependency()` method, the result of which is injected into a newly created worker.

The lifecycle of a worker is:

1. Entrypoint fires
2. Worker instantiated from service class
3. Dependencies injected into worker
4. Method executes
5. Worker is destroyed

In pseudocode this looks like:

```
worker = Service()
worker.other_rpc = worker.other_rpc.get_dependency()
worker.method()
del worker
```

Dependency providers live for the duration of the service, whereas the injected dependency can be unique to each worker.

1.2.3 Concurrency

Nameko is built on top of the eventlet library, which provides concurrency via "greenthreads". The concurrency model is co-routines with implicit yielding.

Implicit yielding relies on monkey patching the standard library, to trigger a yield when a thread waits on I/O. If you host services with `nameko run` on the command line, Nameko will apply the monkey patch for you.

Each worker executes in its own greenthread. The maximum number of concurrent workers can be tweaked based on the amount of time each worker will spend waiting on I/O.

Workers are stateless so are inherently thread safe, but dependencies should ensure they are unique per worker or otherwise safe to be accessed concurrently by multiple workers.

Note that many C-extensions that are using sockets and that would normally be considered thread-safe may not work with greenthreads. Among them are librabbitmq, MySQLdb and others.

1.2.4 Extensions

All entrypoints and dependency providers are implemented as "extensions". We refer to them this way because they're outside of service code but are not required by all services (for example, a purely AMQP-exposed service won't use the HTTP entrypoints).

Nameko has a number of *built-in extensions*, some are *provided by the community* and you can *write your own*.

1.2.5 Running Services

All that's required to run a service is the service class and any relevant configuration. The easiest way to run one or multiple services is with the Nameko CLI:

```
$ nameko run module:[ServiceClass]
```

This command will discover Nameko services in the given `modules` and start running them. You can optionally limit it to specific `ServiceClasss`.

Service Containers

Each service class is delegated to a `ServiceContainer`. The container encapsulates all the functionality required to run a service, and also encloses any *extensions* on the service class.

Using the `ServiceContainer` to run a single service:

```
from nameko.containers import ServiceContainer

class Service:
    name = "service"

# create a container
container = ServiceContainer(Service, config={})

# ``container.extensions`` exposes all extensions used by the service
service_extensions = list(container.extensions)

# start service
container.start()

# stop service
container.stop()
```

Service Runner

`ServiceRunner` is a thin wrapper around multiple containers, exposing methods for starting and stopping all the wrapped containers simultaneously. This is what `nameko run` uses internally, and it can also be constructed programmatically:

```
from nameko.runners import ServiceRunner
from nameko.testing.utils import get_container

class ServiceA:
    name = "service_a"

class ServiceB:
```

```
    name = "service_b"

# create a runner for ServiceA and ServiceB
runner = ServiceRunner(config={})
runner.add_service(ServiceA)
runner.add_service(ServiceB)

# ``get_container`` will return the container for a particular service
container_a = get_container(runner, ServiceA)

# start both services
runner.start()

# stop both services
runner.stop()
```

If you create your own runner rather than using *nameko run*, you must also apply the eventlet monkey patch.

See the nameko.cli.run module for an example.

1.3 Installation

1.3.1 Install with Pip

You can install nameko and its dependencies from PyPI with pip:

```
pip install nameko
```

1.3.2 Source Code

Nameko is actively developed on GitHub. Get the code by cloning the public repository:

```
git clone git@github.com:nameko/nameko.git
```

You can install from the source code using setuptools:

```
python setup.py install
```

1.3.3 RabbitMQ

Several of Nameko's built-in features rely on RabbitMQ. Installing RabbitMQ is straightforward on most platforms and they have excellent documentation.

With homebrew on a mac you can install with:

```
brew install rabbitmq
```

On debian-based operating systems:

```
apt-get install rabbitmq-server
```

For other platforms, consult the RabbitMQ installation guidelines.

The RabbitMQ broker will be ready to go as soon as it's installed – it doesn't need any configuration. The examples in this documentation assume you have a broker running on the default ports on localhost and the rabbitmq_management plugin is enabled.

1.4 Command Line Interface

Nameko ships with a command line interface to make hosting and interacting with services as easy as possible.

1.4.1 Running a Service

```
$ nameko run <module>[:<ServiceClass>]
```

Discover and run a service class. This will start the service in the foreground and run it until the process terminates.

It is possible to override the default settings using a `--config` switch

```
$ nameko run --config ./foobar.yaml <module>[:<ServiceClass>]
```

and providing a simple YAML configuration file:

```
# foobar.yaml

AMQP_URI: 'pyamqp://guest:guest@localhost'
WEB_SERVER_ADDRESS: '0.0.0.0:8000'
rpc_exchange: 'nameko-rpc'
max_workers: 10
parent_calls_tracked: 10

LOGGING:
    version: 1
    handlers:
        console:
            class: logging.StreamHandler
    root:
        level: DEBUG
        handlers: [console]
```

The `LOGGING` entry is passed to `logging.config.dictConfig()` and should conform to the schema for that call.

Config values can be read via the built-in *Config* dependency provider.

1.4.2 Environment variable substitution

YAML configuration files have support for environment variables. You can use bash style syntax: `${ENV_VAR}`. Optionally you can provide default values `${ENV_VAR:default_value}`. Default values can contains environment variables recursively `${ENV_VAR:default_${OTHER_ENV_VAR:value}}` (note: this feature requires regex package).

```
# foobar.yaml
AMQP_URI: pyamqp://${RABBITMQ_USER:guest}:${RABBITMQ_PASSWORD:password}@${RABBITMQ_HOST:localhost}
```

To run your service and set environment variables for it to use:

```
$ RABBITMQ_USER=user RABBITMQ_PASSWORD=password RABBITMQ_HOST=host nameko run --config ./foobar.yaml
```

If you need to quote the values in your YAML file, the explicit `!env_var` resolver is required:

```
# foobar.yaml
AMQP_URI: !env_var "pyamqp://${RABBITMQ_USER:guest}:${RABBITMQ_PASSWORD:password}@${RABBITMQ_HOST:lo
```

If you need to use values as raw strings in your YAML file without them getting converted to native python, the explicit `!raw_env_var` resolver is required:

```
# foobar.yaml
ENV_THAT_IS_NEEDED_RAW: !raw_env_var "${ENV_THAT_IS_NEEDED_RAW:1234.5660}"
```

This will turn into the string value `1234.5660`, instead of a float number.

You can provide many levels of default values

```
# foobar.yaml
AMQP_URI: ${AMQP_URI:pyamqp://${RABBITMQ_USER:guest}:${RABBITMQ_PASSWORD:password}@${RABBITMQ_HOST:lo
```

this config accepts AMQP_URI as an environment variable, if provided RABBITMQ_* nested variables will not be used.

The environment variable value is interpreted as YAML, so it is possible to use rich types:

```
# foobar.yaml
...
THINGS: ${A_LIST_OF_THINGS}
```

```
$ A_LIST_OF_THINGS=[A,B,C] nameko run --config ./foobar.yaml <module>[:<ServiceClass>]
```

the parser for environment variables will pair all brackets.

```
# foobar.yaml
LANDING_URL_TEMPLATE: ${LANDING_URL_TEMPLATE:https://example.com/{path}}
```

so the default value for this config will be https://example.com/{path}

1.4.3 Interacting with running services

```
$ nameko shell
```

Launch an interactive python shell for working with remote nameko services. This is a regular interactive interpreter, with a special module `n` added to the built-in namespace, providing the ability to make RPC calls and dispatch events.

Making an RPC call to "target_service":

```
$ nameko shell
>>> n.rpc.target_service.target_method(...)
# RPC response
```

Dispatching an event as "source_service":

```
$ nameko shell
>>> n.dispatch_event("source_service", "event_type", "event_payload")
```

1.5 Built-in Extensions

Nameko includes a number of built-in *extensions*. This section introduces them and gives brief examples of their usage.

1.5.1 RPC

Nameko includes an implementation of RPC over AMQP. It comprises the `@rpc` entrypoint, a proxy for services to talk to other services, and a standalone proxy that non-Nameko clients can use to make RPC calls to a cluster:

```
from nameko.rpc import rpc, RpcProxy

class ServiceY:
    name = "service_y"

    @rpc
    def append_identifier(self, value):
        return u"{}-y".format(value)


class ServiceX:
    name = "service_x"

    y = RpcProxy("service_y")

    @rpc
    def remote_method(self, value):
        res = u"{}-x".format(value)
        return self.y.append_identifier(res)
```

```
from nameko.standalone.rpc import ClusterRpcProxy

config = {
    'AMQP_URI': AMQP_URI  # e.g. "pyamqp://guest:guest@localhost"
}

with ClusterRpcProxy(config) as cluster_rpc:
    cluster_rpc.service_x.remote_method("hellø")  # "hellø-x-y"
```

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

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

In a cluster with more than one instance of the target service, RPC requests round-robin between instances. The request will be handled by exactly one instance of the target service.

AMQP messages are ack'd only after the request has been successfully processed. If the service fails to acknowledge the message and the AMQP connection is closed (e.g. if the service process is killed) the broker will revoke and then allocate the message to the available service instance.

Request and response payloads are serialized into JSON for transport over the wire.

1.5.2 Events (Pub-Sub)

Nameko Events is an asynchronous messaging system, implementing the Publish-Subscriber pattern. Services dispatch events that may be received by zero or more others:

```
from nameko.events import EventDispatcher, event_handler
from nameko.rpc import rpc

class ServiceA:
    """ Event dispatching service. """
    name = "service_a"

    dispatch = EventDispatcher()

    @rpc
    def dispatching_method(self, payload):
        self.dispatch("event_type", payload)


class ServiceB:
    """ Event listening service. """
    name = "service_b"

    @event_handler("service_a", "event_type")
    def handle_event(self, payload):
        print("service b received:", payload)
```

The `EventHandler` entrypoint has three `handler_types` that determine how event messages are received in a cluster:

- `SERVICE_POOL` – event handlers are pooled by service name and one instance from each pool receives the event, similar to the cluster behaviour of the RPC entrypoint. This is the default handler type.
- `BROADCAST` – every listening service instance will receive the event.
- `SINGLETON` – exactly one listening service instance will receive the event.

An example of using the `BROADCAST` mode:

```
from nameko.events import BROADCAST, event_handler

class ListenerService:
    name = "listener"

    @event_handler(
        "monitor", "ping", handler_type=BROADCAST, reliable_delivery=False
    )
    def ping(self, payload):
        # all running services will respond
        print("pong from {}".format(self.name))
```

Events are serialized into JSON for transport over the wire.

1.5.3 HTTP

The HTTP entrypoint is built on top of werkzeug, and supports all the standard HTTP methods (GET/POST/DELETE/PUT etc)

The HTTP entrypoint can specify multiple HTTP methods for a single URL as a comma-separated list. See example below.

Service methods must return one of:

- a string, which becomes the response body
- a 2-tuple `(status code, response body)`
- a 3-tuple `(status_code, headers dict, response body)`
- an instance of `werkzeug.wrappers.Response`

```
# http.py

import json
from nameko.web.handlers import http

class HttpService:
    name = "http_service"

    @http('GET', '/get/<int:value>')
    def get_method(self, request, value):
        return json.dumps({'value': value})

    @http('POST', '/post')
    def do_post(self, request):
        return u"received: {}".format(request.get_data(as_text=True))

    @http('GET,PUT,POST,DELETE', '/multi')
    def do_multi(self, request):
        return request.method
```

```
$ nameko run http
starting services: http_service
```

```
$ curl -i localhost:8000/get/42
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 13
Date: Fri, 13 Feb 2015 14:51:18 GMT

{'value': 42}
```

```
$ curl -i -d "post body" localhost:8000/post
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 19
Date: Fri, 13 Feb 2015 14:55:01 GMT

received: post body
```

A more advanced example:

```
# advanced_http.py

from nameko.web.handlers import http
from werkzeug.wrappers import Response

class Service:
    name = "advanced_http_service"
```

```
    @http('GET', '/privileged')
    def forbidden(self, request):
        return 403, "Forbidden"

    @http('GET', '/headers')
    def redirect(self, request):
        return 201, {'Location': 'https://www.example.com/widget/1'}, ""

    @http('GET', '/custom')
    def custom(self, request):
        return Response("payload")
```

```
$ nameko run advanced_http
starting services: advanced_http_service
```

```
$ curl -i localhost:8000/privileged
HTTP/1.1 403 FORBIDDEN
Content-Type: text/plain; charset=utf-8
Content-Length: 9
Date: Fri, 13 Feb 2015 14:58:02 GMT
```

```
curl -i localhost:8000/headers
HTTP/1.1 201 CREATED
Location: https://www.example.com/widget/1
Content-Type: text/plain; charset=utf-8
Content-Length: 0
Date: Fri, 13 Feb 2015 14:58:48 GMT
```

You can control formatting of errors returned from your service by overriding `response_from_exception()`:

```
import json
from nameko.web.handlers import HttpRequestHandler
from werkzeug.wrappers import Response
from nameko.exceptions import safe_for_serialization


class HttpError(Exception):
    error_code = 'BAD_REQUEST'
    status_code = 400


class InvalidArgumentsError(HttpError):
    error_code = 'INVALID_ARGUMENTS'


class HttpEntrypoint(HttpRequestHandler):
    def response_from_exception(self, exc):
        if isinstance(exc, HttpError):
            response = Response(
                json.dumps({
                    'error': exc.error_code,
                    'message': safe_for_serialization(exc),
                }),
                status=exc.status_code,
                mimetype='application/json'
            )
            return response
        return HttpRequestHandler.response_from_exception(self, exc)
```

```
http = HttpEntrypoint.decorator


class Service:
    name = "http_service"

    @http('GET', '/custom_exception')
    def custom_exception(self, request):
        raise InvalidArgumentsError("Argument `foo` is required.")
```

```
$ nameko run http_exceptions
starting services: http_service
```

```
$ curl -i http://localhost:8000/custom_exception
HTTP/1.1 400 BAD REQUEST
Content-Type: application/json
Content-Length: 72
Date: Thu, 06 Aug 2015 09:53:56 GMT

{"message": "Argument `foo` is required.", "error": "INVALID_ARGUMENTS"}
```

You can change the HTTP port and IP using the *WEB_SERVER_ADDRESS* config setting:

```
# foobar.yaml

AMQP_URI: 'pyamqp://guest:guest@localhost'
WEB_SERVER_ADDRESS: '0.0.0.0:8000'
```

1.5.4 Timer

The `Timer` is a simple entrypoint that fires once per a configurable number of seconds. The timer is not "cluster-aware" and fires on all services instances.

```
from nameko.timer import timer

class Service:
    name ="service"

    @timer(interval=1)
    def ping(self):
        # method executed every second
        print("pong")
```

1.6 Built-in Dependency Providers

Nameko includes some commonly used *dependency providers*. This section introduces them and gives brief examples of their usage.

1.6.1 Config

Config is a simple dependency provider that gives services read-only access to configuration values at run time, see *Running a Service*.

```
from nameko.dependency_providers import Config
from nameko.web.handlers import http


class Service:

    name = "test_config"

    config = Config()

    @property
    def foo_enabled(self):
        return self.config.get('FOO_FEATURE_ENABLED', False)

    @http('GET', '/foo')
    def foo(self, request):
        if not self.foo_enabled:
            return 403, "FeatureNotEnabled"

        return 'foo'
```

1.7 Community

There are a number of nameko extensions and supplementary libraries that are not part of the core project but that you may find useful when developing your own nameko services:

1.7.1 Extensions

- nameko-sqlalchemy

  A `DependencyProvider` for writing to databases with SQLAlchemy. Requires a pure-python or otherwise eventlet-compatible database driver.

  Consider combining it with SQLAlchemy-filters to add filtering, sorting and pagination of query objects when exposing them over a REST API.

- nameko-sentry

  Captures entrypoint exceptions and sends tracebacks to a Sentry server.

- nameko-amqp-retry

  Nameko extension allowing AMQP entrypoints to retry later.

- nameko-bayeux-client

  Nameko extension with a Cometd client implementing Bayeux protocol

- nameko-slack

  Nameko extension for interaction with Slack APIs. Uses Slack Developer Kit for Python.

- nameko-eventlog-dispatcher

  Nameko dependency provider that dispatches log data using Events (Pub-Sub).

- nameko-redis-py

  Redis dependency and utils for Nameko.

- nameko-redis

  Redis dependency for nameko services

- nameko-statsd

  A StatsD dependency for nameko, enabling services to send stats.

- nameko-twilio

  Twilio dependency for nameko, so you can send SMS, make calls, and answer calls in your service.

- nameko-sendgrid

  SendGrid dependency for nameko, for sending transactional and marketing emails.

- nameko-cachetools

  Tools to cache RPC interactions between your nameko services.

1.7.2 Supplementary Libraries

- django-nameko

  Django wrapper for Nameko microservice framework.

- flask_nameko

  A wrapper for using nameko services with Flask.

- nameko-proxy

  Standalone async proxy to communicate with Nameko microservices.

Search PyPi for more nameko packages

If you would like your own nameko extension or library to appear on this page, please *get in touch*.

1.8 Testing Services

1.8.1 Philosophy

Nameko's conventions are designed to make testing as easy as possible. Services are likely to be small and single-purpose, and dependency injection makes it simple to replace and isolate pieces of functionality.

The examples below use pytest, which is what Nameko's own test suite uses, but the helpers are test framework agnostic.

1.8.2 Unit Testing

Unit testing in Nameko usually means testing a single service in isolation – i.e. without any or most of its dependencies.

The `worker_factory()` utility will create a worker from a given service class, with its dependencies replaced by `mock.MagicMock` objects. Dependency functionality can then be imitated by adding `side_effects` and `return_values`:

```
""" Service unit testing best practice.
"""

from nameko.rpc import RpcProxy, rpc
from nameko.testing.services import worker_factory


class ConversionService(object):
    """ Service under test
    """
    name = "conversions"

    maths_rpc = RpcProxy("maths")

    @rpc
    def inches_to_cm(self, inches):
        return self.maths_rpc.multiply(inches, 2.54)

    @rpc
    def cms_to_inches(self, cms):
        return self.maths_rpc.divide(cms, 2.54)


def test_conversion_service():
    # create worker with mock dependencies
    service = worker_factory(ConversionService)

    # add side effects to the mock proxy to the "maths" service
    service.maths_rpc.multiply.side_effect = lambda x, y: x * y
    service.maths_rpc.divide.side_effect = lambda x, y: x / y

    # test inches_to_cm business logic
    assert service.inches_to_cm(300) == 762
    service.maths_rpc.multiply.assert_called_once_with(300, 2.54)

    # test cms_to_inches business logic
    assert service.cms_to_inches(762) == 300
    service.maths_rpc.divide.assert_called_once_with(762, 2.54)
```

In some circumstances it's helpful to provide an alternative dependency, rather than use a mock. This may be a fully functioning replacement (e.g. a test database session) or a lightweight shim that provides partial functionality.

```
""" Service unit testing best practice, with an alternative dependency.
"""

import pytest
from sqlalchemy import Column, Integer, String, create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

from nameko.rpc import rpc
from nameko.testing.services import worker_factory

# using community extension from http://pypi.python.org/pypi/nameko-sqlalchemy
from nameko_sqlalchemy import Session


Base = declarative_base()
```

```
class Result(Base):
    __tablename__ = 'model'
    id = Column(Integer, primary_key=True)
    value = Column(String(64))


class Service:
    """ Service under test
    """
    name = "service"

    db = Session(Base)

    @rpc
    def save(self, value):
        result = Result(value=value)
        self.db.add(result)
        self.db.commit()


@pytest.fixture
def session():
    """ Create a test database and session
    """
    engine = create_engine('sqlite:///:memory:')
    Base.metadata.create_all(engine)
    session_cls = sessionmaker(bind=engine)
    return session_cls()


def test_service(session):

    # create instance, providing the test database session
    service = worker_factory(Service, db=session)

    # verify ``save`` logic by querying the test database
    service.save("helloworld")
    assert session.query(Result.value).all() == [("helloworld",)]
```

1.8.3 Integration Testing

Integration testing in Nameko means testing the interface between a number of services. The recommended way is to run all the services being tested in the normal way, and trigger behaviour by "firing" an entrypoint using a helper:

```
""" Service integration testing best practice.
"""

from nameko.rpc import rpc, RpcProxy
from nameko.testing.utils import get_container
from nameko.testing.services import entrypoint_hook


class ServiceX:
    """ Service under test
    """
```

```
    name = "service_x"

    y = RpcProxy("service_y")

    @rpc
    def remote_method(self, value):
        res = "{}-x".format(value)
        return self.y.append_identifier(res)


class ServiceY:
    """ Service under test
    """
    name = "service_y"

    @rpc
    def append_identifier(self, value):
        return "{}-y".format(value)


def test_service_x_y_integration(runner_factory, rabbit_config):

    # run services in the normal manner
    runner = runner_factory(rabbit_config, ServiceX, ServiceY)
    runner.start()

    # artificially fire the "remote_method" entrypoint on ServiceX
    # and verify response
    container = get_container(runner, ServiceX)
    with entrypoint_hook(container, "remote_method") as entrypoint:
        assert entrypoint("value") == "value-x-y"
```

Note that the interface between `ServiceX` and `ServiceY` here is just as if under normal operation.

Interfaces that are out of scope for a particular test can be deactivated with one of the following test helpers:

restrict_entrypoints

nameko.testing.services.**restrict_entrypoints**(*container*, **entrypoints*)

Restrict the entrypoints on `container` to those named in `entrypoints`.

This method must be called before the container is started.

Usage

The following service definition has two entrypoints:

```
    class Service(object):
        name = "service"

        @timer(interval=1)
        def foo(self, arg):
            pass

        @rpc
        def bar(self, arg)
            pass

        @rpc
```

```
        def baz(self, arg):
            pass

    container = ServiceContainer(Service, config)
```

To disable the timer entrypoint on `foo`, leaving just the RPC entrypoints:

```
    restrict_entrypoints(container, "bar", "baz")
```

Note that it is not possible to identify multiple entrypoints on the same method individually.

replace_dependencies

nameko.testing.services.**replace_dependencies**(*container*, **dependencies*, ***dependency_map*)

Replace the dependency providers on `container` with instances of `MockDependencyProvider`.

Dependencies named in *dependencies will be replaced with a `MockDependencyProvider`, which injects a MagicMock instead of the dependency.

Alternatively, you may use keyword arguments to name a dependency and provide the replacement value that the *MockDependencyProvider* should inject.

Return the `MockDependencyProvider.dependency` for every dependency specified in the (*dependencies) args so that calls to the replaced dependencies can be inspected. Return a single object if only one dependency was replaced, and a generator yielding the replacements in the same order as `dependencies` otherwise. Note that any replaced dependencies specified via kwargs **dependency_map* will not be returned.

Replacements are made on the container instance and have no effect on the service class. New container instances are therefore unaffected by replacements on previous instances.

Usage

```
    from nameko.rpc import RpcProxy, rpc
    from nameko.standalone.rpc import ServiceRpcProxy

    class ConversionService(object):
        name = "conversions"

        maths_rpc = RpcProxy("maths")

        @rpc
        def inches_to_cm(self, inches):
            return self.maths_rpc.multiply(inches, 2.54)

        @rpc
        def cm_to_inches(self, cms):
            return self.maths_rpc.divide(cms, 2.54)

    container = ServiceContainer(ConversionService, config)
    mock_maths_rpc = replace_dependencies(container, "maths_rpc")
    mock_maths_rpc.divide.return_value = 39.37

    container.start()

    with ServiceRpcProxy('conversions', config) as proxy:
        proxy.cm_to_inches(100)
```

```
    # assert that the dependency was called as expected
    mock_maths_rpc.divide.assert_called_once_with(100, 2.54)
```

Providing a specific replacement by keyword:

```
    class StubMaths(object):

        def divide(self, val1, val2):
            return val1 / val2

    replace_dependencies(container, maths_rpc=StubMaths())

    container.start()

    with ServiceRpcProxy('conversions', config) as proxy:
        assert proxy.cm_to_inches(127) == 50.0
```

Complete Example

The following integration testing example makes use of both scope-restricting helpers:

```
"""
This file defines several toy services that interact to form a shop of the
famous ACME Corporation. The AcmeShopService relies on the StockService,
InvoiceService and PaymentService to fulfil its orders. They are not best
practice examples! They're minimal services provided for the test at the
bottom of the file.

``test_shop_integration`` is a full integration test of the ACME shop
"checkout flow". It demonstrates how to test the multiple ACME services in
combination with each other, including limiting service interactions by
replacing certain entrypoints and dependencies.
"""

from collections import defaultdict

import pytest

from nameko.extensions import DependencyProvider
from nameko.events import EventDispatcher, event_handler
from nameko.exceptions import RemoteError
from nameko.rpc import rpc, RpcProxy
from nameko.standalone.rpc import ServiceRpcProxy
from nameko.testing.services import replace_dependencies, restrict_entrypoints
from nameko.testing.utils import get_container
from nameko.timer import timer


class NotLoggedInError(Exception):
    pass


class ItemOutOfStockError(Exception):
    pass


class ItemDoesNotExistError(Exception):
```

```
    pass


class ShoppingBasket(DependencyProvider):
    """ A shopping basket tied to the current ``user_id``.
    """
    def __init__(self):
        self.baskets = defaultdict(list)

    def get_dependency(self, worker_ctx):

        class Basket(object):
            def __init__(self, basket):
                self._basket = basket
                self.worker_ctx = worker_ctx

            def add(self, item):
                self._basket.append(item)

            def __iter__(self):
                for item in self._basket:
                    yield item

        try:
            user_id = worker_ctx.data['user_id']
        except KeyError:
            raise NotLoggedInError()
        return Basket(self.baskets[user_id])


class AcmeShopService:
    name = "acmeshopservice"

    user_basket = ShoppingBasket()
    stock_rpc = RpcProxy('stockservice')
    invoice_rpc = RpcProxy('invoiceservice')
    payment_rpc = RpcProxy('paymentservice')

    fire_event = EventDispatcher()

    @rpc
    def add_to_basket(self, item_code):
        """ Add item identified by ``item_code`` to the shopping basket.

        This is a toy example! Ignore the obvious race condition.
        """
        stock_level = self.stock_rpc.check_stock(item_code)
        if stock_level > 0:
            self.user_basket.add(item_code)
            self.fire_event("item_added_to_basket", item_code)
            return item_code

        raise ItemOutOfStockError(item_code)

    @rpc
    def checkout(self):
        """ Take payment for all items in the shopping basket.
        """
```

```
        total_price = sum(self.stock_rpc.check_price(item)
                          for item in self.user_basket)

        # prepare invoice
        invoice = self.invoice_rpc.prepare_invoice(total_price)

        # take payment
        self.payment_rpc.take_payment(invoice)

        # fire checkout event if prepare_invoice and take_payment succeeded
        checkout_event_data = {
            'invoice': invoice,
            'items': list(self.user_basket)
        }
        self.fire_event("checkout_complete", checkout_event_data)
        return total_price


class Warehouse(DependencyProvider):
    """ A database of items in the warehouse.

    This is a toy example! A dictionary is not a database.
    """
    def __init__(self):
        self.database = {
            'anvil': {
                'price': 100,
                'stock': 3
            },
            'dehydrated_boulders': {
                'price': 999,
                'stock': 12
            },
            'invisible_paint': {
                'price': 10,
                'stock': 30
            },
            'toothpicks': {
                'price': 1,
                'stock': 0
            }
        }

    def get_dependency(self, worker_ctx):
        return self.database


class StockService:
    name = "stockservice"

    warehouse = Warehouse()

    @rpc
    def check_price(self, item_code):
        """ Check the price of an item.
        """
        try:
            return self.warehouse[item_code]['price']
```

```
        except KeyError:
            raise ItemDoesNotExistError(item_code)

    @rpc
    def check_stock(self, item_code):
        """ Check the stock level of an item.
        """
        try:
            return self.warehouse[item_code]['stock']
        except KeyError:
            raise ItemDoesNotExistError(item_code)

    @rpc
    @timer(100)
    def monitor_stock(self):
        """ Periodic stock monitoring method. Can also be triggered manually
        over RPC.

        This is an expensive process that we don't want to exercise during
        integration testing...
        """
        raise NotImplementedError()

    @event_handler('acmeshopservice', "checkout_complete")
    def dispatch_items(self, event_data):
        """ Dispatch items from stock on successful checkouts.

        This is an expensive process that we don't want to exercise during
        integration testing...
        """
        raise NotImplementedError()


class AddressBook(DependencyProvider):
    """ A database of user details, keyed on user_id.
    """
    def __init__(self):
        self.address_book = {
            'wile_e_coyote': {
                'username': 'wile_e_coyote',
                'fullname': 'Wile E Coyote',
                'address': '12 Long Road, High Cliffs, Utah',
            },
        }

    def get_dependency(self, worker_ctx):
        def get_user_details():
            try:
                user_id = worker_ctx.data['user_id']
            except KeyError:
                raise NotLoggedInError()
            return self.address_book.get(user_id)
        return get_user_details


class InvoiceService:
    name = "invoiceservice"
```

```
    get_user_details = AddressBook()

    @rpc
    def prepare_invoice(self, amount):
        """ Prepare an invoice for ``amount`` for the current user.
        """
        address = self.get_user_details().get('address')
        fullname = self.get_user_details().get('fullname')
        username = self.get_user_details().get('username')

        msg = "Dear {}. Please pay ${} to ACME Corp.".format(fullname, amount)
        invoice = {
            'message': msg,
            'amount': amount,
            'customer': username,
            'address': address
        }
        return invoice


class PaymentService:
    name = "paymentservice"

    @rpc
    def take_payment(self, invoice):
        """ Take payment from a customer according to ``invoice``.

        This is an expensive process that we don't want to exercise during
        integration testing...
        """
        raise NotImplementedError()

# ============================================================================
# Begin test
# ============================================================================


@pytest.yield_fixture
def rpc_proxy_factory(rabbit_config):
    """ Factory fixture for standalone RPC proxies.

    Proxies are started automatically so they can be used without a ``with``
    statement. All created proxies are stopped at the end of the test, when
    this fixture closes.
    """
    all_proxies = []

    def make_proxy(service_name, **kwargs):
        proxy = ServiceRpcProxy(service_name, rabbit_config, **kwargs)
        all_proxies.append(proxy)
        return proxy.start()

    yield make_proxy

    for proxy in all_proxies:
        proxy.stop()
```

```
def test_shop_checkout_integration(
    rabbit_config, runner_factory, rpc_proxy_factory
):
    """ Simulate a checkout flow as an integration test.

    Requires instances of AcmeShopService, StockService and InvoiceService
    to be running. Explicitly replaces the rpc proxy to PaymentService so
    that service doesn't need to be hosted.

    Also replaces the event dispatcher dependency on AcmeShopService and
    disables the timer entrypoint on StockService. Limiting the interactions
    of services in this way reduces the scope of the integration test and
    eliminates undesirable side-effects (e.g. processing events unnecessarily).
    """
    context_data = {'user_id': 'wile_e_coyote'}
    shop = rpc_proxy_factory('acmeshopservice', context_data=context_data)

    runner = runner_factory(
        rabbit_config, AcmeShopService, StockService, InvoiceService)

    # replace ``event_dispatcher`` and ``payment_rpc``  dependencies on
    # AcmeShopService with ``MockDependencyProvider``\s
    shop_container = get_container(runner, AcmeShopService)
    fire_event, payment_rpc = replace_dependencies(
        shop_container, "fire_event", "payment_rpc")

    # restrict entrypoints on StockService
    stock_container = get_container(runner, StockService)
    restrict_entrypoints(stock_container, "check_price", "check_stock")

    runner.start()

    # add some items to the basket
    assert shop.add_to_basket("anvil") == "anvil"
    assert shop.add_to_basket("invisible_paint") == "invisible_paint"

    # try to buy something that's out of stock
    with pytest.raises(RemoteError) as exc_info:
        shop.add_to_basket("toothpicks")
    assert exc_info.value.exc_type == "ItemOutOfStockError"

    # provide a mock response from the payment service
    payment_rpc.take_payment.return_value = "Payment complete."

    # checkout
    res = shop.checkout()

    total_amount = 100 + 10
    assert res == total_amount

    # verify integration with mocked out payment service
    payment_rpc.take_payment.assert_called_once_with({
        'customer': "wile_e_coyote",
        'address': "12 Long Road, High Cliffs, Utah",
        'amount': total_amount,
        'message': "Dear Wile E Coyote. Please pay $110 to ACME Corp."
    })
```

```
    # verify events fired as expected
    assert fire_event.call_count == 3


if __name__ == "__main__":
    import sys
    pytest.main(sys.argv)
```

1.8.4 Other Helpers

entrypoint_hook

The entrypoint hook allows a service entrypoint to be called manually. This is useful during integration testing if it is difficult or expensive to fake to external event that would cause an entrypoint to fire.

You can provide *context_data* for the call to mimic specific call context, for example language, user agent or auth token.

```
import pytest

from nameko.contextdata import Language
from nameko.rpc import rpc
from nameko.testing.services import entrypoint_hook


class HelloService:
    """ Service under test
    """
    name = "hello_service"

    language = Language()

    @rpc
    def hello(self, name):
        greeting = "Hello"
        if self.language == "fr":
            greeting = "Bonjour"
        elif self.language == "de":
            greeting = "Gutentag"

        return "{}, {}!".format(greeting, name)


@pytest.mark.parametrize("language, greeting", [
    ("en", "Hello"),
    ("fr", "Bonjour"),
    ("de", "Gutentag"),
])
def test_hello_languages(language, greeting, container_factory, rabbit_config):

    container = container_factory(HelloService, rabbit_config)
    container.start()

    context_data = {'language': language}
    with entrypoint_hook(container, 'hello', context_data) as hook:
        assert hook("Matt") == "{}, Matt!".format(greeting)
```

entrypoint_waiter

The entrypoint waiter is a context manager that does not exit until a named entrypoint has fired and completed. This is useful when testing integration points between services that are asynchronous, for example receiving an event:

```
from nameko.events import event_handler
from nameko.standalone.events import event_dispatcher
from nameko.testing.services import entrypoint_waiter


class ServiceB:
    """ Event listening service.
    """
    name = "service_b"

    @event_handler("service_a", "event_type")
    def handle_event(self, payload):
        print("service b received", payload)


def test_event_interface(container_factory, rabbit_config):

    container = container_factory(ServiceB, rabbit_config)
    container.start()

    dispatch = event_dispatcher(rabbit_config)

    # prints "service b received payload" before "exited"
    with entrypoint_waiter(container, 'handle_event'):
        dispatch("service_a", "event_type", "payload")
    print("exited")
```

Note that the context manager waits not only for the entrypoint method to complete, but also for any dependency teardown. Dependency-based loggers such as (TODO: link to bundled logger) for example would have also completed.

1.8.5 Using pytest

Nameko's test suite uses pytest, and makes some useful configuration and fixtures available for your own tests if you choose to use pytest.

They are contained in `nameko.testing.pytest`. This module is automatically registered as a pytest plugin by setuptools if you have pytest installed.

1.9 Writing Extensions

1.9.1 Structure

Extensions should subclass `nameko.extensions.Extension`. This base class provides the basic structure for an extension, in particular the following methods which can be overridden to add functionality:

Binding

An extension in a service class is merely a declaration. When a service is *hosted* by a *service container* its extensions are "bound" to the container.
The binding process is transparent to a developer writing new extensions. The only consideration should be that `__init__()` is called in `bind()` as well as at service class declaration time, so you should avoid side-effects in this method and use `setup()` instead.

`Extension.`**`setup`**`()`
Called on bound Extensions before the container starts.

Extensions should do any required initialisation here.

`Extension.`**`start`**`()`
Called on bound Extensions when the container has successfully started.

This is only called after all other Extensions have successfully returned from `Extension.setup()`. If the Extension reacts to external events, it should now start acting upon them.

`Extension.`**`stop`**`()`
Called when the service container begins to shut down.

Extensions should do any graceful shutdown here.

1.9.2 Writing Dependency Providers

Almost every Nameko application will need to define its own dependencies – maybe to interface with a database for which there is no *community extension* or to communicate with a specific web service.

Dependency providers should subclass `nameko.extensions.DependencyProvider` and implement a `get_dependency()` method that returns the object to be injected into service workers.

The recommended pattern is to inject the minimum required interface for the dependency. This reduces the test surface and makes it easier to exercise service code under test.

Dependency providers may also hook into the *worker lifecycle*. The following three methods are called on all dependency providers for every worker:

`DependencyProvider.`**`worker_setup`**`(`*worker_ctx*`)`
Called before a service worker executes a task.

Dependencies should do any pre-processing here, raising exceptions in the event of failure.

Example: ...

Parameters

worker_ctx [WorkerContext] See `nameko.containers.ServiceContainer.spawn_worker`

`DependencyProvider.`**`worker_result`**`(`*worker_ctx*, *result=None*, *exc_info=None*`)`
Called with the result of a service worker execution.

Dependencies that need to process the result should do it here. This method is called for all *Dependency* instances on completion of any worker.

Example: a database session dependency may flush the transaction

Parameters

worker_ctx [WorkerContext] See `nameko.containers.ServiceContainer.spawn_worker`

`DependencyProvider.`**`worker_teardown`**`(`*`worker_ctx`*`)`

Called after a service worker has executed a task.

Dependencies should do any post-processing here, raising exceptions in the event of failure.

Example: a database session dependency may commit the session

Parameters

worker_ctx [WorkerContext] See `nameko.containers.ServiceContainer.spawn_worker`

Concurrency and Thread-Safety

The object returned by `get_dependency()` should be thread-safe, because it may be accessed by multiple concurrently running workers.

The *worker lifecycle* are called in the same thread that executes the service method. This means, for example, that you can define thread-local variables and access them from each method.

Example

A simple `DependencyProvider` that sends messages to an SQS queue.

```
from nameko.extensions import DependencyProvider

import boto3

class SqsSend(DependencyProvider):

    def __init__(self, url, region="eu-west-1", **kwargs):
        self.url = url
        self.region = region
        super(SqsSend, self).__init__(**kwargs)

    def setup(self):
        self.client = boto3.client('sqs', region_name=self.region)

    def get_dependency(self, worker_ctx):

        def send_message(payload):
            # assumes boto client is thread-safe for this action, because it
            # happens inside the worker threads
            self.client.send_message(
                QueueUrl=self.url,
                MessageBody=payload
            )
        return send_message
```

1.9.3 Writing Entrypoints

You can implement new Entrypoint extensions if you want to support new transports or mechanisms for initiating service code.

The minimum requirements for an Entrypoint are:

1. Inherit from `nameko.extensions.Entrypoint`

2. Implement the `start()` method to start the entrypoint when the container does. If a background thread is required, it's recommended to use a thread managed by the service container (see *Spawning Background Threads*)
3. Call `spawn_worker()` on the bound container when appropriate.

Example

A simple `Entrypoint` that receives messages from an SQS queue.

```
from nameko.extensions import Entrypoint
from functools import partial

import boto3


class SqsReceive(Entrypoint):

    def __init__(self, url, region="eu-west-1", **kwargs):
        self.url = url
        self.region = region
        super(SqsReceive, self).__init__(**kwargs)

    def setup(self):
        self.client = boto3.client('sqs', region_name=self.region)

    def start(self):
        self.container.spawn_managed_thread(
            self.run, identifier="SqsReceiver.run"
        )

    def run(self):
        while True:
            response = self.client.receive_message(
                QueueUrl=self.url,
                WaitTimeSeconds=5,
            )
            messages = response.get('Messages', ())
            for message in messages:
                self.handle_message(message)

    def handle_message(self, message):
        handle_result = partial(self.handle_result, message)

        args = (message['Body'],)
        kwargs = {}

        self.container.spawn_worker(
            self, args, kwargs, handle_result=handle_result
        )

    def handle_result(self, message, worker_ctx, result, exc_info):
        # assumes boto client is thread-safe for this action, because it
        # happens inside the worker threads
        self.client.delete_message(
            QueueUrl=self.url,
            ReceiptHandle=message['ReceiptHandle']
        )
```

```
        return result, exc_info


receive = SqsReceive.decorator
```

Used in a service:

```
from .sqs_receive import receive


class SqsService(object):
    name = "sqs-service"

    @receive('https://sqs.eu-west-1.amazonaws.com/123456789012/nameko-sqs')
    def handle_sqs_message(self, body):
        """ This method is called by the `receive` entrypoint whenever
        a message sent to the given SQS queue.
        """
        print(body)
        return body
```

Expected Exceptions

The Entrypoint base class constructor will accept a list of classes that should be considered to be "expected" if they are raised by the decorated service method. This can used to differentiate *user errors* from more fundamental execution errors. For example:

```
class Service:
    name = "service"

    auth = Auth()

    @rpc(expected_exceptions=Unauthorized)
    def update(self, data):
        if not self.auth.has_role("admin"):
            raise Unauthorized()

        # perform update
        raise TypeError("Whoops, genuine error.")
```

The list of expected exceptions are saved to the Entrypoint instance so they can later be inspected, for example by other extensions that process exceptions, as in nameko-sentry.

Sensitive Arguments

In the same way as *expected exceptions*, the Entrypoint constructor allows you to mark certain arguments or parts of arguments as sensitive. For example:

```
class Service:
    name = "service"

    auth = Auth()

    @rpc(sensitive_arguments="password", expected_exceptions=Unauthenticated)
    def login(self, username, password):
```

```
        # raises Unauthenticated if username/password do not match
        return self.auth.authenticate(username, password)
```

This can to be used in combination with the utility function `nameko.utils.get_redacted_args()`, which will return an entrypoint's call args (similar to `inspect.getcallargs()`) but with sensitive elements redacted.

This is useful in Extensions that log or save information about entrypoint invocations, such as nameko-tracer.

For entrypoints that accept sensitive information nested within an otherwise safe argument, you can specify partial redaction. For example:

```
# by dictionary key
@entrypoint(sensitive_arguments="foo.a")
def method(self, foo):
    pass

>>> get_redacted_args(method, foo={'a': 1, 'b': 2})
... {'foo': {'a': '******', 'b': 2}}

# list index
@entrypoint(sensitive_arguments="foo.a[1]")
def method(self, foo):
    pass

>>> get_redacted_args(method, foo=[{'a': [1, 2, 3]}])
... {'foo': {'a': [1, '******', 3]}}
```

Slices and relative list indexes are not supported.

1.9.4 Spawning Background Threads

Extensions needing to execute work in a thread may choose to delegate the management of that thread to the service container using `spawn_managed_thread()`.

```
    def start(self):
        self.container.spawn_managed_thread(
            self.run, identifier="SqsReceiver.run"
        )
```

Delegating thread management to the container is advised because:

- Managed threads will always be terminated when the container is stopped or killed.
- Unhandled exceptions in managed threads are caught by the container and will cause it to terminate with an appropriate message, which can prevent hung processes.

CHAPTER 2

More Information

2.1 About Microservices

> An approach to designing software as a suite of small services, each running in its own process and communicating with lightweight mechanisms.
>
> —Martin Fowler, Microservices Architecture

Microservices are usually described in contrast to a "monolith" – an application built as a single unit where changes to any part of it require building and deploying the whole thing.

With microservices, functionality is instead split into "services" with well defined boundaries. Each of these services can be developed and deployed individually.

There are many benefits as well as drawbacks to using microservices, eloquently explained in Martin Fowler's paper. Not all of them always apply, so below we'll outline some that are relevant to Nameko.

2.1.1 Benefits

- Small and single-purpose

  Breaking a large application into smaller loosely-coupled chunks reduces the cognitive burden of working with it. A developer working on one service isn't required to understand the internals of the rest of the application; they can rely on a higher-level interface of the other services.

  This is harder to achieve in a monolithic application where the boundaries and interfaces between "chunks" are fuzzier.

- Explicit published interface

  The entrypoints for a Nameko service explicitly declare its published interface. This is the boundary between the service and its callers, and thus the point beyond which backwards compatibility must be considered or maintained.

- Individually deployable

  Unlike a monolith which can only be released all at once, Nameko services can be individually deployed. A change in one service can be made and rolled out without touching any of the others. Long running and highly considered release cycles can be broken into smaller, faster iterations.

- Specialization

  Decoupled chunks of application are free to use specialized libraries and dependencies. Where a monolith might be forced to choose a one-size-fits-all library, microservices are unshackled from the choices made by the rest of the application.

2.1.2 Drawbacks

- Overhead

  RPC calls are more expensive than in-process method calls. Processes will spend a lot of time waiting on I/O. Nameko mitigates wastage of CPU cycles with concurrency and eventlet, but the latency of each call will be longer than in a monolithic application.

- Cross-service transactions

  Distributing transactions between multiple processes is difficult to the point of futility. Microservice architectures work around this by changing the APIs they expose (see below) or only offering eventual consistency.

- Coarse-grained APIs

  The overhead and lack of transactions between service calls encourages coarser APIs. Crossing service boundaries is expensive and non-atomic.

  Where in a monolithic application you might write code that makes many calls to achieve a certain goal, a microservices architecture will encourage you to write fewer, heavier calls to be atomic or minimize overhead.

- Understanding interdependencies

  Splitting an application over multiple separate components introduces the requirement to understand how those components interact. This is hard when the components are in different code bases (and developer head spaces).

  In the future we hope to include tools in Nameko that make understanding, documenting and visualizing service interdependencies easier.

2.1.3 Further Notes

Microservices can be adopted incrementally. A good approach to building a microservices architecture is to start by pulling appropriate chunks of logic out of a monolithic application and turning them into individual services.

2.2 Benefits of Dependency Injection

The dependency injection pattern in nameko facilitates a separation of concerns between component parts of a service. There is a natural division between "service code" – application logic tied to the *single-purpose* of the service – and the rest of the code required for the service to operate.

Say you had a caching service that was responsible for reading and writing from memcached, and included some business-specific invalidation rules. The invalidation rules are clearly application logic, whereas the messy network interface with memcached can be abstracted away into a dependency.

Separating these concerns makes it easier to test service code in isolation. That means you don't need to have a memcached cluster available when you test your caching service. Furthermore it's easy to specify mock responses from the memcached cluster to test invalidation edge cases.

Separation also stops test scopes bleeding into one another. Without a clear interface between the caching service and the machinery it uses to communicate with memcached, it would be tempting to cover network-glitch edge cases as part of the caching service test suite. In fact the tests for this scenario should be as part of the test suite for the memcached dependency. This becomes obvious when dependencies are used by more than one service – without a separation you would have to duplicate the network-glitch tests or seem to have holes in your test coverage.

A more subtle benefit manifests in larger teams. Dependencies tend to encapsulate the thorniest and most complex aspects of an application. Whereas service code is stateless and single-threaded, dependencies must deal with concurrency and thread-safety. This can be a helpful division of labour between junior and senior developers.

Dependencies separate common functionality away from bespoke application logic. They can be written once and re-used by many services. Nameko's *community extensions* aims to promote sharing even between teams.

2.3 Similar and Related Projects

2.3.1 Celery

Celery is a distributed task queue. It lets you define "tasks" as Python functions and execute them on a set of remote workers, not unlike Nameko RPC.

Celery is usually used as an add-on to existing applications, to defer processing or otherwise outsource some work to a remote machine. You could also achieve this with Nameko, but Celery includes many more primitives for the distribution of tasks and the collection of results.

2.3.2 Zato

Zato is a full Enterprise Service Bus (ESB) and application server written in Python. It concentrates on joining lots of different services together, including APIs and a GUI for configuration. It also includes tools for load-balancing and deployment.

ESBs are often used as middleware between legacy services. You can write new Python services in Zato but they are structured quite differently and its scope is significantly larger than that of Nameko. See Martin Fowler's paper on microservices for a comparison to ESBs.

2.3.3 Kombu

Kombu is a Python messaging library, used by both Celery and Nameko. It exposes a high-level interface for AMQP and includes support for "virtual" transports, so can be run with non-AMQP transports such as Redis, ZeroMQ and MongoDB.

Nameko's AMQP features are built using Kombu but don't include support for the "virtual" transports.

Also, due to the usage of eventlet for green concurrency, Nameko can't make use of C-extensions such as librabbitmq that Kombu uses by default if it's available. If you want to have librabbitmq in your environment for other purposes than Nameko, you can force Kombu to use standard Python implementation of AMQP by defining broker urls as `pyamqp://` instead of `amqp://`

2.3.4 Eventlet

Eventlet is a Python library that provides concurrency via "greenthreads". You can check more details on how it's used by Nameko in the *Concurrency* section.

2.4 Getting in touch

If you have questions for the Nameko community or developers, there are a number of ways to get in touch:

2.4.1 GitHub

To raise a bug or issue against the project, visit the GitHub page.

2.4.2 Mailing list

For help, comments or questions, please use the mailing list on google groups.

2.5 Contributing

Nameko is developed on GitHub and contributions are welcome.

Use GitHub issues to report bugs and make feature requests.

You're welcome to fork the repository and raise a pull request with your contributions.

You can install all the development dependencies using:

```
pip install -e .[dev]
```

and the requirements for building docs using:

```
pip install -e .[docs]
```

Pull requests are automatically built with Travis-CI. Travis will fail the build unless all of the following are true:

- All tests pass
- 100% line coverage by tests
- Documentation builds successfully (including spell-check)

See *getting in touch* for more guidance on contributing.

2.5.1 Running the tests

There is a Makefile with convenience commands for running the tests. To run them locally you must have RabbitMQ *installed* and running, then call:

```
$ make test
```

2.6 License

Nameko is licensed under the Apache License, Version 2.0. Please see LICENSE in the project root for details.

2.7 Release Notes

Here you can see the full list of changes between nameko versions. Versions are in form of *headline.major.minor* numbers. Backwards-compatible changes increment the minor version number only.

2.7.1 Version 2.12.0

Released: 2019-03-18

- Refactor utils so standalone.events does not import eventlet (#580)
- Compatibility with latest dependencies (moto #577, pyyaml and kombu #612)
- Timer now waits for the spawned entrypoint to complete before firing again, as documented (#579, #303)
- Timer is also improved to avoid drift (#614)
- Hide password in logged amqp uri (#582)
- Docs updates (#587, #591, #276, #596)
- Nameko shell changed to not catch exceptions when used in non-TTY mode (#597)

2.7.2 Version 2.11.0

Released: 2018-08-09

- Compatibility with kombu 4 and pyamqp 2+, minimum supported kombu version is now 4.2 (#564)

2.7.3 Version 2.10.0

Released: 2018-08-06

- Bump the minimum supported eventlet version to 0.20.1 (#557)

2.7.4 Version 2.9.1

Released: 2018-07-20

- SSL connections now supported by all AMQP extensions, configurable using the *AMQP_SSL* config key. (#524)
- Restore compatibility with eventlet 0.22+ (#556)
- Log unhandled worker exceptions at *ERROR* level instead of *INFO* (#547)

2.7.5 Version 2.9.0

Released: 2018-05-30

- RPC reply queues are now set to expire rather than auto-delete, and are no longer exclusive, allowing clients reconnect. Fixes #359.
- It's now possible to accept messages in multiple serialization formats. Adds config-based mechanism for specifying custom serializers. See #535.
- Enhanced environment variable substitution including recursive references. See #515.

2.7.6 Version 2.8.5

Released: 2018-03-15

- Workaround for a Kombu bug causing new sockets to sometimes have short timeouts. (#521)

2.7.7 Version 2.8.4

Released: 2018-02-18

- Fixes a bug where the container crashed if the connection to RabbitMQ was lost while an AMQP entrypoint was running (#511)
- Correction to WorkerContext.immediate_call_id which actually referred to the id of original call. Adds WorkerContext.origin_call_id to replicate the previous behaviour.

2.7.8 Version 2.8.3

Released: 2018-01-12

- Restrict eventlet to *<0.22.0* until we're compatible.

2.7.9 Version 2.8.2

Released: 2017-12-11

- Remove the pytest *–log-level* argument added by the pytest plugin since this conflicts with newer versions of *pytest* (>= *3.3.0*). For older versions this can be restored by installing the *pytest-catchlog* package.

2.7.10 Version 2.8.1

Released: 2017-11-29

- Added the 'show-config' command which will print the service configuration to the console, after environment variable substitution.

2.7.11 Version 2.8.0

Released: 2017-10-31

- Environment variables substituted into config files are now interpreted as YAML rather than bare values, allowing use of rich data types.

2.7.12 Version 2.7.0

Released: 2017-10-07

- Set stopped flag in register_provider() to allow PollingQueueConsumer object reuse [fixes #462]
- Refactor of AMQP message publishing logic into *nameko.amqp.publish*
- Exposes delivery options and other messaging configuration to AMQP-based DependencyProviders. [addresses #374]
- Class attributes for configuring *use_confirms*, *retry* and *retry_policy* have been deprecated from the Publisher, EventDispatcher, and RPC MethodProxy classes. If you were subclassing these classes to set these options, you should now set them at class instantiation time.

2.7.13 Version 2.6.0

Released: 2017-04-30

- Environment variables are now interpreted as native YAML data types rather than just strings
- The WSGI Server now uses an explicit logger so it can be controlled using a logging config
- Drops several backwards-compatibility shims that were marked as being maintained only until this release

2.7.14 Version 2.5.4

Released: 2017-04-20

- Don't block the QueueConsumer thread on the worker pool, which could cause service deadlock and occasional dropped AMQP connections [fixes #428]
- Revert prefetch count change from 2.5.3 and ack messages outside of the QueueConsumer thread. [fixes #417 more robustly]

2.7.15 Version 2.5.3

Released: 2017-03-16

- Bump the amqp *prefetch_count* by 1 to *max_workers + 1* to fix throughput issue. [fixes 417]

2.7.16 Version 2.5.2

Released 2017-02-28

- Improves teardown speed of the *rabbit_config* pytest fixture
- Support for providing an alternative reply listener to the standalone RPC proxy

2.7.17 Version 2.5.1

Released 2017-01-19

- Adds a DependencyProvider to give services access to the config object
- Internal refactor to make all worker lifecycle steps run in the same thread

2.7.18 Version 2.5.0

Released 2016-12-20

- Enables publish confirms by default for all AMQP message publishers
- Refactors common AMQP connection code into *nameko.amqp*

2.7.19 Version 2.4.4

Released 2016-11-28

- Adds AMQP heartbeats to Consumer connections
- Handles an uncaught exception caused by a fast-disconnecting client under Python 3 [fixes #367]

2.7.20 Version 2.4.3

Released 2016-11-16

- Pins kombu back to a compatible release (<4) [fixes #378]
- Fixes compatibility with latest bpython and ipython shells [fixes #355 and #375]
- Fixes socket cleanup bug in websocket hub [fixes #367]

2.7.21 Version 2.4.2

Released 2016-10-10

- Added support for environment variables in YAML config files
- Enhanced `entrypoint_waiter()`. The new implementation is backwards compatible but additionally:
  - Gives you access to the result returned (or exception raised)
  - Adds the ability to wait for a specific result
  - Doesn't fire until the worker is completely torn down

2.7.22 Version 2.4.1

Released 2016-09-14

- Enhanced :class: *~nameko.web.server.WebServer* with *get_wsgi_app* and *get_wsgi_server* to allow easy usage of WSGI middleware and modifications of the WSGI server.
- Enhanced `replace_dependencies()` to allow specific replacement values to be provided with named arguments.

2.7.23 Version 2.4.0

Released 2016-08-30

- Add dictionary access to `standalone.rpc.ClusterProxy` to allow the proxy to call services whose name is not a legal identifier in python (e.g. name has a - in it).
- Add the ability to specify a custom *ServiceContainer* class via config key. Deprecate the keyword arguments to *ServiceRunner* and *run_services* for the same purpose.
- Deprecate the keyword arguments to *run_services*, *ServiceContainer* and *ServiceRunner.add_service* for specifying a custom *WorkerContext* class. Custom *WorkerContext* classes can now only be specified with a custom *ServiceContainer* class that defines the *worker_ctx_cls* attribute.
- Remove the *context_keys* attribute of the *WorkerContext*, which was previously used to "whitelist" worker context data passed from call to call. It was a feature that leaked from a specific implementation into the main framework, and not useful enough in its own right to continue to be supported.

- Refactor *ServiceContainer* internals for better separation between "managed" and "worker" threads. Improved logging when threads are killed.

2.7.24 Version 2.3.1

Released 2016-05-11

- Deprecate `MethodProxy.async` in favour of `MethodProxy.call_async` in preparation for async becoming a keyword
- Add support for loading logging configuration from `config.yaml`

2.7.25 Version 2.3.0

Released 2016-04-05

- Add support for loading configuration file in `nameko shell` via `--config` option
- Changed `HttpRequestHandler` to allow override how web exceptions are handled
- Enabled reliable delivery on broadcast events when combined with a custom `broadcast_identity`. Reliable delivery now defaults to enabled for all handler types. It must be explicitly turned off with broadcast mode unless you override the default *broadcast_identity*.
- Update bundled pytest fixtures to use a random vhost in RabbitMQ by default
- Now requires eventlet>=0.16.1 because older versions were removed from PyPI

2.7.26 Version 2.2.0

Released 2015-10-04

- Add support for alternative serializers in AMQP messages
- Add pytest plugin with common fixtures
- Fix examples in documentation and add tests to prevent future breakage
- Fix bug handling non-ascii characters in exception messages
- Various documentation fixes

2.7.27 Version 2.1.2

Released 2015-05-26

- Refactor the standalone queue consumer for more extensibility

2.7.28 Version 2.1.1

Released 2015-05-11

- Nameko shell to use bpython or ipython interpreter if available
- Support for marking entrypoint arguments as sensitive (for later redaction)

2.7.29 Version 2.1.0

Released 2015-04-13

- Changed default AMQP URI so examples work with an unconfigured RabbitMQ.
- Heuristic messages for AMQP connection errors.
- Added six to requirements.
- Minor documentation fixes.

2.7.30 Version 2.0.0

Released 2015-03-31

- python 3 compatibility
- Added HTTP entrypoints and experimental websocket support (contributed by Armin Ronacher)
- Added CLI and console script
- Introduction of nameko "extensions" and nomenclature clarification
- Removal of `DependencyFactory` in favour of prototype pattern
- Complete documentation rewrite
- Spun out `nameko.contrib.sqlalchemy` into nameko-sqlalchemy as a **community extension**.
- Spun out `nameko.legacy` package into nameko-nova-compat
- Rename the standalone rpc proxy to `ServiceRpcProxy` and add a `ClusterRpcProxy`, using a single reply queue for communicating with multiple remote services.
- Make the standalone event dispatcher more shell-friendly, connecting on demand.

2.7.31 Version 1.14.0

Released 2014-12-19

- Remove parallel provider in favour of async RPC
- Update `worker_factory()` to raise if asked to replace a non-existent injection.
- Add various `__repr__` methods for better logging
- Support for timeouts in the (non-legacy) standalone RPC proxy
- Add helper for manipulating an AMQP URI into a dict

2.7.32 Version 1.13.0

Released 2014-12-02

- RPC reply queues now auto-delete.
- Extra protection against badly-behaved dependencies during container kill
- Make legacy `NovaRpcConsumer` more robust against failures in the `NovaRpc` provider.

2.7.33 Version 1.12.0

Released 2014-11-25

- Add ability to make asynchronous rpc calls using the rpc proxy.
- Add a new nameko context key `user_agent` to support including such info in the rpc header.

2.7.34 Version 1.11.5

Released 2014-11-18

- Have the standalone rpc proxy mark its reply queues as auto-delete, to stop them staying around after use.

2.7.35 Version 1.11.4

Released 2014-11-10

- Make `RpcConsumer` more robust against failures in the `Rpc` provider.
- Add a new exception `MalformedRequest` that RPC providers can raise if they detect an invalid message. Raise this exception in the default `Rpc` provider if `args` or `kwargs` keys are missing from the message.
- Fix issues in queue consumer tests against non-localhost brokers.
- Upgrade to eventlet 0.15.2.
- Include pyrabbit in `requirements.txt` (no longer just for tests).
- Catch dying containers in the entrypoint_hook to avoid hangs.
- Add `expected_exceptions` kwarg to the rpc entrypoint to enable different exception handling (in dependencies) for user vs system errors.

2.7.36 Version 1.11.3

Released 2014-10-10

- Add more logging for workers killed by `kill()`.

2.7.37 Version 1.11.2

Released 2014-09-18

- Add a default implementation for `acquire_injection` (returning `None`) for dependency providers that are used for side-effects rather than injecting dependencies.

2.7.38 Version 1.11.1

Released 2014-09-15

- New test helper `entrypoint_waiter()` to wait for entrypoints (e.g. event handlers) to complete.

2.7.39 Version 1.11.0

Released 2014-09-01

- Raise a specific `RpcTimeout` error in the RPC proxy rather than `socket.timeout` to avoid confusing kombu's `Connection.ensure`
- Improve logging helpers
- Use `inspect.getcallargs` instead of shadow lambda for RPC argument checking
- Add default retry policies to all publishers
- Stricter handling of connections between tests
- Workarounds for RabbitMQ bugs described at https://groups.google.com/d/topic/rabbitmq-users/lrl0tYd1L38/discussion

2.7.40 Version 1.10.1

Released 2014-08-27

- Inspect the service class (instead of an instance) in `worker_factory()`. Works better with descriptors.
- Explicitly delete `exc_info` variable when not needed, to help the garbage collector.

2.7.41 Version 1.10.0

Released 2014-08-14

- Entrypoint providers' handle_result is now able to manipulate and modify and return the `(result, exc_info)` tuple. This enables default post-processing (e.g. serialization, translations)
- Added serialization safety to legacy RPC entrypoint.

2.7.42 Version 1.9.1

Released 2014-08-12

- Bugfix to exception handling in `nameko.legacy.dependencies`

2.7.43 Version 1.9.0

Released 2014-07-15

- No longer relying on eventlet for standalone RPC proxy timeouts.
- Introduced RPC entrypoints compatible with the 'legacy' proxy.

2.7.44 Version 1.8.2

Released 2014-07-07

- Documentation generator accepts a function listing event classes and adds to output accordingly.

2.7.45 Version 1.8.1

Released 2014-06-23

- Adding `wait_for_worker_idle` test helper.

2.7.46 Version 1.8.0

Released 2014-06-13

- Now passing `exc_info` tuples instead of bare exceptions to `worker_result` and `handle_result`, to enable exception processing in non-worker greenthreads.

2.7.47 Version 1.7.2

Released 2014-06-10

- `_run_worker()` now calls any `handle_result` method before dependency teardown.
- Serialization errors now generate a specific error message rather than bubbling into the container.
- Minor change to `nameko_doc` output.

2.7.48 Version 1.7.1

Released 2014-05-20

- Added `language`, `auth_token` and `user_id` dependency providers to make context data available to service workers.
- Refactored constants into their own module.
- Minor test changes to enable testing on shared rabbit brokers.

2.7.49 Version 1.7.0

Released 2014-05-07

- `spawn_worker()` now throws `ContainerBeingKilled` if a `kill` is in progress, since some providers may already be dead. Providers should catch this and e.g. requeue rpc messages. There is a race condition between completing the kill sequence and remaining entrypoints firing.

2.7.50 Version 1.6.1

Released 2014-04-03

- Revert changes to legacy exception serialization to maintain backwards compatibility with old clients.
- Add forwards compatibility for future clients that wish to serialize exceptions into more data
- Promote conftest rabbit manipulations to test helpers

2.7.51 Version 1.6.0

Released 2014-03-31

- Rename instance_factory to worker_factory
- Raise `IncorrectSignature` instead of `RemoteError: TypeError` if an RPC method is called with invalid arguments.
- Raise `MethodNotFound` instead of `RemoteError: MethodNotFound` if a non-existent RPC method is called.
- Let log handlers format warning messages so that aggregators group them correctly.
- Expose the entire dependency provider (rather than just the method name) to the worker context.

2.7.52 Version 1.5.0

Released 2014-03-27

- Improvements to `kill()` enabling better tracebacks and cleaner teardown:
  - Using `sys.exc_info` to preserve tracebacks
  - No longer passing exception into `kill()`, removing race conditions.
  - No longer requiring `exc` in `kill()`

2.7.53 Version 1.4.1

Released 2014-03-26

- Adds the `nameko_doc` package, for easing the creation of service-oriented documentation.

2.7.54 Version 1.4.0

Released 2014-03-20

- RPC calls to non-existent services (no queues bound to the RPC exchange with the appropriate routing key) now raise an exception instead of hanging indefinitely. Note that calls to existing but non-running services (where the queue exists but has no consumer) behave as before.

2.7.55 Version 1.3.5

Released 2014-03-05

- Increased test resilience. Force-closing existing connections on rabbit reset

2.7.56 Version 1.3.4

Released 2014-03-05

- Use `MagicMock` for dependency replacement in test utilities
- Use `autospec=True` wherever possible when mocking
- Merge `ServiceContainers` into a single class

2.7.57 Version 1.3.3

Released 2014-02-25

- Bugfixes enabling reconnection to the broker if the connection is temporarily lost.

2.7.58 Version 1.3.2

Released 2014-02-13

- Dropping headers with a `None` value because they can't be serialized by AMQP

2.7.59 Version 1.3.1

Released 2014-01-28

- Add `event_handler_cls` kwarg to the `event_handler` entrypoint, for using a custom subclass of the `EventHandler` provider

2.7.60 Version 1.3.0

Released 2014-01-23

- Standalone RPC proxy interface changed to class with contextmanager interface and manual `start()` and `stop()` methods.

S

W