gemstone Documentation

Release 0.12.0

Vlad Calin

Apr 22, 2017

Contents

The **gemstone** library aims to provide an easy way to develop simple and scalable microservices by using the asynchronous features of Python.

This library offers support for writing a microservice that:

- exposes a public Json RPC 2.0 HTTP API (see The JSON RPC 2.0 specifications)
- can communicate with other microservices through the JSON RPC protocol.
- can communicate with other microservices through events (messages).

This documentation is structured in multiple parts:

- *Topics* - A compilation in-depth explanations on various topics of interest.
- *Modules* - The reference to the classes, functions, constants that can be used.

See also:

- JSON RPC 2.0 specifications: http://www.jsonrpc.org/specification
- Tornado: http://www.tornadoweb.org/en/stable/

CHAPTER 1

Installation

```
pip install gemstone
# or
pip install gemstone[redis] # to use the Redis features
# or
pip install gemstone[rabbitmq] # to use the RabbitMq features
```

CHAPTER 2

First look

In a script `hello_world.py` write the following:

```
import gemstone.core

class HelloWorldService(gemstone.core.MicroService):
    name = "hello_world_service"
    host = "127.0.0.1"
    port = 8000

    @gemstone.core.exposed_method()
    def say_hello(self, name):
        return "hello {}".format(name)

if __name__ == '__main__':
    service = HelloWorldService()
    service.start()
```

We have now a microservice that exposes a public method `say_hello` and returns a `"hello {name}"`.

What we did is the following:

- declared the class of our microservice by inheriting `gemstone.MicroService`
- assigned a name for our service (this is required)
- assigned the `host` and the `port` where the microservice should listen
- exposed a method by using the `gemstone.exposed_method()` decorator.
- after that, when the script is directly executed, we start the service by calling the `gemstone.MicroService.start()` method.

To run it, run script

```
python hello_world.py
```

Now we have the service listening on `http://localhost:8000/api` (the default configuration for the URL endpoint). In order to test it, you have to do a HTTP `POST` request to that URL with the content:

```
curl -i -X POST \
   -H "Content-Type:application/json" \
   -d '{"jsonrpc": "2.0","id": 1,"method": "say_hello","params": {"name": "world"}}' \
 'http://localhost:8000/api'
```

The answer should be

```
{"result": "hello world", "error": null, "jsonrpc": "2.0", "id": 1}
```

Table of contents:

Topics

Various topics of interest

RPC communication via JSON RPC 2.0

Note: Check out the JSONRPC 2.0 protocol specifications .

The implementation

The RPC functionality is provided by the `gemstone.TornadoJsonRpcHandler`. It is important to note that the blocking methods (that are not coroutines) are not executed in the main thread, but in a `concurrent.features.ThreadPoolExecutor`. The methods that are coroutines are executed on the main thread.

In order to create a basic microservice, you have to create a class that inherits the `gemstone.MicroService` class as follows

```
import gemstone

class MyMicroService(gemstone.MicroService):

    name = "hello_world_service"
    ...
```

Check out the `gemstone.MicroService` documentation for the available attributes

Public methods

Any method decorated with :py:func:`gemstone.exposed_method` is a public method that can be accessed by anybody.

Example exposed method:

```
class MyMicroService(gemstone.MicroService):
    # ...

    @gemstone.exposed_method()
```

```
    def say_hello(self, world):
        return "hello {}".format(world)

    # ...
```

By default, a public method is blocking and will be executed in a threaded executor.

You can make an exposed method a coroutine as shown in the next example. From a coroutine you can call other coroutines and can call blocking function by using the provided executor (`gemstone.MicroService.get_executor()`)

```
class MyMicroService(gemstone.MicroService):
    # ...

    @gemstone.exposed_method(is_coroutine=True)
    def say_hello_coroutine(self, world):
        yield self.get_executor.submit(time.sleep, 3)
        return "hello {}".format(world)

    # ...
```

You can expose the public method under another name by giving the desired name as parameter to the `gemstone.exposed_method()` decorator. The new name can even use dots!

```
class MyMicroService(gemstone.MicroService):
    # ...

    @gemstone.exposed_method("myservice.say_hello")
    def ugly_name(self, world):
        return "hello {}".format(world)

    # ...
```

Private methods

TODO

Interacting with the microservice

Interaction with the microservice can be done by using the JSON RPC 2.0 protocol over HTTP.

Interacting with another microservice

Interaction with another microservice can be done by using the `gemstone.RemoteService` class. If at least one service registry was configured, you can use the `gemstone.MicroService.get_service()` method.

Or, as an alternative, if you know the exact network location of the remote service, you can instantiate a `gemstone.RemoteService` yourself

```
remote_service = gemstone.RemoteService("http://10.0.0.1:8000/api")
r = remote_service.call_method("say_hello", ("world",))
print(r.result)
# "hello world"
```

FAQ

TODO

Publisher-subscriber pattern

In order to use the publisher-subscriber paradigm, you need to define at least one event transport

The currently implemented event transports are

- `gemstone.event.transport.RabbitMqEventTransport`
- `gemstone.event.transport.RedisEventTransport`

```
class ExampleService(gemstone.MicroService):
    # ...
    event_transports = [
        gemstone.events.transport.RedisEventTransport("redis://127.0.0.1:6379/0"),
        gemstone.events.transport.RedisEventTransport("redis://redis.example.com:6379/
↪0"),
        # ...
    ]
    # ...
```

After that, for publishing an event, you must call the `gemstone.MicroService.emit_event()` method

```
@gemstone.exposed_method()
def some_method(self):
    self.emit_event("test_event", {"message": "hello there"})
    self.emit_event("method_calls", {"method": "some_method"})
    # ...
```

In order to subscribe to some kind of events, you need to designate a method as the event handler

```
@gemstone.event_handler("test_event")
def my_event_handler(self, event_body):
    self.logger.info("Received event: {}".format(event_body))
```

Note: Event handler methods will be executed on the main thread, so they should not be blocking.

Service discovery

Enabling automatic service discovery

In order to enable automatic service discovery, you need to define at least one discovery strategy.

```
class ExampleService(gemstone.MicroService):
    # ...
    discovery_strategies = [
        gemstone.discovery.RedisDiscoveryStrategy("redis://registry.example.com:6379/0
↪"),
        # ...
    ]
    # ...
```

Using service discovery

You can use the `gemstone.MicroService.get_service()` method to automatically discover other microservice that uses at least one discovery strategy as your service does, and interact with it directly.

Example

```
# ...
remote_service = self.get_service("user_manager_service")
if not remote_service:
    raise RuntimeError("Service could not be located")

res = remote_service.call_method("find_user", {"username": "example"}
# ...
```

The `gemstone.MicroService.get_service()` method returns a `gemstone.RemoteService()` instance. See gemstone_client for more information on this topic.

Configurable features

In the context of this framework, configurables are entities that designate what properties of the microservice can be dynamically set and configurators are strategies that, on service startup, collects the required properties from the environment.

Currently, the available confugurators are:

- `gemstone.config.configurator.CommandLineConfigurator` - collects values from the command line arguments

In order to specify configurables for the microservice, you have to provide set the `gemstone.MicroService.configurables` attribute to a list of `Configurable` objects.

Configurators are specified in the `gemstone.MicroService.configurators` attribute. On service startup, each configurator tries to extract the required values from the environment in the order they are defined.

In order to trigger the configurators, you need to explicitly call the `gemstone.MicroService.configure()` method before calling `gemstone.MicroService.start()`

Defining configurators

Configurators are defined in the `gemstone.MicroService.configurators` class attribute.

```
class ExampleService(gemstone.MicroService):
    # ...
    configurators = [
        gemstone.config.CommandLineConfigurator()
    ]
    # ...
```

Defining configurables

Configurables are defined in the `gemstone.MicroService.configurables` class attribute.

```
class ExampleService(gemstone.MicroService):
    # ...
    configurables = [
```

```
        gemstone.config.Configurable("port", template=lambda x: int(x)),
        gemstone.config.Configurable("discovery_strategies",
                                     template=lambda x: [RedisDiscoveryStrategy(a)␣
↪for a in x.split(",")]),
        gemstone.config.Configurable("host"),
        gemstone.config.Configurable("accessible_at"),
    ]
    # ...
```

In the example above, we defined 4 configurables:

- `gemstone.config.Configurable("port", template=lambda x: int(x))` the `--port` command-line argument will be casted to `int` and assigned to `gemstone.MicroService.port`
- `gemstone.config.Configurable("host")` the `--host` command-line argument assigned to `gemstone.MicroService.host`

Modules

The gemstone.core module

The gemstone.core.MicroService class

Can be called

Can be overridden

The gemstone.core.Container class

Decorators

The gemstone.client module

The gemstone.client.RemoteService class

class gemstone.client.**RemoteService**(*service_endpoint*, ***, *authentication_method=None*)

call_method(*method_name_or_object*, *params=None*)
Calls the `method_name` method from the given service and returns a `gemstone.client.structs.Result` instance.

Parameters
- **method_name_or_object** – The name of te called method or a `MethodCall` instance
- **params** – A list of dict representing the parameters for the request

Returns a `gemstone.client.structs.Result` instance.

call_method_async(*method_name_or_object*, *params=None*)
Calls the `method_name` method from the given service asynchronously and returns a `gemstone.client.structs.AsyncMethodCall` instance.

Parameters
- **method_name_or_object** – The name of te called method or a `MethodCall` instance
- **params** – A list of dict representing the parameters for the request

Returns a `gemstone.client.structs.AsyncMethodCall` instance.

handle_single_request(*request_object*)
Handles a single request object and returns the raw response

Parameters **request_object** –

notify(*method_name_or_object*, *params=None*)
Sends a notification to the service by calling the `method_name` method with the `params` parameters. Does not wait for a response, even if the response triggers an error.

Parameters
- **method_name_or_object** – the name of the method to be called or a `Notification` instance
- **params** – a list of dict representing the parameters for the call

Returns None

Various structures

class gemstone.client.**AsyncMethodCall**(*req_obj*, *async_resp_object*)

result(*wait=False*)
Gets the result of the method call. If the call was successful, return the result, otherwise, reraise the exception.

Parameters **wait** – Block until the result is available, or just get the result.

Raises RuntimeError when called and the result is not yet available.

class gemstone.client.**MethodCall**(*method_name*, *params=None*, *id=None*)

class gemstone.client.**Notification**(*method_name*, *params=None*, *id=None*)

class gemstone.client.**Result**(*result*, *error*, *id*, *method_call*)

class gemstone.client.**BatchResult**(**responses*)

The gemstone.config module

Configurables

class gemstone.config.**Configurable**(*name*, ***, *template=None*)
Defines a configurable value for the application.

Example (You should not use configurables in this way unless you are writing a custom `Configurator`)

```
c = Configurable("test", template=lambda x: x * 2)
c.set_value("10")
c.get_final_value()  # "10" * 2 -> 1010

c2 = Configurable("list_of_ints", template=lambda x: [int(y) for y in x.
↪split(",")])
c.set_value("1,2,3,4,5")
c.get_final_value()  # [1,2,3,4,5]
```

Parameters

- **name** – The name of the configurable parameter
- **template** – A callable template to apply over the extracted value

Configurators

class gemstone.config.**BaseConfigurator**

Base class for defining configurators. A configurator is a class that, starting from a set of name-configurable pairs, depending on the configurables' options and the environment, builds a configuration for the application.

get(*name*)

Gets the extracted value for the specified name, if available. If no value could be loaded for the specified name, `None` must be returned.

get_configurable_by_name(*name*)

Returns the registered configurable with the specified name or `None` if no such configurator exists.

load()

Loads the configuration for the application

register_configurable(*configurable*)

Registers a configurable instance with this configurator

Parameters **configurable** – a *Configurable* instance

class gemstone.config.**CommandLineConfigurator**

Configurator that collects values from command line arguments. For each registered configurable, will attempt to get from command line the value designated by the argument `--name` where `name` is the name of the configurable.

Example

For the configurables

•Configurable("a")

•Configurable("b")

•Configurable("c")

the following command line interface will be exposed

```
usage: service.py [-h] [--a A] [--b B] [--c C]

optional arguments:
```

```
  -h, --help  show this help message and exit
  --a A
  --b B
  --c C
```

The `service.py` can be called like this

```
python service.py --a=1 --b=2 --c=3
```

The gemstone.event module

Event transports

class `gemstone.event.`**`BaseEventTransport`**

Base class for defining event transports.

The basic workflow would be the following:

- the handlers are registered with the *`BaseEventTransport.register_event_handler()`* method
- the *`BaseEventTransport.start_accepting_events()`* is invoked
- for each incoming event, call *`BaseEventTransport.on_event_received()`* whose responsibility is to invoke the proper handler function (recommended to use the `run_on_main_thread` method)

`emit_event`(*event_name*, *event_body*)

Emits an event of type `event_name` with the `event_body` content using the current event transport.

Parameters
- **`event_name`** –
- **`event_body`** –

Returns

`on_event_received`(*event_name*, *event_body*)

Handles generic event. This function should treat every event that is received with the designated handler function.

Parameters
- **`event_name`** – the name of the event to be handled
- **`event_body`** – the body of the event to be handled

Returns

`register_event_handler`(*handler_func*, *handled_event_name*)

Registers a function to handle all events of type `handled_event_name`

Parameters
- **`handler_func`** – the handler function
- **`handled_event_name`** – the handled event type

`run_on_main_thread`(*func*, *args=None*, *kwargs=None*)

Runs the `func` callable on the main thread, by using the provided microservice instance's IOLoop.

Parameters
- **`func`** – callable to run on the main thread
- **`args`** – tuple or list with the positional arguments.
- **`kwargs`** – dict with the keyword arguments.

Returns

set_microservice(*microservice*)
Used by the microservice instance to send reference to itself. Do not override this.

start_accepting_events()
Starts accepting and handling events.

class gemstone.event.**RedisEventTransport**(*redis_url*)
Event transport that uses a Redis server as message transport by using the PUBSUB mechanism.

Parameters **redis_url** – A string that specifies the network location of the redis server: - `redis://[:password@]hostaddr:port/dbnumber` (plaintext) - `rediss://[:password@]hostaddr:port/dbnumber` (over TLS) - `unix://[:password@]/path/to/socket?db=dbnumber` (Unix socket)

class gemstone.event.**RabbitMqEventTransport**(*host='127.0.0.1'*, *port=5672*, *username=''*, *password=''*, ***connection_options*)
Event transport via RabbitMQ server.

Parameters

- **host** – ipv4 or hostname
- **port** – the port where the server listens
- **username** – username used for authentication
- **password** – password used for authentication
- **connection_options** – extra arguments that will be used in `pika.BlockingConnection` initialization.

The gemstone.discovery module

Discovery strategies

Caches

The gemstone.plugins module

The gemstone.plugins.BasePlugin class

Exceptions

The gemstone.util module

gemstone.util.**as_completed**(**async_result_wrappers*)
Yields results as they become available from asynchronous method calls.

Example usage

```
async_calls = [service.call_method_async("do_stuff", (x,)) for x in range(25)]

for async_call in gemstone.as_completed(*async_calls):
    print("just finished with result ", async_call.result())
```

Parameters **async_result_wrappers** – gemstone.client.structs.AsyncMethodCall instances.

Returns a generator that yields items as soon they results become available.

New in version 0.5.0.

gemstone.util.**dynamic_load**(*module_or_member*)
Dynamically loads a class or member of a class.

If `module_or_member` is something like `"a.b.c"`, will perform `from a.b import c`.

If `module_or_member` is something like `"a"` will perform `import a`

Parameters **module_or_member** – the name of a module or member of a module to import.

Returns the returned entity, be it a module or member of a module.

gemstone.util.**first_completed**(**async_result_wrappers*)
Just like *as_completed()*, but returns only the first item and discards the rest.

Parameters **async_result_wrappers** –

Returns

New in version 0.5.0.

Changes

0.12.0 (22.04.2017)

- restructured modules
- bug fixes
- improved documentation
- improved tests

0.11.0 (08.04.2017)

- added `Container.get_io_loop` method
- added `Container.get_executor` method
- added `RedisEventTransport`

- `emit_event` now emits just events. Removed the `broadcast` parameter. A task handling functionality will be added in a further version
- improved docs (still a work in progress)
- added some more tests (still a work in progress)

0.10.1 (27.03.2017)

- removed some forgotten debug messages

0.10.0 (23.03.2017)

- added `broadcast` parameter to `MicroService.emit_event`
- added the `broadcast` parameter to `BaseEventTransport.emit_event`
- added the `broadcast` parameter to `RabbitMqEventTransport.emit_event`
- improved tests and documentation
- removed `mappings` and `type` parameters from `Configurable`
- added `gemstone.Module` for better modularization of the microservice
- added `gemstone.MicroService.authenticate_request` method for a more flexible authentication mechanism
- deprecated `gemstone.MicroService.api_token_is_valid` method

0.9.0 (06.03.2017

- **added the `gemstone.exposed_method` decorator for general usage that allows**
  - to customize the name of the method
  - to specify if the method is a coroutine
  - to specify that the method requires a handler reference
  - to specify that the method is public or private
- **deprecated**
  - `gemstone.public_method` decorator
  - `gemstone.private_api_method` decorator
  - `gemstone.async_method` decorator
  - `gemstone.requires_handler_reference` decorator
- removed `gemstone.MicroService.get_cli` method in favor of the `CommandLineConfigurator`
- improved documentation a little bit

0.8.0 (05.03.2017)

- added the `gemstone.requires_handler_reference` decorator to enable the methods to get a reference to the Tornado request handler when called.
- added the `gemstone.async_method` decorator to make a method a coroutine and be able to execute things asynchronously on the main thread. For example, a method decorated with `async_method` will be able to `yield self._executor.submit(make_some_network_call)` without blocking the main thread.
- **added two new examples:**
  - `example_coroutine_method` - shows a basic usage if the `async_method` decorator
  - `example_handler_ref` - shows a basic usage if the `requires_handler_reference` decorator

0.7.0 (27.02.2017)

- added `gemstone.GemstoneCustomHandler` class
- modified the way one can add custom Tornado handler to the microservice. Now these handlers must inherit `gemstone.GemstoneCustomHandler`
- restructured docs, now it is based more on docstrings
- improved tests and code quality

0.6.0 (14.02.2017)

- **added configurable framework:**
  - `gemstone.config.configurable.Configurable` class
  - `gemstone.config.configurator.*` classes
  - `gemstone.MicroService.configurables` and `gemstone.MicroService.configurators` attributes
  - switched testing to pytest
  - improved documentation (restructured and minor additions). Still a work in progress

0.5.0 (09.02.2017)

- **added support for publisher-subscriber communication method:**
  - base class for event transports: `gemstone.event.transport.BaseEventTransport`
  - first concrete implementation: `gemstone.event.transport.RabbitMqEventTransport`
  - `gemstone.MicroService.emit_event` for publishing an event
  - `gemstone.event_handler` decorator for designating event handlers
- restructured documentation (added tutorial, examples and howto sections).
- added asynchronous method calls in `gemstone.RemoteService`.
- added `gemstone.as_completed`, `gemstone.first_completed`, `gemstone.make_callbacks` utility functions for dealing with asynchronous method calls.

0.4.0 (25.01.2017)

- modified `accessible_at` attribute of the `gemstone.MicroService` class
- added the `endpoint` attribute to the `gemstone.MicroService` class
- improved how the microservice communicates with the service registry

0.3.1 (25.01.2017)

- fixed event loop freezing on Windows
- fixed a case when a `TypeError` was silenced when handling the bad parameters error in JSON RPC 2.0 handler (#21)
- major refactoring (handling of JSON RPC objects as Python objects instead of dicts and lists) to improve readability and maintainability
- improved documentation

0.3.0 (23.01.2017)

- added validation strategies (method for extraction of api token from the request)
- base subclass for implementing validation strategies
- built in validation strategies: `HeaderValidationStrategy`, `BasicCookieStrategy`
- improved documentation

0.2.0 (17.01.2017)

- added `gemstone.RemoteService.get_service_by_name` method
- added `call` command to cli
- added `call_raw` command to cli
- improved documentation a little

0.1.3 (16.01.2017)

- fixed manifest to include required missing files

0.1.2 (16.01.2017)

- added py36 to travis-ci
- refactored setup.py and reworked description files and documentation for better rendering

0.1.1 (13.01.2017)

- changed the name of the library from `pymicroservice` to `gemstone`
- added the `gemstone.MicroService.accessible_at` attribute

0.1.0 (09.01.2017)

- added the `pymicroservice.PyMicroService.get_cli` method
- improved documentation a little bit

0.0.4

- fixed bug when sending a notification that would result in an error was causing the microservice to respond abnormally (see #10)
- fixed a bug that was causing the service to never respond with the invalid parameters status when calling a method with invalid parameters

0.0.3

- added `pymicroservice.RemoteService` class
- added the `pymicroservice.PyMicroService.get_service(name)`
- improved documentation

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index