gemstone Documentation

Release 0.10.1

Vlad Calin

Mar 27, 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 protect API access based on API token identification.
- 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:

- *Overview* - General information to get you started.
- *Topics* - A compilation in-depth explanations on various topics of interest.
- *Reference* - 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

Hello world

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

```
import gemstone

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

    @gemstone.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 address 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:

Overview

Overview

Motivation

In the past years, the microservice-based architecture became very popular in the computing field. Although this architecture grew more and more popular, there are a few tools that can help an individual to build such systems. The current alternatives are using nameko or by building a web application that acts like a microservice. I started developing this framework in order to provide a tool for creating and managing such systems with ease, and that are capable of being specialized in a certain role, be it entity management, data storage or just computing.

Few words ahead

This library uses the asynchronous features of the Tornado web framework for creating a JSON RPC endpoint through which one can call exposed methods. The method calls are treated asynchronously. If you have no knowledge about asynchronous programming in Python, I suggest to read a few words from the Tornado documentation .

Although it is not required for you to know about all that coroutines and event loop theory, it sure helps to understand what happens *under the hood*.

Features

The main features of this framework are:

- microservices that communicate over JSON RPC 2.0 protocol.
- possibility to extend with custom functionality (via Tornado request handlers)
- automatic service discovery
- dynamic configuration (no need to modify the code to change the running parameters)
- support for the publisher-subscriber communication pattern

Installation

The library can be installed via pip

```
pip install gemstone
```

or from source files

```
git clone https://github.com/vladcalin/gemstone.git
cd gemstone
python setup.py install
```

Creating a microservice

Basic example

In order to create a simple microservice, you have to subclass the *gemstone.MicroService* base class:

```
class HelloWorldService(MicroService):
    name = "hello.world.service"
    host = "127.0.0.1"
    port = 5000

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

    @private_api_method
    def say_private_hello(self, name):
        return "this is secret: hello {}".format(name)

    def api_token_is_valid(self, api_token):
        return api_token == "hello_world_token"


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

After you created your service, run the script that contains it and enjoy.

Exposing public methods

Public methods can be exposed by decorating them with the *gemstone.public_method()* decorator

```
class MyMicroService(MicroService):

    # stuff

    @public_method
    def exposed_public_method(self):
        return "it works!"

    # more stuff
```

Exposing private methods

In order to expose private methods, we have to decorate them with the *gemstone.private_api_method()*. These methods can be accessed only by providing a valid Api Token with the request. In addition, we must override the `gemstone.MicroService.api_token_is_valid()` method to implement the token validation logic

```
class MyMicroService(MicroService):

    # stuff

    @private_api_method
    def exposed_private_method(self):
        return "it works!"

    def api_token_is_valid(self, api_token):
        return api_token == "correct_token"

    # more stuff
```

Customize the microservice

We can define various specifications for our microservice. The following class attributes can be overridden to customize the behavior of our microservice.

Required attributes

- *gemstone.MicroService.name* is required and defines the name of the microservice. **MUST** be defined by the concrete implementation, otherwise an error will be thrown at startup

Specifying different host, port and location

- *gemstone.MicroService.host* - specifies the address to bind to (hostname or IP address). Defaults to `127.0.0.1`.
- *gemstone.MicroService.port* - an `int` that specifies the port to bind to. Defaults to `8000`
- *gemstone.MicroService.endpoint* - a string representing the url where the service api will be accessible. Defaults to `"/api"`, so by default, the service will be accessible at `http://{host}:{port}/api`.
- *gemstone.MicroService.accessible_at* - a string representing a http(s) address specifying a custom location where the service can be found. If at least one service registry is configured, the service will send this value to it so that other services can access at the specified location.

  Example: `"http://2a330155abfc.myservice.com/workers/api"`

  For example, it is useful when the service runs behind a load balancer and the *gemstone.MicroService.accessible_at* attribute will point to the address of the load balancer, so that when another service queries the registry for this service, it will access the load balancer instead.

Event dispatching

- *gemstone.MicroService.event_transports* - a list of `gemstone.event.transport.BaseEventTransport`. See *Event transports* for available implementations and *Publisher-subscriber pat-*

tern for usage.

Other options

- `gemstone.MicroService.validation_strategies` - a list of validation strategy instances that will be used to extract the api token that will be forwarded to the `MicroService.api_token_is_valid` method. Defaults to `[HeaderValidationStrategy(header="X-Api-Token", template=None)]`

  See *Token validation strategies* for more details, available options and how to implement custom validation strategies

  If multiple strategies are specified, they will be run in the order they are defined until the first one extracts a value which is not `None`.

  In order to interact with a service that uses a validation strategy, we have to specify the proper arguments in the *gemstone.RemoteService* constructor (See the class definition for more info on this).

  New in version 0.3.0.

- *gemstone.MicroService.max_parallel_blocking_tasks* - the number of threads that will handle blocking actions (function calls). Defaults to `os.cpu_count()`.

Adding web application functionality

There might be situations when we want to extend the functionality of the microservice so that it will display some stats on some pages (or other scenarios). This library provides a way to quickly add behaviour that is not API-related.

- *gemstone.MicroService.static_dirs* - a list of `(str, str)` tuples that represent the URL to which the static directory will be mapped, and the path of the directory that contain the static files. For example, if the directory `/home/user/www/static` contains the file `index.html`, and we specify the static dir attribute with the value `[("/static", "/home/user/www/static")]`, the service will serve `index.html` at the URL `/static/index.html`.
- *gemstone.MicroService.extra_handlers* - a list of tuples of URLs and Tornado request handlers to be included in the service.

  Note: Make sure that no other handle overwrites the endpoint of the service.

- *gemstone.MicroService.template_dir* - a directory where templates will be searched in, when, in a custom handler we render a template via `tornado.web.RequestHandler.render()`.

Periodic tasks

- *gemstone.MicroService.periodic_tasks* - a list of function - interval (in seconds) mappings that schedules the given function to be executed every given seconds

```
def periodic_func():
    print("hello there")

class MyService(MicroService):

    # stuff
```

```
    periodic_tasks = [(periodic_func, 1)]

    # stuff
```

In te above example, the `periodic_func` will be executed every second.

Note: There might be a little delay in the execution of the function, depending on the main event loop availability. See the Tornado documentation on PeriodicCallback for more details.

Note: If you want to pass parameters to a function, you can use the `functools.partial()` to specify the parameters for the function to be called with.

Using a service registry

A service registry is a remote service that keeps mappings of service names and network locations, so that each microservice will be able to locate another one dynamically. A service can be a service registry if it exposes via JSON RPC a `ping(name, url)` method and a `locate_service(name)` method.

- *`gemstone.MicroService.service_registry_urls`* - a list of URLS where a service registry is located and accessible via JSON RPC.

```
service_registry_urls = ["http://registry.domain.com:8000/api", "http://registry.
↪domain2.com"]
```

On service startup, a ping will be sent to the registry, and after that, a ping will be sent periodically.

- *`gemstone.MicroService.service_registry_ping_interval`* - the interval (in seconds) when the service will ping the registry. Defaults to 30 seconds.

```
service_registry_ping_interval = 120  # ping every two minutes
```

Generating a command-line interface

See `gemstone.MicroService.get_cli()` for more details.

Interacting with services

There are a few methods to communicate with microservices. This framework, being written around the JSON RPC protocol, allows microservices to be easily integrated with each other.

Through raw HTTP requests

First method to interact with a service is through raw HTTP requests. All you have to do is making a `POST` request to `http://service_ip:service_port/api` with:

- **the headers**
  - `Content-Type: application/json`
- the content

```
{
  "jsonrpc": "2.0"
  "method": "the_name_of_the_method",
  "params": {
      "param_name": "value",
      "param_name_2": "value2"
  },
  "id": 1,
}
```

or

```
{
  "jsonrpc": "2.0"
  "method": "the_name_of_the_method",
  "params": ["value1", "value2"],
  "id": 1,
}
```

If you want to send a notification (you don't care about the answer, don't include the `"id"` field in the request.

Note: See the JSON RPC 2.0 specifications for more details.

Through the `gemstone.RemoteService` class

This library offers the *gemstone.RemoteService* class to interact with other services programmatically.

Example

```
client = RemoteService("http://127.0.0.1:5000/api")

print(client.name)  # "hello.world.service"
print(client.methods.say_hello("world"))  # "hello world"
print(client.notifications.say_hello("world"))  # None -> we sent a notification,␣
↪therefore discarding the result
```

In addition, this class provides a method to asynchronously call methods by passing an extra keyword argument `__async` as shown in the following example

```
async_response = client.methods.say_hello("world", __async=True)

print(async_response)
# <AsyncMethodCall ...>

async_response.wait()
print(async_response.finished())
# True
print(async_response.result())
# "hello world"
print(async_response.error())
# None
```

See also:

- *gemstone.client.remote_service.AsyncMethodCall*,

- *gemstone.as_completed()*,
- *gemstone.first_completed()*
- *gemstone.make_callbacks()*

Using a service registry

We can configure a microservice to use a service registry. A service registry is a service that help services identify other services without needing to know their exact location (services are identified by name).

A service registry can be a client that exposes via JSON RPC 2.0 the methods: `ping(name, host, port)` and `locate_service(name)`.

In order for a service to make use of a service registry, we must override the *gemstone.MicroService.service_registry_urls* class attribute.

When we do that, a periodic task will spawn when the service starts that calls the `ping` method of the remote service, every *gemstone.MicroService.service_registry_ping_interval* seconds.

Note: A service can use multiple service registries. When multiple service registries are used, the service will send `ping` requests to all of them with the specified delay between them.

Example:

```
class ExampleService(MicroService):
    name = "example.1"

    # stuff

    service_registries_urls = ["http://reg.hostname:5000/api", "http://reg.
↪hostname2:8000/api"]

    # more stuff

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

    # even more stuff
```

When at least one service registry is used, we can use the *gemstone.MicroService.get_service()* method to identify a service by name (or glob pattern). For example, if we call the method with the `"myservice.workers.*"` pattern, it will match `"myservice.workers.01"`, `"myservice.workers.02"` and `"myservice.workers.03"`.

Via the gemstone executable

We can interact with the `gemstone` executable using the `call` command:

```
Usage: gemstone call [OPTIONS] NAME METHOD [PARAMS]...

Options:
  --registry TEXT  The service registry URL used for queries
  --help           Show this message and exit.
```

The `registry` option specifies the URL where a service registry is accessible. For example: `"http://192.168.0.1:8000/api"`.

- `NAME` - a glob pattern for the service you want to interact. Keep in mind that in the glob syntax, `*` matches a sequence of characters while `?` matches a single character.
- `METHOD` - the name of the method to call
- `PARAMS` - parameters for the call in the format `name=value`. Current implementation supports only simple string values. In other words you can only send values in the format `key=some_value` that will be translated to `func(key="some_value" ...)`. You can specify multiple parameters

Example:

```
gemstone call --registry=http://localhost:8000/api servicename say_hello name=world
# calls servicename.say_hello with the parameter name="world"
```

But if we want to interact with a service without having a service registry, we can use the `call_raw` command

```
Usage: gemstone call_raw [OPTIONS] URL METHOD [PARAMS]...

Options:
  --help  Show this message and exit.
```

- `URL` - a valid http(s) url where the service is located.
- `METHOD` - the name of the method to be called
- `PARAMS` - same as above

Example:

```
gemstone.exe call_raw http://service.local/api get_service_specs
[!] Service identification: 0.12918 seconds
[!] Method call: 0.01701 seconds
[!] Result:

{'host': '0.0.0.0',
 'max_parallel_blocking_tasks': 4,
 ...
```

Examples

In the `examples` directory you can find some examples of microservices

1. `example_client` - an example usage of the *`gemstone.RemoteService`* class for communication with microservices.

   There you will find two files: `service.py` and `client.py`

   In `service.py` you have a basic microservice that exposes two methods: `say_hello(name)` and `slow_method(seconds)`. You can start it with the command

   ```
   python service.py
   ```

   In `client.py` you can find some basic interaction with the service started above.

2. `example_events` - an example for the publisher-subscribe pattern in the microservice communication. There are two files: `service.py` and `service2.py`. You can start them with the commands

> **Warning:** You are going to need a RabbitMQ server running somewhere because the example uses it as message exchange transport

```
python service.py
python service2.py
```

Note: Those two commands must be executed in separate terminals/cmds because they are blocking.

What happens here is:

- the `service.py` subscribes to `"said_hello"` events.
- the `service2.py` exposes a public method `say_hello(name)`. When called, emits an `"said_hello"` event and then processes the request.

After that, you can send a JSONRPC request to `http://127.0.0.1:8000/api` with the body

```
{
   "jsonrpc": "2.0",
   "method": "say_hello",
   "params": {"name": "world"},
   "id": 1
}
```

and watch what happens.

Using a template for writing a microservice

There is the gemstone-template cookiecutter template for easily setting up a microservice. Check out its readme for more info.

Quick usage

```
pip install cookiecutter gemstone
git clone https://github.com/vladcalin/gemstone-template.git
cookiecutter ./gemstone-template

# answer the questions
# Name: myservice
# Author: Me
# Version: 1.0
# Short description: None

# Now we have the myservice directory with the new service

pip install myservice
myservice start --host=0.0.0.0 --port=8000

# now our first service is up and running

# the service logic is in myservice/service.py
# if you want to create extra handlers (for a web interface for example)
#   add them to myservice/handlers
```

```
# static files are in myservice/html/static
# templates are in myservice/html/templates
```

Enjoy!

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 methods are not executed in the main thread, but in a `concurrent.features.ThreadPoolExecutor`.

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 or *Creating a microservice* for the available attributes

Public methods

TODO

Private methods

TODO

Interacting with the microservice

TODO

Interacting with another microservice

TODO

FAQ

TODO

Publisher-subscriber pattern

TODO

Service discovery

TODO

Configurable features

TODO

Reference

Reference

The `gemstone` module (main classes)

Core classes

class `gemstone.`**`MicroService`**(*io_loop=None*)
The base class for implementing microservices.

Parameters **`io_loop`** – A `tornado.ioloop.IOLoop` instance - can be used to share the same io loop between multiple microservices running from the same process.

Attributes

You can (and should) define various class attributes in order to provide the desired functionality for your microservice. These attributes can be configured at runtime by using the configurable sub-framework (read more at *Configurable features*)

Identification

`MicroService.`**`name`** **= None**
The name of the service. Is required.

`MicroService.`**`host`** **= '127.0.0.1'**
The host where the service will listen

`MicroService.`**`port`** **= 8000**
The port where the service will bind

`MicroService.`**`endpoint`** **= '/api'**
The path in the URL where the microservice JSON RPC endpoint will be accessible.

MicroService.**accessible_at** = None
The url where the service can be accessed by other microservices. Useful when using a service registry.

Access validation

See also:

Private methods

Event dispatching

MicroService.**event_transports** = []
A list of Event transports that will enable the Event dispatching feature.

See also:

- *Publisher-subscriber pattern*
- *Event transports*

Dynamic configuration

MicroService.**skip_configuration** = False
Flag that if set to True, will disable the configurable sub-framework.

MicroService.**configurables** = [<Configurable name=port>, <Configurable name=host>, <Configurable name=ac
A list of configurable objects that allows the service's running parameters to be changed dynamically without changing its code.

MicroService.**configurators** = [<CommandLineConfigurator>]
A list of configurator objects that will extract in order values for the defined configurators

See also:

Configurables and configurators

Web application functionality

MicroService.**extra_handlers** = []
A list of extra Tornado handlers that will be included in the created Tornado application.

MicroService.**template_dir** = '.'
Template directory used by the created Tornado Application. Useful when you plan to add web application functionality to the microservice.

MicroService.**static_dirs** = []
A list of directories where the static files will looked for.

Periodic tasks

MicroService.**periodic_tasks** = []
A list of (callable, time_in_seconds) that will enable periodic task execution.

Service auto-discovery

`MicroService.`**`service_registry_urls`** = []
A list of service registry complete URL which will enable service auto-discovery.

`MicroService.`**`service_registry_ping_interval`** = 30
Interval (in seconds) when the microservice will ping all the service registries.

Misc

`MicroService.`**`max_parallel_blocking_tasks`** = 4
How many methods can be executed in parallel at the same time. Note that every blocking method is executed in a `concurrent.features.ThreadPoolExecutor`

Methods

Can be overridden

`MicroService.`**`on_service_start`**`()`
Override this method to do a set of actions when the service starts

Returns `None`

`MicroService.`**`get_logger`**`()`
Override this method to designate the logger for the application

Returns a `logging.Logger` instance

Can be called

`MicroService.`**`get_service`**`(`*name*`)`
Locates a remote service by name. The name can be a glob-like pattern (`"project.worker.*"`). If multiple services match the given name, a random instance will be chosen. There might be multiple services that match a given name if there are multiple services with the same name running, or when the pattern matches multiple different services.

Todo

Make this use self.io_loop to resolve the request. The current implementation is blocking and slow

---

Parameters **`name`** – a pattern for the searched service.

Returns a *`gemstone.RemoteService`* instance

Raises

- **`ValueError`** – when the service can not be located
- **`ServiceConfigurationError`** – when there is no configured discovery strategy

`MicroService.`**`start_thread`**`(`*target*, *args*, *kwargs*`)`
Shortcut method for starting a thread.

Parameters

- **target** – The function to be executed.
- **args** – A tuple or list representing the positional arguments for the thread.
- **kwargs** – A dictionary representing the keyword arguments.

New in version 0.5.0.

`MicroService.`**`emit_event`**(*event_name*, *event_body*, *, *broadcast=True*)
Publishes an event of type `event_name` to all subscribers, having the body `event_body`. The event is pushed through all available event transports.

The event body must be a Python object that can be represented as a JSON.

Parameters

- **event_name** – a `str` representing the event type
- **event_body** – a Python object that can be represented as JSON.
- **broadcast** – flag that specifies if the event should be received by all subscribers or only by one

New in version 0.5.0.

Changed in version 0.10.0: Added parameter broadcast

`MicroService.`**`get_current_configuration`**()

`MicroService.`**`make_tornado_app`**()
Creates a :py:class‘tornado.web.Application‘ instance that respect the JSON RPC 2.0 specs and exposes the designated methods. Can be used in tests to obtain the Tornado application.

Returns a `tornado.web.Application` instance

`MicroService.`**`start`**()
The main method that starts the service. This is blocking.

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

Decorators

`gemstone.`**`exposed_method`**(*name=None*, *private=False*, *is_coroutine=True*, *requires_handler_reference=False*, ***kwargs*)
Marks a method as exposed via JSON RPC.

Parameters

- **name** – the name of the exposed method. Must contains only letters, digits, dots and underscores. If not present or is set explicitly to `None`, this parameter will default to the name of the exposed method. If two methods with the same name are exposed, a `ValueError` is raised.
- **public** – Flag that specifies if the exposed method is public (can be accessed without token)
- **private** – Flag that specifies if the exposed method is private.
- **is_coroutine** – Flag that specifies if the method is a Tornado coroutine. If True, it will be wrapped with the `tornado.gen.coroutine()` decorator.
- **kwargs** – Not used.

New in version 0.9.0.

gemstone.**event_handler**(*event_name*)
Decorator for designating a handler for an event type. `event_name` must be a string representing the name of the event type.

The decorated function must accept a parameter: the body of the received event, which will be a Python object that can be encoded as a JSON (dict, list, str, int, bool, float or None)

Parameters **event_name** –

Returns

gemstone.**public_method**(*func*)
Decorates a method to be exposed from a `gemstone.PyMicroService` concrete implementation. The exposed method will be public.

Deprecated since version 0.9.0: Use *exposed_method()* instead.

gemstone.**private_api_method**(*func*)
Decorates a method to be exposed (privately) from a `gemstone.PyMicroService` concrete implementation. The exposed method will be private.

Deprecated since version 0.9.0: Use *exposed_method()* instead.

gemstone.**requires_handler_reference**(*func*)
Marks a method tha requires access to the *gemstone.TornadoJsonRpcHandler* instance when calling the request. If a method is decorated with this, when it is called it will receive a `handler` argument as the first argument.

Useful when you need to do specific operations such as setting a cookie, setting a secure cookie, get the `current_user` of the request, etc.

Deprecated since version 0.9.0: Use *exposed_method()* instead.

Request handlers

class gemstone.**GemstoneCustomHandler**(**args*, ***kwargs*)
Base class for custom Tornado handlers that can be added to the microservice.

Offers a reference to the microservice through the `self.microservice` attribute.

class gemstone.**TornadoJsonRpcHandler**(**args*, ***kwargs*)

call_method(*method*)
Calls a blocking method in an executor, in order to preserve the non-blocking behaviour

If `method` is a coroutine, yields from it and returns, no need to execute in in an executor.

Parameters **method** – The method or coroutine to be called (with no arguments).

Returns the result of the method call

handle_single_request(*request_object*)
Handles a single request object and returns the correct result as follows:

- A valid response object if it is a regular request (with ID)
- `None` if it was a notification (if None is returned, a response object with "received" body was already sent to the client.

Parameters **request_object** – A `gemstone.core.structs.JsonRpcRequest` object representing a Request object

Returns A `gemstone.core.structs.JsonRpcResponse` object representing a Response object or None if no response is expected (it was a notification)

prepare_method_call(*method*, *args*)
Wraps a method so that method() will call `method(*args)` or `method(**args)`, depending of args type

Parameters

- **method** – a callable object (method)
- **args** – dict or list with the parameters for the function

Returns a 'patched' callable

write_single_response(*response_obj*)
Writes a json rpc response `{"result": result, "error": error, "id": id}`. If the `id` is `None`, the response will not contain an `id` field. The response is sent to the client as an `application/json` response. Only one call per response is allowed

Parameters **response_obj** – A Json rpc response object

Returns

Token validation strategies

Configurables and configurators

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
- `gemstone.config.configurator.JsonFileConfigurator` - collects values from a JSON file

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.

Configurable

class `gemstone.config.configurable.`**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()  # int("10") * 2 -> 20

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.configurator.**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.

load()

Loads the configuration for the application

class gemstone.config.configurator.**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

•Configurator("a")

•Configurator("b", type=int)

•Configurator("c", type=bool)

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=true
```

Event transports

Utility classes and functions

class gemstone.client.remote_service.**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.

gemstone.**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.**first_completed**(**async_result_wrappers*)

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

Parameters **async_result_wrappers** –

Returns

New in version 0.5.0.

gemstone.**make_callbacks**(*async_result_wrappers*, *on_result*, *on_error*, *run_in_threads=False*)

Monitors the *gemstone.client.remote_service.AsyncMethodCall* instances from *async_result_wrappers* and apply callbacks depending on their outcome.

Parameters

- **async_result_wrappers** – An iterable of *gemstone.client.remote_service.AsyncMethodCall*
- **on_result** – a callable that takes a single positional argument (the result)
- **on_error** – a callabke that takes a single positional argument (the error)
- **run_in_threads** – flag tha specifies if the callbacks should be called in the current thread or in background threads

New in version 0.5.0.

Changes

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

Todo

Make this use self.io_loop to resolve the request. The current implementation is blocking and slow

---

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/gemstone/envs/stable/lib/python3.5/site-packages/gemstone-0.10.1-py3.5.egg/gemstone/core/microservice.py:docstring of gemstone.MicroService.get_service, line 7.)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

S

T

W