ASAB Documentation

Ales Teska

Sep 28, 2018

Content

Asynchronous Server App Boilerplate (or ASAB for short) minimizes the amount of code that one needs to write when building a server application in Python 3.5+. ASAB can also be seen as the extension to *asyncio* that provides a (more or less) complete application framework.

ASAB is developed on GitHub. Contributions are welcome!

CHAPTER 1

ASAB is designed to be simple

```
import asab

class MyApplication(asab.Application):
    async def main(self):
        print("Hello world!")
        self.stop()

if __name__ == "__main__":
    app = MyApplication()
    app.run()
```

1.1 Getting started

Make sure you have both pip and at least version 3.5 of Python before starting. ASAB uses the new `async`/`await` syntax, so earlier versions of python won't work.

1. Install ASAB: `python3 -m pip install asab`
2. Create a file called `main.py` with the following code:

```
#!/usr/bin/env python3
import asab

class MyApplication(asab.Application):
    async def main(self):
        print("Hello world")
        self.stop()

if __name__ == '__main__':
    app = MyApplication()
    app.run()
```

3. Run the server: `python3 main.py`

You now have a working ASAB application server, ready for your mission!

1.2 Application

class asab.**Application**

The *Application* class maintains the global application state. You can provide your own implementation by creating a subclass. There should be only one *Application* object in the process.

Subclassing:

```
import asab

class MyApplication(asab.Application):
    pass

if __name__ == '__main__':
    app = MyApplication()
    app.run()
```

Direct use of *Application* object:

```
import asab

if __name__ == '__main__':
    app = asab.Application()
    app.run()
```

1.2.1 Event Loop

Application.**Loop**

The `asyncio` event loop that is used by this application.

```
asyncio.ensure_future(my_coro(), loop=Application.Loop)
```

1.2.2 Application Lifecycle

The application lifecycle is divided into 3 phases: init-time, run-time and exit-time.

Init-time

Application.**__init__**()

The init-time happens during *Application* constructor call. The Publish-Subscribe message *Application.init!* is published during init-time. The *Config* is loaded during init-time.

Application.**initialize**()

The application object executes asynchronous callback `Application.initialize()`, which can be overriden by an user.

```
class MyApplication(asab.Application):
    async def initialize(self):
        # Custom initialization
        from module_sample import Module
        self.add_module(Module)
```

Run-time

`Application.run()`

Enter a run-time. This is where the application spends the most time typically. The Publish-Subscribe message *`Application.run!`* is published when run-time begins.

The method returns the value of *`Application.ExitCode`*.

`Application.main()`

The application object executes asynchronous callback `Application.main()`, which can be overriden. If `main()` method is completed without calling `stop()`, then the application server will run forever (this is the default behaviour).

```
class MyApplication(asab.Application):
    async def main(self):
        print("Hello world!")
        self.stop()
```

`Application.stop(`*exit_code:int=None*`)`

The method `Application.stop()` gracefully terminates the run-time and commence the exit-time. This method is automatically called by `SIGINT` and `SIGTERM`. It also includes a response to `Ctrl-C` on UNIX-like system. When this method is called 3x, it abruptly exits the application (aka emergency abort).

The parameter `exit_code` allows you to specify the application exit code (see *Exit-Time* chapter).

Note: You need to install `win32api` module to use `Ctrl-C` or an emergency abord properly with ASAB on Windows. It is an optional dependency of ASAB.

Exit-time

`Application.finalize()`

The application object executes asynchronous callback `Application.finalize()`, which can be overriden by an user.

```
class MyApplication(asab.Application):
    async def finalize(self):
        # Custom finalization
        ...
```

The Publish-Subscribe message *`Application.exit!`* is published when exit-time begins.

`Application.set_exit_code(`*exit_code:int, force:bool=False*`)`

Set the exit code of the application, see `os.exit()` in the Python documentation. If `force` is `False`, the exit code will be set only if the previous value is lower than the new one. If `force` is `True`, the exit code value is set to a `exit_code` disregarding the previous value.

`Application.ExitCode`

The actual value of the exit code.

The example of the exit code handling in the `main()` function of the application.

```
if __name__ == '__main__':
    app = asab.Application()
    exit_code = app.run()
    sys.exit(exit_code)
```

1.2.3 Module registry

For more details see *`Module`* class.

`Application.`**`add_module`**(*module_class*)

Initialize and add a new module. The `module_class` class will be instantiated during the method call.

```
class MyApplication(asab.Application):
    async def initialize(self):
        from my_module import MyModule
        self.add_module(MyModule)
```

`Application.`**`Modules`**

A list of modules that has been added to the application.

1.2.4 Service registry

Each service is identified by its unique service name. For more details see *`Service`* class.

`Application.`**`get_service`**(*service_name*)

Locate a service by its service name in a registry and return the `Service` object.

```
svc = app.get_service("service_sample")
svc.hello()
```

`Application.`**`Services`**

A dictionary of registered services.

1.2.5 Command-line parser

`Application.`**`parse_args`**()

The application object calls this method during init-time to process a command-line arguments. `argparse` is used to process arguments. You can overload this method to provide your own implementation of command-line argument parser.

`Application.`**`Description`**

The `Description` attribute is a text that will be displayed in a help text (`--help`). It is expected that own value will be provided. The default value is `""` (empty string).

1.3 Configuration

asab.**Config**

The configuration is provided by *Config* object which is a singleton. It means that you can access *Config* from any place of your code, without need of explicit initialisation.

```
import asab

# Initialize application object and hence the configuration
app = asab.Application()

# Access configuration values anywhere
my_conf_value = asab.Config['section_name']['key1']
```

1.3.1 Based on ConfigParser

The *Config* is inherited from Python Standard Library `configparser.ConfigParser` class. which implements a basic configuration language which provides a structure similar to what's found in Microsoft Windows INI files.

class asab.config.**ConfigParser**

Example of the configuration file:

```
[bitbucket.org]
User = hg

[topsecret.server.com]
Port = 50022
ForwardX11 = no
```

And this is how you access configuration values:

```
>>> asab.Config['topsecret.server.com']['ForwardX11']
'no'
```

1.3.2 Automatic load of configuration

If a configuration file name is specified, the configuration is automatically loaded from a configuration file during initialiation time of *Application*. The configuration file name can be specified by one of `-c` command-line argument (1), `ASAB_CONFIG` environment variable (2) or config `[general] config_file` default value (3).

```
./sample_app.py -c ./etc/sample.conf
```

1.3.3 Including other configuration files

You can specify one or more additional configuration files that are loaded and merged from an main configuration file. It is done by `[general] include` configuration value. Multiple paths are separated by `os.pathsep` (`:` on Unix). The path can be specified as a glob (e.g. use of `*` and `?` wildcard characters), it will be expanded by `glob` module from Python Standard Library. Included configuration files may not exists, this situation is silently ignored.

```
[general]
include=./etc/site.conf:./etc/site.d/*.conf
```

1.3.4 Configuration default values

`Config.add_defaults(dictionary)`

This is how you can extend configuration default values:

```
asab.Config.add_defaults(
    {
        'section_name': {
            'key1': 'value',
            'key2': 'another value'
        },
        'other_section': {
            'key3': 'value',
        },
    }
)
```

1.3.5 Environment variables in configration

Environment variables found in values are automaticall expanded.

```
[section_name]
persistent_dir=${HOME}/.myapp/
```

```
>>> asab.Config['section_name']['persistent_dir']
'/home/user/.myapp/'
```

1.4 Logging

ASAB logging is built on top of a standard Python `logging` module. It means that it logs to `stderr` when running on a console and ASAB also provides file and syslog output (both RFC5424 and RFC3164) for background mode of operations.

Log timestamps are captured with sub-second precision (depending on the system capabilities) and displayed including microsecond part.

1.4.1 Recommended use

We recommend to create a logger `L` in every module that captures all necessary logging output. Alternative logging strategies are also supported.

```
import logging
L = logging.getLogger(__name__)

...

L.info("Hello world!")
```

Example of the output to the console:

```
25-Mar-2018 23:33:58.044595 INFO myapp.mymodule :  Hello world!
```

1.4.2 Verbose mode

The command-line argument `-v` enables verbose logging, respectively sets `logging.DEBUG` and enables `asyncio` debug logging.

The actual verbose mode is avaiable at `asab.Config["logging"]["verbose"]` boolean option.

1.4.3 Logging Levels

ASAB uses Python logging levels with the addition of `LOG_NOTICE` level. `LOG_NOTICE` level is similar to `logging.INFO` level but it is visible in even in non-verbose mode.

Level	Numeric value	Syslog Severity level
`CRITICAL`	50	Critical / `crit` / 2
`ERROR`	40	Error / `err` / 3
`WARNING`	30	Warning / `warning` / 4
`LOG_NOTICE`	25	Notice / `notice` / 5
`INFO`	20	Informational / `info` / 6
`DEBUG`	10	Debug / `debug` / 7
`NOTSET`	0	

1.4.4 Structured data

ASAB supports a structured data to be added to a log entry. It follows the RFC 5424, section `STRUCTURED-DATA`. Structured data are a dictionary, that has to be seriazable to JSON.

```
L.info("Hello world!", struct_data={'key1':'value1', 'key2':2})
```

1.4.5 Logging to file

The command-line argument `-l` on command-line enables logging to file. ASAB supports a log rotation mechanism. A log rotation is triggered by a UNIX signal `SIGHUP`.

It is implemented using `logging.handlers.RotatingFileHandler` from a Python standard library.

A configuration section `[[logging:file]]` can be used to specify details about desired syslog logging.

Example of the configuration file section:

```
[[logging:file]]
path=/var/log/asab.log
format="%%(asctime)s %%(levelname)s %%(name)s %%(struct_data)s%%(message)s",
datefmt="%%d-%%b-%%Y %%H:%%M:%%S.%%f"
backup_count=0
```

Note: Putting non-empty `path` option in the configuration file is the equivalent for `-l` argument respectively it enables logging to file as well.

1.4.6 Logging to syslog

The command-line argument `-s` enables logging to syslog.

A configuration section `[[logging:syslog]]` can be used to specify details about desired syslog logging.

Example of the configuration file section:

```
[[logging:syslog]]
enabled=true
format=5
address=tcp://syslog.server.lan:1554/
```

`enabled` is equivalent to command-line switch `-s` and it enables syslog logging target.

`format` speficies which logging format will be used. Possible values are:

- `5` for (new) syslog format (RFC 5424) ,
- `3` for old BSD syslog format (RFC 3164), typically used by `/dev/log` and
- `m` for Mac OSX syslog flavour that is based on BSD syslog format but it is not fully compatible.

The default value is `3` on Linux and `m` on Mac OSX.

`address` specifies the location of the Syslog server. It could be a UNIX path such as `/dev/log` or URL. Possible URL values:

- `tcp://syslog.server.lan:1554/` for Syslog over TCP
- `udp://syslog.server.lan:1554/` for Syslog over UDP
- `unix-connect:///path/to/syslog.socket` for Syslog over UNIX socket (stream)
- `unix-sendto:///path/to/syslog.socket` for Syslog over UNIX socket (datagram), equivalent to `/path/to/syslog.socket`, used by a `/dev/log`.

The default value is a `/dev/log` on Linux or `/var/run/syslog` on Mac OSX.

1.4.7 Reference

class `asab.log.`**`AsyncIOHandler`**(*loop*, *family*, *sock_type*, *address*, *facility=17*)

Bases: `logging.Handler`

A logging handler similar to a standard `logging.handlers.SocketHandler` that utilizes `asyncio`. It implements a queue for decoupling logging from a networking. The networking is fully event-driven via `asyncio` mechanisms.

emit(*record*)

This is the entry point for log entries.

class `asab.log.`**`Logging`**(*app*)

Bases: `object`

rotate()

class `asab.log.`**`MacOSXSyslogFormatter`**(*fmt=None*, *datefmt=None*, *style='%'*, *sd_id='sd'*)

Bases: *`asab.log.StructuredDataFormatter`*

It implements Syslog formatting for Mac OSX syslog (aka format `m`).

class asab.log.**StructuredDataFormatter**(*facility=16*, *fmt=None*, *datefmt=None*, *style='%'*, *sd_id='sd'*)

Bases: logging.Formatter

The logging formatter that renders log messages that includes structured data.

empty_sd = ''

format(*record*)

Format the specified record as text.

formatTime(*record*, *datefmt=None*)

Return the creation time of the specified LogRecord as formatted text.

render_struct_data(*struct_data*)

Return the string with structured data.

class asab.log.**SyslogRFC3164Formatter**(*fmt=None*, *datefmt=None*, *style='%'*, *sd_id='sd'*)

Bases: *asab.log.StructuredDataFormatter*

It implements Syslog formatting for Mac OSX syslog (aka format 3).

class asab.log.**SyslogRFC5424Formatter**(*fmt=None*, *datefmt=None*, *style='%'*, *sd_id='sd'*)

Bases: *asab.log.StructuredDataFormatter*

It implements Syslog formatting for Mac OSX syslog (aka format 5).

empty_sd = ' '

1.5 Metrics

class asab.metrics.**Metrics**(*app*)

class asab.metrics.**Module**(*app*)

Bases: asab.abc.module.Module

1.6 Publish-Subscribe

Publish–subscribe is a messaging pattern where senders of messages, called publishers, send the messages to receivers, called subscribers, via PubSub message bus. Publishers don't directly interact with subscribers in any way. Similarly, subscribers express interest in one or more message types and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

class asab.**PubSub**(*app*)

ASAB PubSub operates with a simple messages, defined by their *message type*, which is a string. We recommend to add ! (explamation mark) at the end of the message type in order to distinguish this object from other types such as Python class names or functions. Example of the message type is e.g. *Application.run!* or *Application.tick/600!*.

The message can carry an optional positional and keyword arguments. The delivery of a message is implemented as a the standard Python function.

Note: There is an default, application-wide Publish-Subscribe message bus at *Application.PubSub* that can be used to send messages. Alternatively, you can create your own instance of *PubSub* and enjoy isolated PubSub delivery space.

1.6.1 Subscription

`PubSub.`**`subscribe`**(*message_type*, *callback*)

Subscribe to a message type. Messages will be delivered to a `callback` callable (function or method). The `callback` can be a standard callable or an `async` coroutine. Asynchronous `callback` means that the delivery of the message will happen in a coroutine, asynchronously.

`Callback` callable will be called with the first argument

Example of a subscription to an *`Application.tick!`* messages.

```
class MyClass(object):
    def __init__(self, app):
        app.PubSub.subscribe("Application.tick!", self.on_tick)

    def on_tick(self, message_type):
        print(message_type)
```

Asynchronous version of the above:

```
class MyClass(object):
    def __init__(self, app):
        app.PubSub.subscribe("Application.tick!", self.on_tick)

    async def on_tick(self, message_type):
        await asyncio.sleep(5)
        print(message_type)
```

`PubSub.`**`subscribe_all`**(*obj*)

To simplify the process of subscription to *`PubSub`*, ASAB offers the decorator-based *"subscribe all"* functionality.

In the followin example, both `on_tick()` and `on_exit()` methods are subscribed to *`Application.PubSub`* message bus.

```
class MyClass(object):
    def __init__(self, app):
        app.PubSub.subscribe_all(self)

    @asab.subscribe("Application.tick!")
    async def on_tick(self, message_type):
        print(message_type)

    @asab.subscribe("Application.exit!")
    def on_exit(self, message_type):
        print(message_type)
```

`PubSub.`**`unsubscribe`**(*message_type*, *callback*)

Unsubscribe from a message delivery.

`class` `asab.`**`Subscriber`**(*pubsub=None*, **message_types*)

`Subscriber` object allows to consume PubSub messages in coroutines. It subscribes for various message types and consumes them. It works on FIFO basis (First message In, first message Out). If `pubsub` argument is None, the initial subscription is skipped.

```
subscriber = asab.Subscriber(
    app.PubSub,
    "Application.tick!",
```

(continues on next page)

(continued from previous page)

```
    "Application.stop!"
)
```

message()

Wait for a message asynchronously. Returns a three-members tuple `(message_type, args, kwargs)`.

 # Use in await statement message = await subscriber.message()

subscribe(*pubsub*, *message_type*)

Subscribe for more message types. This method can be called many times with various `pubsub` objects.

1.6.2 Publishing

`PubSub.`**publish**(*message_type*, **args*, ***kwargs*)

Publish a message to the PubSub message bus. It will be delivered to each subscriber synchronously. It means that the method returns after each subscribed `callback` is called.

The example of a message publish to the *`Application.PubSub`* message bus:

```
def my_function(app):
    app.PubSub.publish("mymessage!")
```

Asynchronous message delivery can be trigged by providing `asynchronously=True` keyword argument. Each subscriber is then handled in a dedicated `Future` object. The method returns immediatelly and the delivery of the message to subscribers happens, when control returns to the event loop.

The example of a **asynchronous version** of a message publish to the *`Application.PubSub`* message bus:

```
def my_function(app):
    app.PubSub.publish("mymessage!", asynchronously=True)
```

1.6.3 Application-wide PubSub

`Application.`**PubSub**

The ASAB provides the application-wide Publish-Subscribe message bus.

Well-Known Messages

`Application.init!`

This message is published when application is in the init-time. It is actually one of the last things done in init-time, so the application environment is almost ready for use. It means that configuration is loaded, logging is setup, the event loop is constructed etc.

`Application.run!`

This message is emitted when application enters the run-time.

`Application.stop!`

This message is emitted when application wants to stop the run-time. It can be sent multiple times because of a process of graceful run-time termination. The first argument of the message is a counter that increases with every `Application.stop!` event.

Application.exit!

This message is emitted when application enter the exit-time.

Application.tick!

Application.tick/10!

Application.tick/60!

Application.tick/300!

Application.tick/600!

Application.tick/1800!

Application.tick/3600!

Application.tick/43200!

Application.tick/86400!

The application publish periodically "tick" messages. The default tick frequency is 1 second but you can change it by configuration `[general] tick_period`. *`Application.tick!`* is published every tick. *`Application.tick/10!`* is published every 10th tick and so on.

Application.hup!

This message is emitted when application receives UNIX signal `SIGHUP` or equivalent.

1.7 Service

class asab.**Service**(*app*)

Service objects are registered at the service registry, managed by an application object. See *`Application.Services`* for more details.

An example of a typical service class skeleton:

```
class MyService(asab.Service):

    def __init__(self, app, service_name):
        super().__init__(app, service_name)
        ...

    async def initialize(self, app):
        ...


    async def finalize(self, app):
        ...


    def service_method(self):
        ....
```

This is how a service is created and registered:

```
mysvc = MyService(app, "my_service")
```

This is how a service is located and used:

```
mysvc = app.get_service("my_service")
mysvc.service_method()
```

`Service.`**`Name`**

Each service is identified by its name.

1.7.1 Lifecycle

`Service.`**`initialize`**(*app*)

Called when the service is initialized. It can be overriden by an user.

`Service.`**`finalize`**(*app*)

Called when the service is finalized e.g. during application exit-time. It can be overriden by an user.

1.8 Module

`class` `asab.`**`Module`**

Modules are registered at the module registry, managed by an application object. See `Application.Modules` for more details. Module can be loaded by ASAB and typically provides one or more `Service` objects.

1.8.1 Structure

Recommended structure of the ASAB module:

```
mymodule/
    __init__.py
    myservice.py
```

Content of the *__init__.py*:

```
import asab
from .myservice import MyService

# Extend ASAB configuration defaults
asab.Config.add_defaults({
    'mymodule': {
        'foo': 'bar'
    }
})

class MyModule(asab.Module):
    def __init__(self, app):
        super().__init__(app)
        self.service = MyService(app, "MyService")
```

And this is how the module is loaded:

```
from mymodule import MyModule
app.add_module(MyModule)
```

For more details see `Application.add_module`.

1.8.2 Lifecycle

`Module.initialize(app)`

Called when the module is initialized. It can be overriden by an user.

`Module.finalize(app)`

Called when the module is finalized e.g. during application exit-time. It can be overriden by an user.

1.9 Various utility classes

1.9.1 Singleton

class `asab.abc.singleton.Singleton`

The singleton pattern is a software design pattern that restricts the instantiation of a class to one object.

Note: The implementation idea is borrowed from "Creating a singleton in Python" question on StackOverflow.

Usage:

```
import asab

class MyClass(metaclass=asab.Singleton):
    ...
```

1.9.2 Persistent dictionary

class `asab.pdict.PersistentDict(path)`

Bases: `collections.abc.MutableMapping`

The persistent dictionary works as the regular Python dictionary but the content of the dictionary is stored in the file. You cat think of a `PersistentDict` as a simple key-value store. It is not optimized for a frequent access. This class provides common `dict` interface.

Warning: You can only store objects in the persistent dictionary that are seriazable.

load([*keys*]) → [values].

Optimised version of the get() operations that load multiple keys from the persistent store at once. It saves IO in exchange for possible race conditions.

Parameters **keys** – A list of keys.

Returns A list of values in the same order to provided key list.

```
v1, v2, v3 = pdict.load('k1', 'k2', 'k3')
```

update([*E*], ***F*) → None.

Update D from mapping/iterable E and F.

- If E present and has a .keys() method, does: for k in E: D[k] = E[k]
- If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v
- In either case, this is followed by: for k, v in F.items(): D[k] = v

Inspired by a https://github.com/python/cpython/blob/3.6/Lib/_collections_abc.py

Note: A recommended way of initializing the persistent dictionary:

```
PersistentState = asab.PersistentDict("some.file")
PersistentState.setdefault('foo', 0)
PersistentState.setdefault('bar', 2)
```

1.9.3 Timer

class `asab.timer.`**`Timer`**(*app*, *handler*, *autorestart=False*) → Timer.

Bases: `object`

The relative and optionally repeating timer for asyncio.

This class is simple relative timer that generate an event after a given time, and optionally repeating in regular intervals after that.

Parameters

- **app** – An ASAB application.
- **handler** – A coro or future that will be called when a timer triggers.
- **autorestart** (*boolean*) – If *True* then a timer will be automatically restarted after triggering.

Variables

- **Handler** – A coro or future that will be called when a timer triggers.
- **Task** – A future that represent the timer task.
- **App** – An ASAB app.
- **AutoRestart** (*boolean*) – If *True* then a timer will be automatically restarted after triggering.

The timer object is initialized as stopped.

Note: The implementation idea is borrowed from "Python - Timer with asyncio/coroutine" question on StackOverflow.

is_started() → boolean

Return *True* is the timer is started otherwise returns *False*.

restart(*timeout*)

Restart the timer.

Parameters **timeout** (*float/int*) – A timer delay in seconds.

start(*timeout*)

Start the timer.

Parameters **timeout** (*float/int*) – A timer delay in seconds.

stop()

Stop the timer.

1.9.4 Sockets

class `asab.socket.`**`StreamSocketServerService`**(*app*)

Bases: `asab.abc.service.Service`

Example of use:

class ServiceMyProtocolServer(asab.StreamSocketServerService):

async def initialize(self, app): host = asab.Config.get('http', 'host') port = asab.Config.getint('http', 'port')

L.debug("Starting server on {} {} ...".format(host, port)) await self.create_server(app, MyProtocol, [(host, port)])

create_server (*app*, *protocol*, *addrs*)

finalize (*app*)

1.10 The web server

ASAB provides a web server in a `asab.web` module. This module offers an integration of a `aiohttp` web server.

1. Before you start, make sure that you have `aiohttp` module installed.

```
$ pip3 install aiohttp
```

2. The following code creates a simple web server application

```
#!/usr/bin/env python3
import asab
import aiohttp

class MyApplication(asab.Application):

    async def initialize(self):
        # Load the web service module
        from asab.web import Module
        self.add_module(Module)

        # Locate the web service
        svc = self.get_service("asab.WebService")

        # Add a route
        svc.WebApp.router.add_get('/hello', self.hello)

    # Simplistic view
    async def hello(self, request):
        return aiohttp.web.Response(text='Hello!\n')

if __name__ == '__main__':
    app = MyApplication()
    app.run()
```

3. Test it with *curl*

```
$ curl http://localhost:8080/hello
Hello!
```

1.10.1 Web Service

class asab.web.service.**WebService**

Service localization example:

```
from asab.web import Module
self.add_module(Module)
svc = self.get_service("asab.WebService")
```

`WebService.`**`Webapp`**

An instance of a *aiohttp.web.Application* class.

```
svc.WebApp.router.add_get('/hello', self.hello)
```

1.10.2 Sessions

ASAB Web Service provides an implementation of the web sessions.

class `asab.web.session.`**`ServiceWebSession`**

TODO: ...

`asab.web.session.`**`session_middleware`**(*storage*)

TODO: ...

1.11 The message-oriented middleware

Message-oriented middleware (MOM) sends and receive messages between distributed systems. MOM allows application components to be distributed over heterogeneous platforms and reduces the complexity of developing such applications. The middleware creates a distributed communications layer that insulates the application developer from the details of the various network interfaces. It is a typical component of the microservice architecture, used for asynchronous tasks, complements synchronous HTTP REST API.

MOM is typically integrated with Message Queue servers such as RabbitMQ or Kafka. Messages are distributed thru these systems from and to various brokers. A message routing mechanism can be added to MQ server to steer a flow of the messages, if needed.

More theory can be found here: https://en.wikipedia.org/wiki/Message-oriented_middleware

1.11.1 MOM Service

class `asab.mom.service.`**`MOMService`**

Message-oriented middleware is provided by a *`MOMService`* in a `asab.mom` module.

Service initialization and localization example:

```
from asab.mom import Module
self.add_module(Module)
svc = self.get_service("asab.MOMService")
```

1.11.2 Broker

class `asab.mom.broker.`**`Broker`**

The broker is an object that provides methods for sending and receiving messages. It is also responsible for a underlaying transport of messages e.g. over the network to other brokers or MQ servers.

A base broker class `Broker` cannot be created directly, see available brokers below. Broker creating example:

```
from asab.mom.amqp import AMQPBroker
broker = AMQPBroker(app, config_section_name="bsfrgeocode:amqp")
```

Note: MOM Service has to be initialized.

Sending messages

`Broker.`**`publish`**(*self*, *body*, *target:str=''*, *correlation_id:str=None*)

Publishe the message to a MQ server.

```
message = "Hello World!"
await broker.publish(message, target="example")
```

Receiving messages

`Broker.`**`subscribe`**(*subscription:str*)

Subscribe the broker to a specific subscription (e.g. topic or queue) on the MQ server. Once completed, messages starts to flow in and they are *routed* based on the target.

`Broker.`**`add`**(*target:str*, *handler*, *reply_to:str=None*)

A message *handler* must be a coroutine that accept *properties* and *body* of the incoming message. Incoming messages are routed based on their *target* to a specific handler. If there is no registered handler for a target, the message is discarded.

```
broker.subscribe("topic")
broker.add('example', example_handler)

async def example_handler(self, properties, body):
        print("Recevied", body)
```

Replying to a message

Message-oriented middleware is the asynchronous message passing model. By a mechanism of a message correlation, MOM service allow to reply to a message in the handler.

Example of the handler:

```
async def example_handler(self, properties, body):
        print("Recevied", body)
        return "Hi there too"
```

Available brokers

`class` `asab.mom.amqp.`**`AMQPBroker`**

1.12 Installation

ASAB is distributed via pypi.

1.12.1 Install ASAB using pip

This is the recommended installation method.

```
$ pip install asab
```

1.12.2 Install ASAB using easy_install

```
$ easy_install asab
```

1.12.3 Install ASAB for a GitHub

To install ASAB from a master branch of the GIT repository, use following command.

Note: Git has to be installed in order to successfuly complete the installation.

```
$ pip install git+https://github.com/TeskaLabs/asab.git
```

1.13 ASAB Command-line interface

ASAB-based application provides the command-line interface by default. Here is an overview of the common command-line arguments.

-h, --help

Show a help.

1.13.1 Configuration

-c `<CONFIG>,--config <CONFIG>`

Load configuration file from a file CONFIG.

1.13.2 Logging

-v, --verbose

Increase the logging level to DEBUG aka be more verbose about what is happening.

-l `<LOG_FILE>,--log-file <LOG_FILE>`

Log to a file LOG_FILE.

-s, --syslog

Log to a syslog.

1.13.3 Daemon

Python module `python-daemon` has to be installed in order to support daemonosation functions.

```
$ pip3 install asab python-daemon
```

`-d, --daemonize`

Launch the application in the background aka daemonized.

Daemon-related section of *`Config`* file:

```
[daemon]
pidfile=/var/run/myapp.pid
uid=nobody
gid=nobody
working_dir=/tmp
```

Configuration options `pidfile`, `uid` , `gid` and `working_dir` are supported.

`-k, --kill`

Shutdown the application running in the background (started previously with `-d` argument).

1.14 Containerisation

ASAB is designed for deployment into containers such as LXC/LXD or Docker. It allows to build e.g. microservices that provides REST interface or consume MQ messages while being deployed into a container for a sake of the infrastructure flexibility.

1.14.1 ASAB in a LXC/LXD container

1. Prepare LXC/LXD container based on Alpine Linux

```
$ lxc launch images:alpine/3.8 asab
```

2. Swich into a container

```
$ lxc exec asab -- /bin/ash
```

3. Adjust a container

```
$ sed -i 's/^tty/# tty/g' /etc/inittab
```

4. Prepare Python3 environment

```
$ apk update
$ apk upgrade
$ apk add --no-cache python3

$ python3 -m ensurepip
$ rm -r /usr/lib/python*/ensurepip
$ pip3 install --upgrade pip setuptools
```

5. Deploy ASAB

```
$ pip3 install asab
```

6. Deploy dependencies

```
$ pip3 install asab python-daemon
```

7. (Optionally if you want to use `asab.web` module) install aiohttp dependecy

```
$ pip3 install aiohttp
```

8. Use OpenRC to automatically start/stop ASAB application

```
$ vi /etc/init.d/asab-app
```

Adjust the example of OpenRC init file.

```
$ chmod a+x /etc/init.d/asab-app
$ rc-update add asab-app
```

Note: If you need to install python packages that require compilation using C compiler, you have to add following dependencies:

```
$ apk add --virtual .buildenv python3-dev
$ apk add --virtual .buildenv gcc
$ apk add --virtual .buildenv musl-dev
```

And removal of the build tools after pip install:

```
$ apk del .buildenv
```

1.15 systemd

1. Create a new Systemd unit file in /etc/systemd/system/:

```
$ sudo vi /etc/systemd/system/asab.service
```

Adjust the example of SystemD unit file.

2. Let systemd know that there is a new service:

```
$ sudo systemctl enable asab
```

To reload existing unit file after changing, use this:

```
$ sudo systemctl daemon-reload
```

3. ASAB Application Server service for systemd is now ready.

1.15.1 Start of ASAB Server

```
$ sudo service asab start
```

1.15.2 Stop of ASAB Server

```
$ sudo service asab stop
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

Symbols

A

B

C

E

F

G

I

L

M

P

R

S

T

U

W