asphalt-wamp

Release 2.2.2

Mar 02, 2018

Contents

This Asphalt framework component provides a WAMP (Web Application Message Protocol) client, implemented on top of the autobahn library.

CHAPTER 1

Configuration

WAMP, being a routed protocol, requires a router to connect to. If you do not have one already, the reference implementation, Crossbar, should work nicely. The recommended way of setting it up is with Docker, though setting up a dedicated virtualenv for it would also do the trick.

Most WAMP clients need very little configuration. You usually have to set the realm name, host name (if not running on localhost) and port (if not running on port 8080) and TLS, if connecting to a remote instance securely.

Suppose you're connecting to realm `myrealm` on `crossbar.example.org`, port 8181 using TLS, your configuration would look like this:

```
components:
    wamp:
      realm: myrealmname
      host: crossbar.example.org
      port: 8181
      tls: true
```

Your wamp client resource (`default`) would then be accessible on the context as `ctx.wamp`.

1.1 Multiple clients

You can also configure multiple WAMP clients if necessary. For that, you will need to have a structure along the lines of:

```
components:
    wamp:
      tls: true
      clients:
        wamp1:
          realm: myrealmname
          host: crossbar.example.org
          port: 8181
```

(continues on next page)

(continued from previous page)

```
        wamp2:
          realm: otherrealm
          host: crossbar.company.com
```

In this example, two client resources (`wamp1 / ctx.wamp1` and `wamp2 / ctx.wamp2`) are created. The first one is like the one in the previous example. The second connects to the realm named `otherrealm` on `crossbar.company.com` on the default port using TLS. Setting `tls:  true` (or any other option) on the same level as `clients` means it's the default value for all clients.

For a comprehensive list of all client options, see the documentation of the the `WAMPClient` class.

CHAPTER 2

User guide

The following sections explain how to use the most common functions of a WAMP client. The more advanced options have been documented in the API reference.

For practical examples, see the examples directory.

2.1 Calling remote procedures

To call a remote procedure, use the `call()` method:

```
result = await ctx.wamp.call('procedurename', arg1, arg2, arg3='foo')
```

To receive progressive results from the call, you can give a callback as the `on_progress` option:

```
def progress(status):
    print('operation status: {}'.format(status))

result = await ctx.wamp.call('procedurename', arg1, arg2, arg3='foo',
                             options=dict(on_progress=progress))
```

To set a time limit for how long to wait for the call to complete, use the `timeout` option:

```
# Wait 10 seconds until giving up
result = await ctx.wamp.call('procedurename', arg1, arg2, arg3='foo',␣
↪options=dict(timeout=10))
```

Note: This will **not** stop the remote handler from finishing; it will just make the client stop waiting and discard the results of the call.

2.2 Registering procedure handlers

To register a procedure on the router, create a callable that takes a `CallContext` as the first argument and use the `call()` method to register it:

```
async def procedure_handler(ctx: CallContext, *args, **kwargs):
    ...

await ctx.wamp.register(procedure_handler, 'my_remote_procedure')
```

The handler can be either an asynchronous function or a regular function, but the latter will obviously have fewer use cases due to the lack of `await`.

To send progressive results, you can call the `progress` callback on the `CallContext` object. For this to work, the caller must have used the `on_progress` option when making the call. Otherwise `progress` will be `None`.

For example:

```
async def procedure_handler(ctx: CallContext, *args, **kwargs):
    for i in range(1, 11):
        await asyncio.sleep(1)
        if ctx.progress:
            ctx.progress('{}% complete'.format(i * 10))

    return 'Done'

await ctx.wamp.register(procedure_handler, 'my_remote_procedure')
```

2.3 Publishing messages

To publish a message on the router, call `publish()` with the topic as the first argument and then add any positional and keyword arguments you want to include in the message:

```
await ctx.wamp.publish('some_topic', 'hello', 'world', another='argument')
```

By default, publications are not acknowledged by the router. This means that a published message could be silently discarded if, for example, the publisher does not have proper permissions to publish it. To avoid this, use the `acknowledge` option:

```
await ctx.wamp.publish('some_topic', 'hello', 'world', another='argument',
                       options=dict(acknowledge=True))
```

2.4 Subscribing to topics

You can use the `subscribe()` method to receive published messages from the router:

```
async def subscriber(ctx: EventContext, *args, **kwargs):
    print('new message: args={}, kwargs={}'.format(args, kwargs))

await ctx.wamp.subscribe(subscriber, 'some_topic')
```

Just like procedure handlers, subscription handlers can be either an asynchronous or regular functions.

2.5 Mapping WAMP exceptions to Python exceptions

Exceptions transmitted over WAMP are identified by a specific URI. WAMP errors can be mapped to Python exceptions by linking a specific URI to a specific exception class by means of either `exception()`, `map_exception()` or `map_exception()`.

When you map an exception, you can raise it in your procedure or subscription handlers and it will be automatically translated using the given error URI so that the recipients will be able to properly map it on their end as well. Likewise, when a matching error is received from the router, the appropriate exception class is instantiated and raised in the calling code.

Any unmapped exceptions manifest themselves as `ApplicationError` exceptions.

2.6 Using registries to structure your application

While it may at first seem convenient to register every procedure and subscription handler using `register()` and `subscribe()`, it does not scale very well when your handlers are distributed over several packages and modules.

The `WAMPRegistry` class provides an alternative to this. Each registry object stores registered procedure handlers, subscription handlers and mapped exceptions, and can apply defaults on each of these. Each registry can have a separate namespace prefix so you don't have to repeat it in every single procedure name, topic or mapped error.

Suppose you want to register two procedures and one subscriber, all under the `foo` prefix and you want to apply the `invoke='roundrobin'` setting to all procedures:

```
from asphalt.wamp import WAMPRegistry

registry = WAMPRegistry('foo', procedure_defaults={'invoke': 'roundrobin'})


@registry.procedure
def multiply(ctx, factor1, factor2):
    return factor1 * factor2


@registry.procedure
def divide(ctx, numerator, denominator):
    return numerator / denominator


@registry.subscriber
def message_received(ctx, message):
    print('new message: %s' % message)
```

To use the registry, pass it to the WAMP component as an option:

```
class ApplicationComponent(ContainerComponent):
    async def start(ctx):
        ctx.add_component('wamp', registry=registry)
        await super.start(ctx)
```

This will register the `foo.multiply`, `foo.divide` procedures and a subscriptions for the `foo.message_received` topic.

Say your procedures and/or subscribers are spread over several modules and you want a different namespace for every module, you could have a separate registry in every module and then combine them into a single registry using `add_from()`:

```python
from asphalt.wamp import WAMPRegistry

from myapp.services import accounting, deliveries, production  # these are modules

registry = WAMPRegistry()
registry.add_from(accounting.registry, 'accounting')
registry.add_from(deliveries.registry, 'deliveries')
registry.add_from(production.registry, 'production')
```

You can set the prefix either in the call to `add_from()` or when creating the registry of each subsection. Note that if you do both, you end up with two prefixes!

CHAPTER 3

Implementing dynamic authentication and authorization

While static configuration of users and permissions may work for trivial applications, you will probably find yourself wanting for more flexibility for both authentication and authorization as your application grows larger. Crossbar, the reference WAMP router implementation, supports *dynamic authentication* and *dynamic authorization*. That means that instead of a preconfigured list of users or permissions, the router itself will call named remote procedures to determine whether the credentials are valid (authentication) or whether a session has permission to register/call a procedure or subscribe/publish to a topic (authorization).

The catch-22 in this is that the WAMP client that *provides* these procedures has to have permission to register these procedures. This chicken and egg problem can be solved by providing a trusted backdoor for this particular client. In the example below, the client providing the authenticator and authorizer services connects via port 8081 which will be only made accessible for that particular client. Unlike the other two configured roles, the `server` role has a static authorization configuration, which is required for this to work.

```
version: 2
workers:
- type: router
  realms:
  - name: myrealm
    roles:
    - name: regular
      authorizer: authorize
    - name: admin
      authorizer: authorize
    - name: server
      permissions:
      - uri: "*"
        allow: {call: true, publish: true, register: true, subscribe: true}
  transports:
  - type: websocket
    endpoint:
      type: tcp
      port: 8080
    auth:
      ticket:
```

(continues on next page)

(continued from previous page)

```
        type: dynamic
        authenticator: authenticate
  - type: websocket
    endpoint:
      type: tcp
      port: 8081
    auth:
      anonymous:
        type: static
        role: server
```

The client performing the `server` role will then register the `authenticate()` and `authorize()` procedures on the router:

```
from typing import Dict

from asphalt.core import ContainerComponent
from asphalt.wamp import CallContext, WAMPRegistry
from autobahn.wamp.exception import ApplicationError

registry = WAMPRegistry()
users = {
    'joe_average': ('1234', 'regular'),
    'bofh': ('B3yt&4_+', 'admin')
}


@registry.procedure
def authenticate(ctx: CallContext, realm: str, auth_id: str, details: Dict[str, Any]):
    # Don't do this in real apps! This is a security hazard!
    # Instead, use a password hashing algorithm like argon2, scrypt or bcrypt
    user = users.get(authid)
    if user:
        # This applies for "ticket" authentication as configured above
        password, role = user
        if password == details['ticket']:
            return {'authrole': role}

    raise ApplicationError(ApplicationError.AUTHENTICATION_FAILED, 'Authentication␣
↪failed')


@registry.procedure
def authorize(ctx: CallContext, session: Dict[str, Any], uri: str, action: str):
    # Cache any positive answers
    if session['authrole'] == 'regular':
        # Allow regular users to call and subscribe to public.*
        if action in ('call', 'subscribe') and uri.startswith('public.'):
            return {'allow': True, 'cache': True}
    elif session['authrole'] == 'admin':
        # Allow admins to call, subscribe and publish anything anywhere
        # (but not register procedures)
        if action in ('call', 'subscribe', 'publish'):
            return {'allow': True, 'cache': True}

    return {'allow': False}
```

(continues on next page)

(continued from previous page)

```


class ServerComponent(ContainerComponent):
    async def start(ctx):
        ctx.add_component('wamp', registry=registry)
        await super().start(ctx)
```

For more information, see the Crossbar documentation:

- Dynamic authentication
- Dynamic authorization

Warning: At the time of this writing (2017-04-29), caching of authorizer responses has not been implemented in Crossbar. This documentation assumes that it will be present in a future release.

CHAPTER 4

Version history

This library adheres to Semantic Versioning.

2.2.2 (2018-03-02)

- Fixed error in `Client.stop()` when the session is already `None`

2.2.1 (2018-02-22)

- Fixed mapped custom exceptions being reported via asphalt-exceptions

2.2.0 (2018-02-15)

- Added integration with asphalt-exceptions
- Raised connection logging level to `INFO`
- Added a configurable shutdown timeout
- Renamed `WAMPClient.close()` to `WAMPClient.stop()`
- Improved the reliability of the connection/session teardown process

2.1.0 (2017-09-21)

- Added the `protocol_options` option to `WAMPClient`
- Added the `connection_timeout` option to `WAMPClient`

2.0.1 (2017-06-07)

- Fixed failure to register option-less procedures and subscriptions added from a registry

2.0.0 (2017-06-07)

- **BACKWARD INCOMPATIBLE** Upgraded minimum Autobahn version to v17.5.1
- **BACKWARD INCOMPATIBLE** Changed the default value of the `path` option on `WAMPClient` to `/ws` to match the default Crossbar configuration
- **BACKWARD INCOMPATIBLE** Changed subscriptions to use the `details` keyword argument to accept subscription details (since `details_arg` is now deprecated in Autobahn)

- **BACKWARD INCOMPATIBLE** Replaced `SessionJoinEvent.session_id` with the `details` attribute which directly exposes all session details provided by Autobahn
- **BACKWARD INCOMPATIBLE** Changed the way registration/subscription/call/publish options are passed. Keyword arguments were replaced with a single `options` keyword-only argument.
- **BACKWARD INCOMPATIBLE** Registry-based subscriptions and exception mappings now inherit the parent prefixes, just like procedures did previously
- Added compatibility with Asphalt 4.0
- Added the `WAMPClient.details` property which returns the session details when joined to one
- Fixed error during `WAMPClient.close()` if a connection attempt was in progress
- Fixed minor documentation errors

1.0.0 (2017-04-29)

- Initial release
- API reference