autobahn

Release 17.5.1

May 23, 2017

Contents

CHAPTER 1

Autobahn|Python

Open-source (MIT) real-time framework for Web, Mobile & Internet of Things.

Autobahn|Python is part of the Autobahn project and provides open-source implementations of

- The WebSocket Protocol
- The Web Application Messaging Protocol (WAMP)

in Python 2 and 3, running on Twisted **or** asyncio.

Autobahn Features

WebSocket allows bidirectional real-time messaging on the Web while WAMP provides applications with high-level communication abstractions (remote procedure calling and publish/subscribe) in an open standard WebSocket-based protocol.

Autobahn|Python features:

- framework for WebSocket and WAMP clients
- compatible with Python 2.7 and 3.3+
- runs on CPython, PyPy and Jython
- runs under Twisted and asyncio
- implements WebSocket RFC6455 (and draft versions Hybi-10+)
- implements WebSocket compression

- implements WAMP, the Web Application Messaging Protocol
- supports TLS (secure WebSocket) and proxies
- Open-source (MIT license)

...and much more.

Further, **Autobahn**|Python is written with these goals:

1. high-performance, fully asynchronous and scalable code
2. best-in-class standards conformance and security

We do take those design and implementation goals quite serious. For example, **Autobahn**|Python has 100% strict passes with AutobahnTestsuite, the quasi industry standard of WebSocket protocol test suites we originally created only to test **Autobahn**|Python ;)

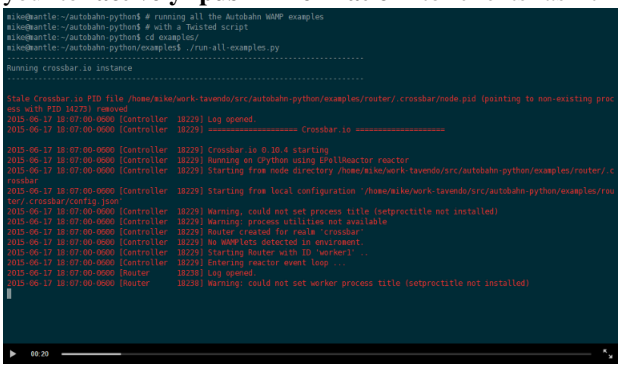
For (hopefully) current test reports from the Testsuite see

- WebSocket client functionality
- WebSocket server functionality

Note: In the following, we will just refer to **Autobahn** instead of the more precise term **Autobahn**|Python and there is no ambiguity.

What can I do with Autobahn?

WebSocket is great for apps like **chat**, **trading**, **multi-player games** or **real-time charts**. It allows you to **actively push information** to clients as it happens. (See also *Automatically Run All Examples*)



Further, WebSocket allows you to real-time enable your Web user interfaces: **always current information** without reloads or polling. UIs no longer need to be a boring, static thing. Looking for the right communication technology for your next-generation Web apps? Enter WebSocket.

And WebSocket works great not only on the Web, but also as a protocol for wiring up the **Internet-of-Things (IoT)**. Connecting a sensor or actor to other application components in real-time over an efficient protocol. Plus: you are using the *same* protocol to connect frontends like Web browsers.

While WebSocket already is quite awesome, it is still low-level. Which is why we have WAMP. WAMP allows you to **compose your application from loosely coupled components** that talk in real-time with each other - using nice high-level communication patterns ("Remote Procedure Calls" and "Publish & Subscribe").

WAMP enables application architectures with application code **distributed freely across processes and devices** according to functional aspects. Since WAMP implementations exist for **multiple languages**, WAMP applications can be **polyglot**. Application components can be implemented in a language and run on a device which best fit the particular use case.

WAMP is a routed protocol, so you need a WAMP router. We suggest using Crossbar.io, but there are also other implementations available.

More:

- WebSocket - Why, what, and - can I use it?
- Why WAMP?

Show me some code!

A sample **WebSocket server**:

```
from autobahn.twisted.websocket import WebSocketServerProtocol
# or: from autobahn.asyncio.websocket import WebSocketServerProtocol

    class MyServerProtocol(WebSocketServerProtocol):

       def onConnect(self, request):
           print("Client connecting: {}".format(request.peer))

       def onOpen(self):
           print("WebSocket connection open.")

       def onMessage(self, payload, isBinary):
           if isBinary:
               print("Binary message received: {} bytes".format(len(payload)))
           else:
               print("Text message received: {}".format(payload.decode('utf8')))

           ## echo back message verbatim
           self.sendMessage(payload, isBinary)

       def onClose(self, wasClean, code, reason):
           print("WebSocket connection closed: {}".format(reason))
```

Complete example code:

- WebSocket Echo (Twisted-based)
- WebSocket Echo (Asyncio-based)

Introduction to WebSocket Programming with Autobahn:

- *WebSocket Programming*

A sample **WAMP application component** implementing all client roles:

```
from autobahn.twisted.wamp import ApplicationSession
# or: from autobahn.asyncio.wamp import ApplicationSession
class MyComponent(ApplicationSession):

   @inlineCallbacks
   def onJoin(self, details):

       # 1) subscribe to a topic
       def onevent(msg):
```

```
            print("Got event: {}".format(msg))
        yield self.subscribe(onevent, 'com.myapp.hello')

        # 2) publish an event
        self.publish('com.myapp.hello', 'Hello, world!')

        # 3) register a procedure for remoting
        def add2(x, y):
            return x + y
        self.register(add2, 'com.myapp.add2');

        # 4) call a remote procedure
        res = yield self.call('com.myapp.add2', 2, 3)
        print("Got result: {}".format(res))
```

Complete example code:

- Twisted Example
- asyncio Example

Introduction to WAMP Programming with Autobahn:

- *WAMP Programming*

Where to start

To get started, jump to *Installation*.

For developers new to asynchronous programming, Twisted or asyncio, we've collected some useful pointers and information in *Asynchronous Programming*.

For **WebSocket developers**, *WebSocket Programming* explains all you need to know about using Autobahn as a WebSocket library, and includes a full reference for the relevant parts of the API.

WebSocket Examples lists WebSocket code examples covering a broader range of uses cases and advanced WebSocket features.

For **WAMP developers**, *WAMP Programming* gives an introduction for programming with WAMP in Python using Autobahn.

WAMP Examples lists WAMP code examples covering all features of WAMP.

Get in touch

Development of Autobahn takes place on the GitHub source repository.

Note: We are open for contributions, whether that's code or documentation! Preferably via pull requests.

We also take **bug reports** at the issue tracker.

The best place to **ask questions** is on the mailing list. We'd also love to hear about your project and what you are using Autobahn for!

Another option is StackOverflow where questions related to Autobahn are tagged "autobahn" (or "autobahnws").

The best way to **Search the Web** for related material is by using these (base) search terms:

- "autobahnpython"
- "autobahnws"

You can also reach users and developers on **IRC** channel `#autobahn` at freenode.net.

Finally, we are on Twitter.

Contributing

Autobahn is an open source project, and hosted on GitHub. The GitHub repository includes the documentation.

We're looking for all kinds of contributions - from simple fixes of typos in the code or documentation to implementation of new features and additions of tutorials.

If you want to contribute to the code or the documentation: we use the Fork & Pull Model.

This means that you fork the repo, make changes to your fork, and then make a pull request here on the main repo.

This article on GitHub gives more detailed information on how the process works.

In order to run the unit-tests, we use Tox to build the various test-environments. To run them all, simply run `tox` from the top-level directory of the clone.

For test-coverage, see the Makefile target `test_coverage`, which deletes the coverage data and then runs the test suite with various tox test-environments before outputting HTML annotated coverage to `./htmlcov/index.html` and a coverage report to the terminal.

There are two environment variables the tests use: `USE_TWISTED=1` or `USE_ASYNCIO=1` control whether to run unit-tests that are specific to one framework or the other.

See `tox.ini` for details on how to run in the different environments.

Release Testing

Before pushing a new release, three levels of tests need to pass:

1. the unit tests (see above)
2. the [WebSocket level tests](wstest/README.md)
3. the [WAMP level tests](examples/README.md) (*)

> (*): these will launch a Crossbar.io router for testing

Sitemap

Please see *Site Contents* for a full site-map.

CHAPTER 2

Installation

This document describes the prerequisites and the installation of **Autobahn**.

Requirements

Autobahn runs on Python on top of these networking frameworks:

- Twisted
- asyncio

You will need at least one of those.

Note: Most of Autobahn's WebSocket and WAMP features are available on both Twisted and asyncio, so you are free to choose the underlying networking framework based on your own criteria.

For Twisted installation, please see here. Asyncio comes bundled with Python 3.4+. For Python 3.3, install it from here. For Python 2, trollius will work.

Supported Configurations

Here are the configurations supported by Autobahn:

Python	Twisted	asyncio	Notes
CPython 2.7	yes	yes	asyncio support via trollius
CPython 3.3	yes	yes	asyncio support via tulip
CPython 3.4+	yes	yes	asyncio in the standard library
PyPy 2.2+	yes	yes	asyncio support via trollius
Jython 2.7+	yes	?	Issues: 1, 2

Performance Note

Autobahn is portable, well tuned code. You can further accelerate performance by

- Running under PyPy (recommended!) or
- on CPython, install the native accelerators wsaccel and ujson (you can use the install variant `acceleration` for that - see below)

To give you an idea of the performance you can expect, here is a blog post benchmarking Autobahn running on the RaspberryPi (a tiny embedded computer) under PyPy.

Installing Autobahn

Using Docker

We offer Docker Images with Autobahn pre-installed. To use this, if you have Docker already installed, just do

```
sudo docker run -it crossbario/autobahn-python:cpy2 python client.py
--url ws://IP _of_WAMP_router:8080/ws --realm realm1
```

This starts up a Docker container and *client.py*, which connects to a Crossbar.io router at the given URL and to the given realm.

There are several docker images to choose from, depending on whether you are using Python 2, 3 or PyPy (Python 2 only for now).

There are the flavors which are based on the official Python 2, 3 and PyPy images, plus Python 2 and 3 versions using Alpine Linux, which have a smaller footprint. (Note: Footprint only matters for the download once per machine, after that the cached image is used. Containers off the same image/layers only take up space corresponding to how different from the image they are, so image size is relatively less important when using multiple containers.)

Install from PyPI

To install Autobahn from the Python Package Index using Pip

```
pip install autobahn
```

You can also specify *install variants* (see below). E.g. to install Twisted automatically as a dependency

```
pip install autobahn[twisted]
```

And to install asyncio backports automatically when required

```
pip install autobahn[asyncio]
```

Install from Sources

To install from sources, clone the repository:

```
git clone git@github.com:crossbario/autobahn-python.git
```

checkout a tagged release:

```
cd AutobahnPython
git checkout v0.9.1
```

Warning: You should only use *tagged* releases, not *master*. The latest code from *master* might be broken, unfinished and untested. So you have been warned ;)

Then do:

```
cd autobahn
python setup.py install
```

You can also use `pip` for the last step, which allows to specify install variants (see below)

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

Install Variants

Autobahn has the following install variants:

Variant	Description
`twisted`	Install Twisted as a dependency
`asyncio`	Install asyncio as a dependency (or use stdlib)
`accelerate`	Install native acceleration packages on CPython
`compress`	Install packages for non-standard WebSocket compression methods
`serialization`	Install packages for additional WAMP serialization formats (currently MsgPack)

Install variants can be combined, e.g. to install Autobahn with all optional packages for use with Twisted on CPython:

```
pip install autobahn[twisted,accelerate,compress,serialization]
```

Windows Installation

For convenience, here are minimal instructions to install both Python and Autobahn/Twisted on Windows:

1. Go to the Python web site and install Python 2.7 32-Bit
2. Add `C:\Python27;C:\Python27\Scripts;` to your `PATH`
3. Download the Pip install script and double click it (or run `python get-pip.py` from a command shell)
4. Open a command shell and run `pip install autobahn[twisted]`

Check the Installation

To check the installation, fire up the Python and run

```
>>> from autobahn import __version__
>>> print(__version__)
0.9.1
```

Depending on Autobahn

To require **Autobahn** as a dependency of your package, include the following in your `setup.py` script

```
install_requires = ["autobahn>=0.9.1"]
```

You can also depend on an *install variant* which automatically installs dependent packages

```
install_requires = ["autobahn[twisted]>=0.9.1"]
```

The latter will automatically install Twisted as a dependency.

Where to go

Now you've got **Autobahn** installed, depending on your needs, head over to

- *Asynchronous Programming* - An very short introduction plus pointers to good Web resources.
- *WebSocket Programming* - A guide to programming WebSocket applications with Autobahn
- *WAMP Programming* - A guide to programming WAMP applications with Autobahn

CHAPTER 3

Asynchronous Programming

Introduction

The asynchronous programming approach

Autobahn is written according to a programming paradigm called *asynchronous programming* (or *event driven programming*) and implemented using *non-blocking* execution - and both go hand in hand.

A very good technical introduction to these concepts can be found in this chapter of an "Introduction to Asynchronous Programming and Twisted".

Here are two more presentations that introduce event-driven programming in Python

- Alex Martelli - Don't call us, we'll call you: callback patterns and idioms
- Glyph Lefkowitz - So Easy You Can Even Do It in JavaScript: Event-Driven Architecture for Regular Programmers

Another highly recommended reading is The Reactive Manifesto which describes guiding principles, motivations and connects the dots

> Non-blocking means the ability to make continuous progress in order to for the application to be responsive at all times, even under failure and burst scenarios. For this all resources needed for a response—for example CPU, memory and network—must not be monopolized. As such it can enable both lower latency, higher throughput and better scalability.
>
> —The Reactive Manifesto

The fact that **Autobahn** is implemented using asynchronous programming and non-blocking execution shouldn't come as a surprise, since both Twisted and asyncio - the foundations upon which Autobahn runs - are *asynchronous network programming frameworks*.

On the other hand, the principles of asynchronous programming are independent of Twisted and asyncio. For example, other frameworks that fall into the same category are:

- NodeJS

- Boost/ASIO
- Netty
- Tornado
- React

Tip: While getting accustomed to the asynchronous way of thinking takes some time and effort, the knowledge and experience acquired can be translated more or less directly to other frameworks in the asynchronous category.

Other forms of Concurrency

Asynchronous programming is not the only approach to concurrency. Other styles of concurrency include

1. OS Threads
2. Green Threads
3. Actors
4. Software Transactional Memory (STM)

Obviously, we cannot go into much detail with all of above. But here are some pointers for further reading if you want to compare and contrast asynchronous programming with other approaches.

With the **Actor model** a system is composed of a set of *actors* which are independently running, executing sequentially and communicate strictly by message passing. There is no shared state at all. This approach is used in systems like

- Erlang
- Akka
- Rust
- C++ Actor Framework

Software Transactional Memory (STM) applies the concept of Optimistic Concurrency Control from the persistent database world to (transient) program memory. Instead of lettings programs directly modify memory, all operations are first logged (inside a transaction), and then applied atomically - but only if no conflicting transaction has committed in the meantime. Hence, it's "optimistic" in that it assumes to be able to commit "normally", but needs to handle the failing at commit time.

Green Threads is using light-weight, run-time level threads and thread scheduling instead of OS threads. Other than that, systems are implemented similar: green threads still block, and still do share state. Python has multiple efforts in this category:

- Eventlet
- Gevent
- Stackless

Twisted or asyncio?

Since **Autobahn** runs on both Twisted and asyncio, which networking framework should you use?

Even more so, as the core of Twisted and asyncio is very similar and relies on the same concepts:

Twisted	asyncio	Description
Deferred	Future	abstraction of a value which isn't available yet
Reactor	Event Loop	waits for and dispatches events
Transport	Transport	abstraction of a communication channel (stream or datagram)
Protocol	Protocol	this is where actual networking protocols are implemented
Protocol Factory	Protocol Factory	responsible for creating protocol instances

In fact, I'd say the biggest difference between Twisted and asyncio is `Deferred` vs `Future`. Although similar on surface, their semantics are different. `Deferred` supports the concept of chainable callbacks (which can mutate the return values), and separate error-backs (which can cancel errors). `Future` has just a callback, that always gets a single argument: the Future.

Also, asyncio is opinionated towards co-routines. This means idiomatic user code for asyncio is expected to use co-routines, and not plain Futures (which are considered too low-level for application code).

But anyway, with asyncio being part of the language standard library (since Python 3.4), wouldn't you just *always* use asyncio? At least if you don't have a need to support already existing Twisted based code.

The truth is that while the *core* of Twisted and asyncio are very similar, **Twisted has a much broader scope: Twisted is "batteries included" for network programming.**

So you get *tons* of actual network protocols already out-of-the-box - in production quality implementations!

asyncio does not include any actual application layer network protocols like HTTP. If you need those, you'll have to look for asyncio implementations *outside* the standard library. For example, here is a HTTP server and client library for asyncio.

Over time, an ecosystem of protocols will likely emerge around asyncio also. But right now, Twisted has a big advantage here.

If you want to read more on this, Glyph (Twisted original author) has a nice blog post here.

Resources

Below we are listing a couple of resources on the Web for Twisted and asyncio that you may find useful.

Twisted Resources

We cannot give an introduction to asynchronous programming with Twisted here. And there is no need to, since there is lots of great stuff on the Web. In particular we'd like to recommend the following resources.

If you have limited time and nevertheless want to have an in-depth view of Twisted, Jessica McKellar has a great presentation recording with Architecting an event-driven networking engine: Twisted Python. That's 45 minutes. Highly recommended.

If you really want to get it, Dave Peticolas has written an awesome Introduction to Asynchronous Programming and Twisted. This is a detailed, hands-on tutorial with lots of code examples that will take some time to work through - but you actually *learn* how to program with Twisted.

Then of course there is

- The Twisted Documentation
- The Twisted API Reference

and lots and lots of awesome Twisted talks on PyVideo.

Asyncio Resources

asyncio is very new (August 2014). So the amount of material on the Web is still limited. Here are some resources you may find useful:

- Guido van Rossum's Keynote at PyCon US 2013
- Tulip: Async I/O for Python 3
- Python 3.4 docs - asyncio
- PEP-3156 - Asynchronous IO Support Rebooted
- OSB 2015 - How Do Python Coroutines Work? - A. Jesse Jiryu Davis

However, we quickly introduce core asynchronous programming primitives provided by Twisted and asyncio:

Asynchronous Programming Primitives

In this section, we have a quick look at some of the asynchronous programming primitive provided by Twisted and asyncio to show similarities and differences.

Twisted Deferreds and inlineCallbacks

Documentation pointers:

- Introduction to Deferreds
- Deferreds Reference
- Twisted inlineCallbacks

Programming with Twisted Deferreds involves attaching *callbacks* to Deferreds which get called when the Deferred finally either resolves successfully or fails with an error

```
d = some_function() # returns a Twisted Deferred ..

def on_success(res):
   print("result: {}".format(res))

def on_error(err):
   print("error: {}".format(err))

d.addCallbacks(on_success, on_error)
```

Using Deferreds offers the greatest flexibility since you are able to pass around Deferreds freely and can run code concurrently.

However, using plain Deferreds comes at a price: code in this style looks very different from synchronous/blocking code and the code can become hard to follow.

Now, Twisted inlineCallbacks let you write code in a sequential looking manner that nevertheless executes asynchronously and non-blocking under the hood.

So converting above snipped to `inlineCallbacks` the code will look like

```
try:
   res = yield some_function()
   print("result: {}".format(res))
```

```
except Exception as err:
   print("error: {}".format(err))
```

As you can see, this code looks very similar to regular synchronous/blocking Python code. The only difference (on surface) is the use of `yield` when calling a function that runs asynchronously. Otherwise, you process success result values and exceptions exactly as with regular code.

Note: We'll only show basic usage here - for a more basic and complete introduction, please have a look at this chapter from this tutorial.

Example

The following demonstrates basic usage of `inlineCallbacks` in a complete example you can run.

First, consider this program using Deferreds. We simulate calling a slow function by sleeping (without blocking) inside the function `slow_square`

```
1  from twisted.internet import reactor
2  from twisted.internet.defer import Deferred
3
4  def slow_square(x):
5     d = Deferred()
6
7     def resolve():
8        d.callback(x * x)
9
10    reactor.callLater(1, resolve)
11    return d
12
13 def test():
14    d = slow_square(3)
15
16    def on_success(res):
17       print(res)
18       reactor.stop()
19
20    d.addCallback(on_success)
21
22 test()
23 reactor.run()
```

This is just regular Twisted code - nothing exciting here:

1. We create a `Deferred` to be returned by our `slow_square` function (line 5)
2. We create a function `resolve` (a closure) in which we resolve the previously created Deferred with the result (lines 7-8)
3. Then we ask the Twisted reactor to call `resolve` after 1 second (line 10)
4. And we return the previously created Deferred to the caller (line 11)

What you can see even with this trivial example already is that the code looks quite differently from synchronous/blocking code. It needs some practice until such code becomes natural to read.

Now, when converted to `inlineCallbacks`, the code becomes:

```
1  from twisted.internet import reactor
2  from twisted.internet.defer import inlineCallbacks, returnValue
3  from autobahn.twisted.util import sleep
4
5  @inlineCallbacks
6  def slow_square(x):
7     yield sleep(1)
8     returnValue(x * x)
9
10 @inlineCallbacks
11 def test():
12    res = yield slow_square(3)
13    print(res)
14    reactor.stop()
15
16 test()
17 reactor.run()
```

Have a look at the highlighted lines - here is what we do:

1. Decorating our squaring function with `inlineCallbacks` (line 5). Doing so marks the function as a coroutine which allows us to use this sequential looking coding style.
2. Inside the function, we simulate the slow execution by sleeping for a second (line 7). However, we are sleeping in a non-blocking way (`autobahn.twisted.util.sleep()`). The `yield` will put the coroutine aside until the sleep returns.
3. To return values from Twisted coroutines, we need to use `returnValue` (line 8).

Note: The reason `returnValue` is necessary goes deep into implementation details of Twisted and Python. In short: co-routines in Python 2 with Twisted are simulated using exceptions. Only Python 3.3+ has gotten native support for co-routines using the new yield from statement, Python 3.5+ use await statement and it is the new recommended method.

In above, we are using a little helper `autobahn.twisted.util.sleep()` for sleeping "inline". The helper is really trivial:

```
from twisted.internet import reactor
from twisted.internet.defer import Deferred

def sleep(delay):
   d = Deferred()
   reactor.callLater(delay, d.callback, None)
   return d
```

The rest of the program is just for driving our test function and running a Twisted reactor.

Asyncio Futures and Coroutines

Asyncio Futures like Twisted Deferreds encapsulate the result of a future computation. At the time of creation, the result is (usually) not yet available, and will only be available eventually.

On the other hand, asyncio futures are quite different from Twisted Deferreds. One difference is that they have no built-in machinery for chaining.

Asyncio Coroutines are (on a certain level) quite similar to Twisted inline callbacks. Here is the code corresponding to our example above:

Example

The following demonstrates basic usage of `asyncio.coroutine` in a complete example you can run.

First, consider this program using plain `asyncio.Future`. We simulate calling a slow function by sleeping (without blocking) inside the function `slow_square`

```
1  import asyncio
2
3  def slow_square(x):
4     f = asyncio.Future()
5
6     def resolve():
7        f.set_result(x * x)
8
9     loop = asyncio.get_event_loop()
10    loop.call_later(1, resolve)
11
12    return f
13
14 def test():
15    f = slow_square(3)
16
17    def done(f):
18       res = f.result()
19       print(res)
20
21    f.add_done_callback(done)
22
23    return f
24
25 loop = asyncio.get_event_loop()
26 loop.run_until_complete(test())
27 loop.close()
```

Using asyncio in this way is probably quite unusual. This is because asyncio is opinionated towards using coroutines from the beginning. Anyway, here is what above code does:

1. We create a `Future` to be returned by our `slow_square` function (line 4)
2. We create a function `resolve` (a closure) in which we resolve the previously created Future with the result (lines 6-7)
3. Then we ask the asyncio event loop to call `resolve` after 1 second (line 10)
4. And we return the previously created Future to the caller (line 12)

What you can see even with this trivial example already is that the code looks quite differently from synchronous/blocking code. It needs some practice until such code becomes natural to read.

Now, when converted to `asyncio.coroutine`, the code becomes:

```
1  import asyncio
2
3  async def slow_square(x):
4     await asyncio.sleep(1)
5     return x * x
```

```
6
7
8  async def test():
9     res = await slow_square(3)
10    print(res)
11
12 loop = asyncio.get_event_loop()
13 loop.run_until_complete(test())
```

The main differences (on surface) are:

1. The declaration of the function with `async` keyword (line 3) in asyncio versus the decorator `@defer.inlineCallbacks` with Twisted
2. The use of `defer.returnValue` in Twisted for returning values whereas in asyncio, you can use plain returns (line 6)
3. The use of `await` in asyncio, versus `yield` in Twisted (line 5)
4. The auxiliary code to get the event loop started and stopped

Most of the examples that follow will show code for both Twisted and asyncio, unless the conversion is trivial.

CHAPTER 4

WebSocket Programming

This guide introduces WebSocket programming with **Autobahn**.

You'll see how to create WebSocket server ("*Creating Servers*") and client applications ("*Creating Clients*").

Resources:

- Example Code for this Guide: Twisted-based or asyncio-based
- More *WebSocket Examples*

Creating Servers

Using **Autobahn** you can create WebSocket servers that will be able to talk to any (compliant) WebSocket client, including browsers.

We'll cover how to define the behavior of your WebSocket server by writing *protocol classes* and show some boilerplate for actually running a WebSocket server using the behavior defined in the server protocol.

Server Protocols

To create a WebSocket server, you need to **write a protocol class to specify the behavior** of the server.

For example, here is a protocol class for a WebSocket echo server that will simply echo back any WebSocket message it receives:

```
class MyServerProtocol(WebSocketServerProtocol):

   def onMessage(self, payload, isBinary):
      ## echo back message verbatim
      self.sendMessage(payload, isBinary)
```

This is just three lines of code, but we will go through each one carefully, since writing protocol classes like above really is core to WebSocket programming using Autobahn.

The **first thing** to note is that you **derive** your protocol class from a base class provided by Autobahn. Depending on whether you write a Twisted or a asyncio based application, here are the base classes to derive from:

- *autobahn.twisted.websocket.WebSocketServerProtocol*
- *autobahn.asyncio.websocket.WebSocketServerProtocol*

So a Twisted-based echo protocol would import the base protocol from `autobahn.twisted.websocket` and derive from *autobahn.twisted.websocket.WebSocketServerProtocol*

Twisted:

```
from autobahn.twisted.websocket import WebSocketServerProtocol

class MyServerProtocol(WebSocketServerProtocol):

   def onMessage(self, payload, isBinary):
      ## echo back message verbatim
      self.sendMessage(payload, isBinary)
```

while an asyncio echo protocol would import the base protocol from `autobahn.asyncio.websocket` and derive from *autobahn.asyncio.websocket.WebSocketServerProtocol*

asyncio:

```
from autobahn.asyncio.websocket import WebSocketServerProtocol

class MyServerProtocol(WebSocketServerProtocol):

   def onMessage(self, payload, isBinary):
      ## echo back message verbatim
      self.sendMessage(payload, isBinary)
```

Note: In this example, only the imports differ between the Twisted and the asyncio variant. The rest of the code is identical. However, in most real world programs you probably won't be able to or don't want to avoid using network framework specific code.

Receiving Messages

The **second thing** to note is that we **override a callback** `onMessage` which is called by Autobahn whenever the callback related event happens.

In case of `onMessage`, the callback will be called whenever a new WebSocket message was received. There are more WebSocket related callbacks, but for now the `onMessage` callback is all we need.

When our server receives a WebSocket message, the *autobahn.websocket.interfaces.IWebSocketChannel.onMessage()* will fire with the message `payload` received.

The `payload` is always a Python byte string. Since WebSocket is able to transmit **text** (UTF8) and **binary** payload, the actual payload type is signaled via the `isBinary` flag.

When the `payload` is **text** (`isBinary == False`), the bytes received will be an UTF8 encoded string. To process **text** payloads, the first thing you often will do is decoding the UTF8 payload into a Python string:

```
s = payload.decode('utf8')
```

Tip: You don't need to validate the bytes for actually being valid UTF8 - Autobahn does that already when receiving the message.

When using WebSocket text messages with JSON `payload`, typical code for receiving and decoding messages into Python objects that works on both Python 2 and 3 would look like this:

```
import json
obj = json.loads(payload.decode('utf8'))
```

We are using the Python standard JSON module `json`.

The `payload` (which is of type `bytes` on Python 3 and `str` on Python 2) is decoded from UTF8 into a native Python string, and then parsed from JSON into a native Python object.

Sending Messages

The **third thing** to note is that we **use methods** like `sendMessage` provided by the base class to perform WebSocket related actions, like sending a WebSocket message.

As there are more methods for performing other actions (like closing the connection), we'll come back to this later, but for now, the `sendMessage` method is all we need.

`autobahn.websocket.interfaces.IWebSocketChannel.sendMessage()` takes the `payload` to send in a WebSocket message as Python bytes. Since WebSocket is able to transmit payloads of **text** (UTF8) and **binary** type, you need to tell Autobahn the actual type of the `payload` bytes. This is done using the `isBinary` flag.

Hence, to send a WebSocket text message, you will usually *encode* the payload to UTF8:

```
payload = s.encode('utf8')
self.sendMessage(payload, isBinary = False)
```

Warning: Autobahn will NOT validate the bytes of a text `payload` being sent for actually being valid UTF8. You MUST ensure that you only provide valid UTF8 when sending text messages. If you produce invalid UTF8, a conforming WebSocket peer will close the WebSocket connection due to the protocol violation.

When using WebSocket text messages with JSON `payload`, typical code for encoding and sending Python objects that works on both Python 2 and 3 would look like this:

```
import json
payload = json.dumps(obj, ensure_ascii = False).encode('utf8')
```

We are using the Python standard JSON module `json`.

The `ensure_ascii == False` option allows the JSON serializer to use Unicode strings. We can do this since we are encoding to UTF8 afterwards anyway. And UTF8 can represent the full Unicode character set.

Running a Server

Now that we have defined the behavior of our WebSocket server in a protocol class, we need to actually start a server based on that behavior.

Doing so involves two steps:

1. Create a **Factory** for producing instances of our protocol class
2. Create a TCP **listening server** using the former Factory

Here is one way of doing that when using Twisted

Twisted:

```
if __name__ == '__main__':

   import sys

   from twisted.python import log
   from twisted.internet import reactor
   log.startLogging(sys.stdout)

   from autobahn.twisted.websocket import WebSocketServerFactory
   factory = WebSocketServerFactory()
   factory.protocol = MyServerProtocol

   reactor.listenTCP(9000, factory)
   reactor.run()
```

What we are doing here is

1. Setup Twisted logging
2. Create a *autobahn.twisted.websocket.WebSocketServerFactory* and set our `MyServerProtocol` on the factory (the highlighted lines)
3. Start a server using the factory, listening on TCP port 9000

Similar, here is the asyncio way

asyncio:

```
if __name__ == '__main__':

   try:
      import asyncio
   except ImportError:
      ## Trollius >= 0.3 was renamed
      import trollius as asyncio

   from autobahn.asyncio.websocket import WebSocketServerFactory
   factory = WebSocketServerFactory()
   factory.protocol = MyServerProtocol

   loop = asyncio.get_event_loop()
   coro = loop.create_server(factory, '127.0.0.1', 9000)
   server = loop.run_until_complete(coro)

   try:
      loop.run_forever()
   except KeyboardInterrupt:
```

```
    pass
finally:
    server.close()
    loop.close()
```

What we are doing here is

1. Import asyncio, or the Trollius backport
2. Create a *autobahn.asyncio.websocket.WebSocketServerFactory* and set our `MyServerProtocol` on the factory (the highlighted lines)
3. Start a server using the factory, listening on TCP port 9000

Note: As can be seen, the boilerplate to create and run a server differ from Twisted, but the core code of creating a factory and setting our protocol (the highlighted lines) is identical (other than the differing import for the WebSocket factory).

You can find complete code for above examples here:

- WebSocket Echo (Twisted-based)
- WebSocket Echo (Asyncio-based)

Connection Lifecycle

As we have seen above, Autobahn will fire *callbacks* on your protocol class whenever the event related to the respective callback occurs.

It is in these callbacks that you will implement application specific code.

The core WebSocket interface *autobahn.websocket.interfaces.IWebSocketChannel* provides the following *callbacks*:

- *autobahn.websocket.interfaces.IWebSocketChannel.onConnect()*
- *autobahn.websocket.interfaces.IWebSocketChannel.onOpen()*
- *autobahn.websocket.interfaces.IWebSocketChannel.onMessage()*
- *autobahn.websocket.interfaces.IWebSocketChannel.onClose()*

We have already seen the callback for *Receiving Messages*. This callback will usually fire many times during the lifetime of a WebSocket connection.

In contrast, the other three callbacks above each only fires once for a given connection.

Opening Handshake

Whenever a new client connects to the server, a new protocol instance will be created and the *autobahn.websocket.interfaces.IWebSocketChannel.onConnect()* callback fires as soon as the WebSocket opening handshake is begun by the client.

For a WebSocket server protocol, `onConnect()` will fire with `autobahn.websocket.protocol.ConnectionRequest` providing information on the client wishing to connect via WebSocket.

```
class MyServerProtocol(WebSocketServerProtocol):

   def onConnect(self, request):
      print("Client connecting: {}".format(request.peer))
```

On the other hand, for a WebSocket client protocol, `onConnect()` will fire with `autobahn.websocket.protocol.ConnectionResponse` providing information on the WebSocket connection that was accepted by the server.

```
class MyClientProtocol(WebSocketClientProtocol):

   def onConnect(self, response):
      print("Connected to Server: {}".format(response.peer))
```

In this callback you can do things like

- checking or setting cookies or other HTTP headers
- verifying the client IP address
- checking the origin of the WebSocket request
- negotiate WebSocket subprotocols

For example, a WebSocket client might offer to speak several WebSocket subprotocols. The server can inspect the offered protocols in `onConnect()` via the supplied instance of `autobahn.websocket.protocol.ConnectionRequest`. When the server accepts the client, it'll chose one of the offered subprotocols. The client can then inspect the selected subprotocol in it's `onConnect()` callback in the supplied instance of `autobahn.websocket.protocol.ConnectionResponse`.

Connection Open

The *autobahn.websocket.interfaces.IWebSocketChannel.onOpen()* callback fires when the WebSocket opening handshake has been successfully completed. You now can send and receive messages over the connection.

```
class MyProtocol(WebSocketProtocol):

   def onOpen(self):
      print("WebSocket connection open.")
```

Closing a Connection

The core WebSocket interface *autobahn.websocket.interfaces.IWebSocketChannel* provides the following *methods*:

- *autobahn.websocket.interfaces.IWebSocketChannel.sendMessage()*
- *autobahn.websocket.interfaces.IWebSocketChannel.sendClose()*

We've already seen one of above in *Sending Messages*.

The *autobahn.websocket.interfaces.IWebSocketChannel.sendClose()* will initiate a WebSocket closing handshake. After starting to close a WebSocket connection, no messages can be sent. Eventually, the *autobahn.websocket.interfaces.IWebSocketChannel.onClose()* callback will fire.

After a WebSocket connection has been closed, the protocol instance will get recycled. Should the client reconnect, a new protocol instance will be created and a new WebSocket opening handshake performed.

Connection Close

When the WebSocket connection has closed, the *autobahn.websocket.interfaces.IWebSocketChannel.onClose()* callback fires.

```
class MyProtocol(WebSocketProtocol):

   def onClose(self, wasClean, code, reason):
      print("WebSocket connection closed: {}".format(reason))
```

When the connection has closed, no messages will be received anymore and you cannot send messages also. The protocol instance won't be reused. It'll be garbage collected. When the client reconnects, a completely new protocol instance will be created.

Creating Clients

Note: Creating WebSocket clients using **Autobahn** works very similar to creating WebSocket servers. Hence you should have read through *Creating Servers* first.

As with servers, the behavior of your WebSocket client is defined by writing a *protocol class*.

Client Protocols

To create a WebSocket client, you need to write a protocol class to **specify the behavior** of the client.

For example, here is a protocol class for a WebSocket client that will send a WebSocket text message as soon as it is connected and log any WebSocket messages it receives:

```
class MyClientProtocol(WebSocketClientProtocol):

   def onOpen(self):
      self.sendMessage(u"Hello, world!".encode('utf8'))

   def onMessage(self, payload, isBinary):
      if isBinary:
         print("Binary message received: {0} bytes".format(len(payload)))
      else:
         print("Text message received: {0}".format(payload.decode('utf8')))
```

Similar to WebSocket servers, you **derive** your WebSocket client protocol class from a base class provided by Autobahn. Depending on whether you write a Twisted or a asyncio based application, here are the base classes to derive from:

- *autobahn.twisted.websocket.WebSocketClientProtocol*
- *autobahn.asyncio.websocket.WebSocketClientProtocol*

So a Twisted-based protocol would import the base protocol from `autobahn.twisted.websocket` and derive from *autobahn.twisted.websocket.WebSocketClientProtocol*

Twisted:

```
from autobahn.twisted.websocket import WebSocketClientProtocol

class MyClientProtocol(WebSocketClientProtocol):

   def onOpen(self):
      self.sendMessage(u"Hello, world!".encode('utf8'))

   def onMessage(self, payload, isBinary):
      if isBinary:
         print("Binary message received: {0} bytes".format(len(payload)))
      else:
         print("Text message received: {0}".format(payload.decode('utf8')))
```

while an asyncio-based protocol would import the base protocol from `autobahn.asyncio.websocket` and derive from *autobahn.asyncio.websocket.WebSocketClientProtocol*

asyncio:

```
from autobahn.asyncio.websocket import WebSocketClientProtocol

class MyClientProtocol(WebSocketClientProtocol):

   def onOpen(self):
      self.sendMessage(u"Hello, world!".encode('utf8'))

   def onMessage(self, payload, isBinary):
      if isBinary:
         print("Binary message received: {0} bytes".format(len(payload)))
      else:
         print("Text message received: {0}".format(payload.decode('utf8')))
```

Note: In this example, only the imports differs between the Twisted and the asyncio variant. The rest of the code is identical. However, in most real world programs you probably won't be able to or don't want to avoid using network framework specific code.

Receiving and sending WebSocket messages as well as connection lifecycle in clients works exactly the same as with servers. Please see

- *Receiving Messages*
- *Sending Messages*
- *Connection Lifecycle*

Running a Client

Now that we have defined the behavior of our WebSocket client in a protocol class, we need to actually start a client based on that behavior.

Doing so involves two steps:

1. Create a **Factory** for producing instances of our protocol class
2. Create a TCP **connecting client** using the former Factory

Here is one way of doing that when using Twisted

Twisted:

```
if __name__ == '__main__':

   import sys

   from twisted.python import log
   from twisted.internet import reactor
   log.startLogging(sys.stdout)

   from autobahn.twisted.websocket import WebSocketClientFactory
   factory = WebSocketClientFactory()
   factory.protocol = MyClientProtocol

   reactor.connectTCP("127.0.0.1", 9000, factory)
   reactor.run()
```

What we are doing here is

1. Setup Twisted logging
2. Create a *autobahn.twisted.websocket.WebSocketClientFactory* and set our `MyClientProtocol` on the factory (the highlighted lines)
3. Start a client using the factory, connecting to localhost `127.0.0.1` on TCP port 9000

Similar, here is the asyncio way

asyncio:

```
if __name__ == '__main__':

   try:
      import asyncio
   except ImportError:
      ## Trollius >= 0.3 was renamed
      import trollius as asyncio

   from autobahn.asyncio.websocket import WebSocketClientFactory
   factory = WebSocketClientFactory()
   factory.protocol = MyClientProtocol

   loop = asyncio.get_event_loop()
   coro = loop.create_connection(factory, '127.0.0.1', 9000)
   loop.run_until_complete(coro)
   loop.run_forever()
   loop.close()
```

What we are doing here is

1. Import asyncio, or the Trollius backport
2. Create a *autobahn.asyncio.websocket.WebSocketClientFactory* and set our `MyClientProtocol` on the factory (the highlighted lines)
3. Start a client using the factory, connecting to localhost `127.0.0.1` on TCP port 9000

Note: As can be seen, the boilerplate to create and run a client differ from Twisted, but the core code of creating a factory and setting our protocol (the highlighted lines) is identical (other than the differing import for the WebSocket

factory).

You can find complete code for above examples here:

- WebSocket Echo (Twisted-based)
- WebSocket Echo (Asyncio-based)

WebSocket Options

You can pass various options on both client and server side WebSockets; these are accomplished by calling `autobahn.websocket.WebSocketServerFactory.setProtocolOptions()` or `autobahn.websocket.WebSocketClientFactory.setProtocolOptions()` with keyword arguments for each option.

Common Options (server and client)

- logOctets: if True, log every byte
- logFrames: if True, log information about each frame
- trackTimings: if True, enable debug timing code
- utf8validateIncoming: if True (default), validate all incoming UTF8
- applyMask: if True (default) apply mask to frames, when available
- maxFramePayloadSize: if 0 (default), unlimited-sized frames allowed
- maxMessagePayloadSize: if 0 (default), unlimited re-assembled payloads
- autoFragmentSize: if 0 (default), don't fragment
- failByDrop: if True (default), failed connections are terminated immediately
- echoCloseCodeReason: if True, echo back the close reason/code
- openHandshakeTimeout: timeout in seconds after which opening handshake will be failed (default: no timeout)
- closeHandshakeTimeout: timeout in seconds after which close handshake will be failed (default: no timeout)
- tcpNoDelay: if True (default), set NODELAY (Nagle) socket option
- autoPingInterval: if set, seconds between auto-pings
- autoPingTimeout: if set, seconds until a ping is considered timed-out
- autoPingSize: bytes of random data to send in ping messages (between 4 [default] and 125)

Server-Only Options

- versions: what versions to claim support for (default 8, 13)
- webStatus: if True (default), show a web page if visiting this endpoint without an Upgrade header
- requireMaskedClientFrames: if True (default), client-to-server frames must be masked
- maskServerFrames: if True, server-to-client frames must be masked
- perMessageCompressionAccept: if provided, a single-argument callable

- serveFlashSocketPolicy: if True, server a flash policy file (default: False)
- flashSocketPolicy: the actual flash policy to serve (default one allows everything)
- allowedOrigins: a list of origins to allow, with embedded *'s for wildcards; these are turned into regular expressions (e.g. *https://*.example.com:443* becomes *^https://.*.example.com:443$*). When doing the matching, the origin is **always** of the form *scheme://host:port* with an explicit port. By default, we match with * (that is, anything). To match all subdomains of *example.com* on any scheme and port, you'd need **://*.example.com:**
- maxConnections: total concurrent connections allowed (default 0, unlimited)
- trustXForwardedFor: number of trusted web servers (reverse proxies) in front of this server which set the X-Forwarded-For header

Client-Only Options

- version: which version we are (default: 18)
- acceptMaskedServerFrames: if True, accept masked server-to-client frames (default False)
- maskClientFrames: if True (default), mask client-to-server frames
- serverConnectionDropTimeout: how long (in seconds) to wait for server to drop the connection when closing (default 1)
- perMessageCompressionOffers:
- perMessageCompressionAccept:

Upgrading

From < 0.7.0

Starting with release 0.7.0, **Autobahn** now supports both Twisted and asyncio as the underlying network library. This required renaming some modules.

Hence, code for Autobahn **< 0.7.0**

```
from autobahn.websocket import WebSocketServerProtocol
```

should be modified for Autobahn **>= 0.7.0** for (using Twisted)

```
from autobahn.twisted.websocket import WebSocketServerProtocol
```

or (using asyncio)

```
from autobahn.asyncio.websocket import WebSocketServerProtocol
```

Two more small changes:

1. The method `WebSocketProtocol.sendMessage` had parameter `binary` renamed to `isBinary` (for consistency with `onMessage`)
2. The `ConnectionRequest` object no longer provides `peerstr`, but only `peer`, and the latter is a plain, descriptive string (this was needed since we now support both Twisted and asyncio, and also non-TCP transports)

CHAPTER 5

WAMP Programming

This guide gives an introduction to programming with WAMP in Python using **Autobahn**. (Go straight to *WAMP Examples*)

WAMP provides two communication patterns for application components to talk to each other

- *Remote Procedure Calls*
- *Publish & Subscribe*

and we will cover all four interactions involved in above patterns

1. *Registering Procedures* for remote calling
2. *Calling Procedures* remotely
3. *Subscribing to Topics* for receiving events
4. *Publishing Events* to topics

Note that WAMP is a "routed" protocol, and defines a Dealer and Broker role. Practically speaking, this means that any WAMP client needs a WAMP Router to talk to. We provide an open-source one called Crossbar (there are other routers available). See also the WAMP specification for more details

Tip: If you are new to WAMP or want to learn more about the design principles behind WAMP, we have a longer text here.

Application Components

WAMP is all about creating systems from loosely coupled *application components*. These application components are where your application-specific code runs.

A WAMP-based system consists of potentially many application components, which all connect to a WAMP router. The router is *generic*, which means, it does *not* run any application code, but only provides routing of events and calls.

These components use either Remote Procedure Calls (RPC) or Publish/Subscribe (PubSub) to communicate. Each component can do any mix of: register, call, subscribe or publish.

For RPC, an application component registers a callable method at a URI ("endpoint"), and other components call it via that endpoint.

In the Publish/Subscribe model, interested components subscribe to an event URI and when a publish to that URI happens, the event payload is routerd to all subscribers:

Hence, to create a WAMP application, you:

1. write application components
2. connect the components to a router

Note that each component can do any mix of registering, calling, subscribing and publishing – it is entirely up to you to logically group functionality as suits your problem space.

Creating Components

You create an application component by deriving from a base class provided by Autobahn.

When using **Twisted**, you derive from *autobahn.twisted.wamp.ApplicationSession*

```
from autobahn.twisted.wamp import ApplicationSession

class MyComponent(ApplicationSession):
   def onJoin(self, details):
      print("session ready")
```

whereas when you are using **asyncio**, you derive from *autobahn.asyncio.wamp.ApplicationSession*

```
from autobahn.asyncio.wamp import ApplicationSession

class MyComponent(ApplicationSession):
   def onJoin(self, details):
      print("session ready")
```

As can be seen, the only difference between Twisted and asyncio is the import (line 1). The rest of the code is identical.

Also, Autobahn will invoke callbacks on your application component when certain events happen. For example, *ISession.onJoin* is triggered when the WAMP session has connected to a router and joined a realm. We'll come back to this topic later.

Running Components

To actually make use of an application components, the component needs to connect to a WAMP router. **Autobahn** includes a *runner* that does the heavy lifting for you.

Here is how you use *autobahn.twisted.wamp.ApplicationRunner* with **Twisted**

```
from autobahn.twisted.wamp import ApplicationRunner

runner = ApplicationRunner(url=u"ws://localhost:8080/ws", realm=u"realm1")
runner.run(MyComponent)
```

and here is how you use *autobahn.asyncio.wamp.ApplicationRunner* with **asyncio**

```
from autobahn.asyncio.wamp import ApplicationRunner

runner = ApplicationRunner(url=u"ws://localhost:8080/ws", realm=u"realm1")
runner.run(MyComponent)
```

As can be seen, the only difference between Twisted and asyncio is the import (line 1). The rest of the code is identical.

There are two mandatory arguments to `ApplicationRunner`:

1. `url`: the WebSocket URL of the WAMP router (for WAMP-over-WebSocket)
2. `realm`: the *Realm* the component should join on that router

Tip: A *Realm* is a routing namespace and an administrative domain for WAMP. For example, a single WAMP router can manage multiple *Realms*, and those realms are completely separate: an event published to topic T on a Realm R1 is NOT received by a subscribe to T on Realm R2.

Here are quick templates for you to copy/paste for creating and running a WAMP component.

Twisted:

```
from twisted.internet.defer import inlineCallbacks
from autobahn.twisted.wamp import ApplicationSession, ApplicationRunner

class MyComponent(ApplicationSession):

    @inlineCallbacks
    def onJoin(self, details):
        print("session joined")
        # can do subscribes, registers here e.g.:
        # yield self.subscribe(...)
        # yield self.register(...)

if __name__ == '__main__':
    runner = ApplicationRunner(url=u"ws://localhost:8080/ws", realm=u"realm1")
    runner.run(MyComponent)
```

asyncio:

```
from autobahn.asyncio.wamp import ApplicationSession, ApplicationRunner

class MyComponent(ApplicationSession):
    async def onJoin(self, details):
        print("session joined")
        # can do subscribes, registers here e.g.:
        # await self.subscribe(...)
        # await self.register(...)

if __name__ == '__main__':
    runner = ApplicationRunner(url=u"ws://localhost:8080/ws", realm=u"realm1")
    runner.run(MyComponent)
```

Running a WAMP Router

The component we've created attempts to connect to a **WAMP router** running locally which accepts connections on port `8080`, and for a realm `realm1`.

Our suggested way is to use Crossbar.io as your WAMP router. There are other WAMP routers besides Crossbar.io as well.

Once you've installed Crossbar.io, initialize an instance of it with the default settings, which will accept WAMP (over WebSocket) connections on `ws://<hostname>:8080/ws` and has a `realm1` pre-configured.

To do this, do

```
crossbar init
```

This will create the default Crossbar.io node configuration `./.crossbar/config.json`. You can then start Crossbar.io by doing:

```
crossbar start
```

Remote Procedure Calls

Remote Procedure Call (RPC) is a messaging pattern involving peers of three roles:

- *Caller*
- *Callee*
- *Dealer*

A *Caller* issues calls to remote procedures by providing the procedure URI and any arguments for the call. The *Callee* will execute the procedure using the supplied arguments to the call and return the result of the call to the Caller.

Callees register procedures they provide with *Dealers*. *Callers* initiate procedure calls first to *Dealers*. *Dealers* route calls incoming from *Callers* to *Callees* implementing the procedure called, and route call results back from *Callees* to *Callers*.

The *Caller* and *Callee* will usually run application code, while the *Dealer* works as a generic router for remote procedure calls decoupling *Callers* and *Callees*. Thus, the *Caller* can be in a separate process (even a separate implementation language) from the *Callee*.

Registering Procedures

To make a procedure available for remote calling, the procedure needs to be *registered*. Registering a procedure is done by calling `ICallee.register` from a session.

Here is an example using **Twisted**

```
1 from autobahn.twisted.wamp import ApplicationSession
2 from twisted.internet.defer import inlineCallbacks
3
4
5 class MyComponent(ApplicationSession):
6     @inlineCallbacks
7     def onJoin(self, details):
8         print("session ready")
```

```
9
10          def add2(x, y):
11              return x + y
12
13          try:
14              yield self.register(add2, u'com.myapp.add2')
15              print("procedure registered")
16          except Exception as e:
17              print("could not register procedure: {0}".format(e))
```

The procedure `add2` is registered (line 14) under the URI `u"com.myapp.add2"` immediately in the `onJoin` callback which fires when the session has connected to a *Router* and joined a *Realm*.

Tip: You can register *local* functions like in above example, *global* functions as well as *methods* on class instances. Further, procedures can also be automatically registered using *decorators*.

When the registration succeeds, authorized callers will immediately be able to call the procedure (see *Calling Procedures*) using the URI under which it was registered (`u"com.myapp.add2"`).

A registration may also fail, e.g. when a procedure is already registered under the given URI or when the session is not authorized to register procedures.

Using **asyncio**, the example looks like this:

```
1  from autobahn.asyncio.wamp import ApplicationSession
2
3  class MyComponent(ApplicationSession):
4      async def onJoin(self, details):
5          print("session ready")
6
7          def add2(x, y):
8              return x + y
9
10         try:
11             await self.register(add2, u'com.myapp.add2')
12             print("procedure registered")
13         except Exception as e:
14             print("could not register procedure: {0}".format(e))
```

The differences compared with the Twisted variant are:

- the `import` of `ApplicationSession`
- the use of `async` keyword to declare co-routines
- the use of `await` instead of `yield`

Calling Procedures

Calling a procedure (that has been previously registered) is done using `autobahn.wamp.interfaces.ICaller.call()`.

Here is how you would call the procedure `add2` that we registered in *Registering Procedures* under URI `com.myapp.add2` in **Twisted**

```
from autobahn.twisted.wamp import ApplicationSession
from twisted.internet.defer import inlineCallbacks


class MyComponent(ApplicationSession):
    @inlineCallbacks
    def onJoin(self, details):
        print("session ready")

        try:
            res = yield self.call(u'com.myapp.add2', 2, 3)
            print("call result: {}".format(res))
        except Exception as e:
            print("call error: {0}".format(e))
```

And here is the same done on **asyncio**

```
from autobahn.asyncio.wamp import ApplicationSession


class MyComponent(ApplicationSession):
    async def onJoin(self, details):
        print("session ready")

        try:
            res = await self.call(u'com.myapp.add2', 2, 3)
            print("call result: {}".format(res))
        except Exception as e:
            print("call error: {0}".format(e))
```

Publish & Subscribe

Publish & Subscribe (PubSub) is a messaging pattern involving peers of three roles:

- *Publisher*
- *Subscriber*
- *Broker*

A *Publisher* publishes events to topics by providing the topic URI and any payload for the event. Subscribers of the topic will receive the event together with the event payload.

Subscribers subscribe to topics they are interested in with *Brokers*. *Publishers* initiate publication first at a *Broker*. *Brokers* route events incoming from *Publishers* to *Subscribers* that are subscribed to respective topics.

The *Publisher* and *Subscriber* will usually run application code, while the *Broker* works as a generic router for events thus decoupling *Publishers* from *Subscribers*. That is, there can be many *Subscribers* written in different languages on different machines which can all receive a single event published by an independant *Publisher*.

Subscribing to Topics

To receive events published to a topic, a session needs to first subscribe to the topic. Subscribing to a topic is done by calling `autobahn.wamp.interfaces.ISubscriber.subscribe()`.

Here is a **Twisted** example:

```
from autobahn.twisted.wamp import ApplicationSession
from twisted.internet.defer import inlineCallbacks


class MyComponent(ApplicationSession):
    @inlineCallbacks
    def onJoin(self, details):
        print("session ready")

        def oncounter(count):
            print("event received: {0}", count)

        try:
            yield self.subscribe(oncounter, u'com.myapp.oncounter')
            print("subscribed to topic")
        except Exception as e:
            print("could not subscribe to topic: {0}".format(e))
```

We create an event handler function `oncounter` (you can name that as you like) which will get called whenever an event for the topic is received.

To subscribe (line 15), we provide the event handler function (`oncounter`) and the URI of the topic to which we want to subscribe (`u'com.myapp.oncounter'`).

When the subscription succeeds, we will receive any events published to `u'com.myapp.oncounter'`. Note that we won't receive events published *before* the subscription succeeds.

The corresponding **asyncio** code looks like this

```
from autobahn.asyncio.wamp import ApplicationSession


class MyComponent(ApplicationSession):
   async def onJoin(self, details):
       print("session ready")

       def oncounter(count):
           print("event received: {0}", count)

       try:
           await self.subscribe(oncounter, u'com.myapp.oncounter')
           print("subscribed to topic")
       except Exception as e:
           print("could not subscribe to topic: {0}".format(e))
```

Again, nearly identical to Twisted.

Publishing Events

Publishing an event to a topic is done by calling `autobahn.wamp.interfaces.IPublisher.publish()`.

Events can carry arbitrary positional and keyword based payload – as long as the payload is serializable in JSON.

Here is a **Twisted** example that will publish an event to topic `u'com.myapp.oncounter'` with a single (positional) payload being a counter that is incremented for each publish:

```
from autobahn.twisted.wamp import ApplicationSession
from autobahn.twisted.util import sleep
```

```
from twisted.internet.defer import inlineCallbacks


class MyComponent(ApplicationSession):
    @inlineCallbacks
    def onJoin(self, details):
        print("session ready")

        counter = 0
        while True:
            self.publish(u'com.myapp.oncounter', counter)
            counter += 1
            yield sleep(1)
```

The corresponding **asyncio** code looks like this

```
from autobahn.asyncio.wamp import ApplicationSession
from asyncio import sleep


class MyComponent(ApplicationSession):
    async def onJoin(self, details):
        print("session ready")

        counter = 0
        while True:
            self.publish(u'com.myapp.oncounter', counter)
            counter += 1
            await sleep(1)
```

When publishing, you can pass an *options*= kwarg which is an instance of *PublishOptions*. Many of the options require support from the router.

- whitelisting and blacklisting (all the *eligible** and *exclude** options) can affect which subscribers receive the publish; see crossbar documentation for more information;
- *retain*= asks the router to retain the message;
- *acknowledge*= asks the router to notify you it received the publish (note that this does *not* wait for every subscriber to have received the publish).

Tip: By default, a publisher will not receive an event it publishes even when the publisher is *itself* subscribed to the topic subscribed to. This behavior can be overridden; see *PublishOptions* and `exclude_me=False`.

Tip: By default, publications are *unacknowledged*. This means, a `publish()` may fail *silently* (like when the session is not authorized to publish to the given topic). This behavior can be overridden; see *PublishOptions* and `acknowledge=True`.

Session Lifecycle

A WAMP application component has this lifecycle:

1. component created

2. transport connected (*ISession.onConnect* called)
3. authentication challenge received (only for authenticated WAMP sessions, *ISession.onChallenge* called)
4. session established (realm joined, *ISession.onJoin* called)
5. session closed (realm left, *ISession.onLeave* called)
6. transport disconnected (*ISession.onDisconnect* called)

The *ApplicationSession* will fire the following events which you can handle by overriding the respective method (see *ISession* for more information):

```
class MyComponent(ApplicationSession):
    def __init__(self, config=None):
        ApplicationSession.__init__(self, config)
        print("component created")

    def onConnect(self):
        print("transport connected")
        self.join(self.config.realm)

    def onChallenge(self, challenge):
        print("authentication challenge received")

    def onJoin(self, details):
        print("session joined")

    def onLeave(self, details):
        print("session left")

    def onDisconnect(self):
        print("transport disconnected")
```

Logging

Internally, **Autobahn** uses txaio as an abstraction layer over Twisted and asyncio APIs. txaio also provides an abstracted logging API, which is what both **Autobahn** and Crossbar use.

There is a txaio Programming Guide which includes information on logging. If you are writing new code, you can choose the txaio APIs for maximum compatibility and runtime-efficiency (see below). If you prefer to write idiomatic logging code to "go with" the event-based frameword you've chosen, that's possible as well. For asyncio this is Python's built-in logging module; for Twisted it is the post-15.2.0 logging API. The logging system in txaio is able to interoperate with the legacy Twisted logging API as well.

The txaio API encourages a more structured approach while still achieving easily-rendered text logging messages. The basic idiom is to use new-style Python formatting strings and pass any "data" as kwargs. So a typical logging call might look like: `self.log.info("Knob {frob.name} moved {degrees} right.", knob=an_obj, degrees=42)` and if the "info" log level is not enabled, the string won't be "interpolated" (i.e. `str()` will not be invoked on any of the args, and a new string won't be produced). On top of that, logging observers may examine the `kwargs` and do things beyond "normal" logging. This is very much inspired by `twisted.logger`; you can read the Twisted logging documentation for more insight.

Before any logging happens of course you must activate the logging system. There is a convenience method in txaio called `txaio.start_logging`. This will use `twisted.logger.globalLogBeginner` on Twisted or `logging.Logger.addHandler` under asyncio and allows you to specify and output stream and/or a log level. Valid levels are the list of strings in `txaio.interfaces.log_levels`.

If you have instead got your own log-starting code (e.g. `twistd`) or Twisted/asyncio specific log handlers (`logging.Handler` subclass on asyncio and `ILogObserver` implementer under Twisted) then you will still get **Autobahn** and Crossbar messages. Probably the formatting will be slightly different from what `txaio.start_logging` provides. In either case, **do not depend on the formatting** of the messages e.g. by "screen-scraping" the logs.

We very much **recommend using the ‘‘txaio.start_logging()‘‘ method** of activating the logging system, as we've gone to pains to ensure that over-level logs are a "no-op" and incur minimal runtime cost. We achieve this by re-binding all out-of-scope methods on any logger created by `txaio.make_logger()` to a do-nothing function (by saving weak-refs of all the loggers created); at least on PyPy this is very well optimized out. This allows us to be generous with `.debug()` or `.trace()` calls without incurring very much overhead. Your Milage May Vary using other methods. If you haven't called `txaio.start_logging()` this optimization is not activated.

Upgrading

From < 0.8.0

Starting with release 0.8.0, **Autobahn** now supports WAMP v2, and also support both Twisted and asyncio. This required changing module naming for WAMP v1 (which is Twisted only).

Hence, WAMP v1 code for Autobahn **< 0.8.0**

```
from autobahn.wamp import WampServerFactory
```

should be modified for Autobahn **>= 0.8.0** for (using Twisted)

```
from autobahn.wamp1.protocol import WampServerFactory
```

> **Warning:** WAMP v1 will be deprecated with the 0.9 release of **Autobahn** which is expected in Q4 2014.

From < 0.9.4

Starting with release 0.9.4, all WAMP router code in **Autobahn** has been split out and moved to Crossbar.io. Please see the announcement here.

CHAPTER 6

WebSocket Examples

Basic Examples

Note: The examples here demonstrate WebSocket programming with Autobahn and are available in Twisted and asyncio-based variants respectively.

Echo

Twisted / asyncio

A simple WebSocket echo server and client.

Slow Square

Twisted / asyncio

This example shows a WebSocket server that will receive a JSON encode float over WebSocket, slowly compute the square, and send back the result. The example is intended to demonstrate how to use co-routines inside WebSocket handlers.

Testee

Twisted / asyncio

The example implements a *testee* for testing against Autobahn|Testsuite.

Additional Examples

Note: The examples here demonstrate various further features and aspects of WebSocket programming with Autobahn. However, these examples are **currently only available for Twisted**.

Secure WebSocket

Twisted

How to run WebSocket over TLS ("wss").

WebSocket and Twisted Web

Twisted

How to run WebSocket under Twisted Web. This is a very powerful feature, as it allows you to create a complete HTTP(S) resource hierarchy with different services like static file serving, REST and WebSocket combined under one server.

Twisted Web, WebSocket and WSGI

Twisted

This example shows how to run Flask (or any other WSGI compliant Web thing) under Twisted Web and combine that with WebSocket.

Secure WebSocket and Twisted Web

Twisted

A variant of the previous example that runs a HTTPS server with secure WebSocket on a subpath.

WebSocket Ping-Pong

Twisted

The example demonstrates how to trigger and process WebSocket pings and pongs.

More

- WebSocket Authentication with Mozilla Persona
- Broadcasting over WebSocket
- WebSocket Compression
- WebSocket over Twisted Endpoints
- Using HTTP Headers with WebSocket
- WebSocket on Multicore

- WebSocket as a Twisted Service
- WebSocket Echo Variants
- WebSocket Fallbacks
- Using multiple WebSocket Protocols
- Streaming WebSocket
- Wrapping Twisted Protocol/Factories over WebSocket
- Using wxPython with Autobahn

CHAPTER 7

WAMP Examples

NOTE that for all examples you will **need to run a router**. We develop Crossbar.io and there are other routers available as well. We include a working Crossbar.io configuration in the examples/router/ subdirectory as well as instructions on how to run it.

Overview of Examples

The examples are organized between asyncio and Twisted at the top-level, with similarly-named examples demonstrating the same functionality with the respective framework.

Each example typically includes four things:

- `frontend.py`: the Caller or Subscriber, in Python
- `backend.py`: the Callee or Publisher, in Python
- `frontend.js`: JavaScript version of the frontend
- `backend.js`: JavaScript version of the backend
- `*.html`: boilerplate so a browser can run the JavaScript

So for each example, you start *one* backend and *one* frontend component (your choice). You can usually start multiple frontend components with no problem, but will get errors if you start two backends trying to register at the same procedure URI (for example).

Still, you are encouraged to try playing with mixing and matching the frontend and backend components, starting multiple front-ends, etc. to explore Crossbar and Autobahn's behavior. Often the different examples use similar URIs for procedures and published events, so you can even try mixing between the examples.

The provided Crossbar.io configuration will run a Web server that you can visit at *http://localhost:8080* and includes links to the frontend/backend HTML for the javascript versions. Usually these just use `console.log()` so you'll have to open up the JavaScript console in your browser to see it working.

Automatically Run All Examples

There is a script (`./examples/run-all-examples.py`) which runs all the WAMP examples for 5 seconds each, this asciicast shows you how (see comments for how to run it yourself):

Publish & Subscribe (PubSub)

- Basic Twisted - asyncio - Demonstrates basic publish and subscribe.
- Complex Twisted - asyncio - Demonstrates publish and subscribe with complex events.
- Options Twisted - asyncio - Using options with PubSub.
- Unsubscribe Twisted - asyncio - Cancel a subscription to a topic.

Remote Procedure Calls (RPC)

- Time Service Twisted - asyncio - A trivial time service - demonstrates basic remote procedure feature.
- Slow Square Twisted - asyncio - Demonstrates procedures which return promises and return asynchronously.
- Arguments Twisted - asyncio - Demonstrates all variants of call arguments.
- Complex Result Twisted - asyncio - Demonstrates complex call results (call results with more than one positional or keyword results).
- Errors Twisted - asyncio - Demonstrates error raising and catching over remote procedures.
- Progressive Results Twisted - asyncio - Demonstrates calling remote procedures that produce progressive results.
- Options Twisted - asyncio - Using options with RPC.

I'm Confused, Just Tell Me What To Run

If all that is too many options to consider, you want to do this:

1. Open 3 terminals
2. In terminal 1, setup and run a local Crossbar in the root of your Autobahn checkout.
3. In terminals 2 and 3, go to the root of your Autobahn checkout and activate the virtualenv from step 2 (`source venv-autobahn/bin/activate`)
4. In terminal 2 run `python ./examples/twisted/wamp/rpc/arguments/backend.py`
5. In terminal 3 run `python ./examples/twisted/wamp/rpc/arguments/frontend.py`

The above procedure is gone over in this this asciicast:

CHAPTER 8

Public API Reference

The following is a API reference of **Autobahn** generated from Python source code and docstrings.

Warning: This is a *complete* reference of the *public* API of **Autobahn**. User code and applications should only rely on the public API, since internal APIs can (and will) change without any guarantees. Anything *not* listed here is considered a private API.

Module `autobahn.util`

`autobahn.util.`**`encode_truncate`**(*text*, *limit*, *encoding='utf8'*, *return_encoded=True*)

Given a string, return a truncated version of the string such that the UTF8 encoding of the string is smaller than the given limit.

This function correctly truncates even in the presence of Unicode code points that encode to multi-byte encodings which must not be truncated in the middle.

Parameters

- **`text`** (*`str`*) – The (Unicode) string to truncate.
- **`limit`** (*`int`*) – The number of bytes to limit the UTF8 encoding to.
- **`encoding`** (*`str`*) – Truncate the string in this encoding (default is `utf-8`).
- **`return_encoded`** (*`bool`*) – If `True`, return the string encoded into bytes according to the specified encoding, else return the string as a string.

Returns The truncated string.

Return type str or bytes

`autobahn.util.`**`xor`**(*d1*, *d2*)

XOR two binary strings of arbitrary (equal) length.

Parameters

- **d1** (*binary*) – The first binary string.
- **d2** (*binary*) – The second binary string.

Returns XOR of the binary strings (`XOR(d1, d2)`)

Return type bytes

`autobahn.util.`**`utcstr`**(*ts=None*)

Format UTC timestamp in ISO 8601 format.

Note: to parse an ISO 8601 formatted string, use the **iso8601** module instead (e.g. `iso8601.parse_date("2014-05-23T13:03:44.123Z")`).

Parameters **ts** (instance of `datetime.datetime` or `None`) – The timestamp to format.

Returns Timestamp formatted in ISO 8601 format.

Return type str

`autobahn.util.`**`utcnow`**()

Get current time in UTC as ISO 8601 string.

Returns Current time as string in ISO 8601 format.

Return type str

`autobahn.util.`**`generate_token`**(*char_groups*, *chars_per_group*, *chars=None*, *sep=None*, *lower_case=False*)

Generate cryptographically strong tokens, which are strings like *M6X5-YO5W-T5IK*. These can be used e.g. for used-only-once activation tokens or the like.

The returned token has an entropy of `math.log(len(chars), 2.) * chars_per_group * char_groups` bits.

With the default charset and 4 characters per group, `generate_token()` produces strings with the following entropy:

character groups	entropy (at least)	recommended use
2	38 bits	
3	57 bits	one-time activation or pairing code
4	76 bits	secure user password
5	95 bits	
6	114 bits	globally unique serial / product code
7	133 bits	

Here are some examples:

•token(3): `9QXT-UXJW-7R4H`

•token(4): `LPNN-JMET-KWEP-YK45`

•token(6): `NXW9-74LU-6NUH-VLPV-X6AG-QUE3`

Parameters

- **char_groups** (*int*) – Number of character groups (or characters if chars_per_group == 1).
- **chars_per_group** (*int*) – Number of characters per character group (or 1 to return a token with no grouping).
- **chars** (*str or None*) – Characters to choose from. Default is 27 character subset of the ISO basic Latin alphabet (see: `DEFAULT_TOKEN_CHARS`).

- **sep** (*str*) – When separating groups in the token, the separater string.
- **lower_case** (*bool*) – If `True`, generate token in lower-case.

Returns The generated token.

Return type str

`autobahn.util.`**generate_activation_code**`()`
Generate a one-time activation code or token of the form `u'W97F-96MJ-YGJL'`. The generated value is cryptographically strong and has (at least) 57 bits of entropy.

Returns The generated activation code.

Return type str

`autobahn.util.`**generate_user_password**`()`
Generate a secure, random user password of the form `u'kgojzi61dn5dtb6d'`. The generated value is cryptographically strong and has (at least) 76 bits of entropy.

Returns The generated password.

Return type str

`autobahn.util.`**generate_serial_number**`()`
Generate a globally unique serial / product code of the form `u'YRAC-EL4X-FQQE-AW4T-WNUV-VN6T'`. The generated value is cryptographically strong and has (at least) 114 bits of entropy.

Returns The generated serial number / product code.

Return type str

`autobahn.util.`**rtime**`()`
Precise, fast wallclock time.

Returns The current wallclock in seconds. Returned values are only guaranteed to be meaningful relative to each other.

Return type float

Module `autobahn.websocket`

WebSocket Interfaces

class `autobahn.websocket.interfaces.`**IWebSocketChannel**
A WebSocket channel is a bidirectional, full-duplex, ordered, reliable message channel over a WebSocket connection as specified in RFC6455.

This interface defines a message-based API to WebSocket plus auxiliary hooks and methods.

onConnect(*request_or_response*)
Callback fired during WebSocket opening handshake when a client connects (to a server with request from client) or when server connection established (by a client with response from server). This method may run asynchronous code.

Parameters **request_or_response** (Instance of *autobahn.websocket.types.ConnectionRequest* or *autobahn.websocket.types.ConnectionResponse*.) – Connection request (for servers) or response (for clients).

Returns When this callback is fired on a WebSocket server, you may return either `None` (in which case the connection is accepted with no specific WebSocket subprotocol) or an instance of *autobahn.websocket.types.ConnectionAccept*. When the callback is fired on a WebSocket client, this method must return `None`. Do deny a connection, raise an Exception. You can also return a Deferred/Future that resolves/rejects to the above.

onOpen()
Callback fired when the initial WebSocket opening handshake was completed. You now can send and receive WebSocket messages.

sendMessage(*payload*, *isBinary*)
Send a WebSocket message over the connection to the peer.

Parameters

- **payload** (*bytes*) – The WebSocket message to be sent.
- **isBinary** (*bool*) – Flag indicating whether payload is binary or UTF-8 encoded text.

onMessage(*payload*, *isBinary*)
Callback fired when a complete WebSocket message was received.

Parameters

- **payload** (*bytes*) – The WebSocket message received.
- **isBinary** (*bool*) – Flag indicating whether payload is binary or UTF-8 encoded text.

sendClose(*code=None*, *reason=None*)
Starts a WebSocket closing handshake tearing down the WebSocket connection.

Parameters

- **code** (*int*) – An optional close status code (`1000` for normal close or `3000-4999` for application specific close).
- **reason** (*str*) – An optional close reason (a string that when present, a status code MUST also be present).

onClose(*wasClean*, *code*, *reason*)
Callback fired when the WebSocket connection has been closed (WebSocket closing handshake has been finished or the connection was closed uncleanly).

Parameters

- **wasClean** (*bool*) – `True` iff the WebSocket connection was closed cleanly.
- **code** (*int or None*) – Close status code as sent by the WebSocket peer.
- **reason** (*str or None*) – Close reason as sent by the WebSocket peer.

class autobahn.websocket.interfaces.**IWebSocketServerChannelFactory**(*url=None*, *protocols=None*, *server=None*, *headers=None*, *externalPort=None*)
WebSocket server protocol factories implement this interface, and create protocol instances which in turn implement *autobahn.websocket.interfaces.IWebSocketChannel*.

Parameters

- **url** (*str*) – The WebSocket URL this factory is working for, e.g. `ws://myhost.com/somepath`. For non-TCP transports like pipes or Unix domain sockets, provide `None`. This will use an implicit URL of `ws://localhost`.
- **protocols** (*list of str*) – List of subprotocols the server supports. The subprotocol used is the first from the list of subprotocols announced by the client that is contained in this list.
- **server** (*str*) – Server as announced in HTTP response header during opening handshake.
- **headers** (*dict*) – An optional mapping of additional HTTP headers to send during the WebSocket opening handshake.
- **externalPort** (*int*) – Optionally, the external visible port this server will be reachable under (i.e. when running behind a L2/L3 forwarding device).

setSessionParameters(*url=None*, *protocols=None*, *server=None*, *headers=None*, *externalPort=None*)

Set WebSocket session parameters.

Parameters

- **url** (*str*) – The WebSocket URL this factory is working for, e.g. `ws://myhost.com/somepath`. For non-TCP transports like pipes or Unix domain sockets, provide `None`. This will use an implicit URL of `ws://localhost`.
- **protocols** (*list of str*) – List of subprotocols the server supports. The subprotocol used is the first from the list of subprotocols announced by the client that is contained in this list.
- **server** (*str*) – Server as announced in HTTP response header during opening handshake.
- **headers** (*dict*) – An optional mapping of additional HTTP headers to send during the WebSocket opening handshake.
- **externalPort** (*int*) – Optionally, the external visible port this server will be reachable under (i.e. when running behind a L2/L3 forwarding device).

setProtocolOptions(*versions=None*, *webStatus=None*, *utf8validateIncoming=None*, *maskServerFrames=None*, *requireMaskedClientFrames=None*, *applyMask=None*, *maxFramePayloadSize=None*, *maxMessagePayloadSize=None*, *autoFragmentSize=None*, *failByDrop=None*, *echoCloseCodeReason=None*, *openHandshakeTimeout=None*, *closeHandshakeTimeout=None*, *tcpNoDelay=None*, *perMessageCompressionAccept=None*, *autoPingInterval=None*, *autoPingTimeout=None*, *autoPingSize=None*, *serveFlashSocketPolicy=None*, *flashSocketPolicy=None*, *allowedOrigins=None*, *allowNullOrigin=False*, *maxConnections=None*, *trustXForwardedFor=0*)

Set WebSocket protocol options used as defaults for new protocol instances.

Parameters

- **versions** (*list of ints or None*) – The WebSocket protocol versions accepted by the server (default: `autobahn.websocket.protocol.WebSocketProtocol.SUPPORTED_PROTOCOL_VERSIONS()`).
- **webStatus** (*bool or None*) – Return server status/version on HTTP/GET without WebSocket upgrade header (default: *True*).
- **utf8validateIncoming** (*bool or None*) – Validate incoming UTF-8 in text message payloads (default: *True*).

- **maskServerFrames** (*bool or None*) – Mask server-to-client frames (default: *False*).
- **requireMaskedClientFrames** (*bool or None*) – Require client-to-server frames to be masked (default: *True*).
- **applyMask** (*bool or None*) – Actually apply mask to payload when mask it present. Applies for outgoing and incoming frames (default: *True*).
- **maxFramePayloadSize** (*int or None*) – Maximum frame payload size that will be accepted when receiving or *0* for unlimited (default: *0*).
- **maxMessagePayloadSize** (*int or None*) – Maximum message payload size (after reassembly of fragmented messages) that will be accepted when receiving or *0* for unlimited (default: *0*).
- **autoFragmentSize** (*int or None*) – Automatic fragmentation of outgoing data messages (when using the message-based API) into frames with payload length <= this size or *0* for no auto-fragmentation (default: *0*).
- **failByDrop** – Fail connections by dropping the TCP connection without performing closing handshake (default: *True*).
- **echoCloseCodeReason** (*bool or None*) – Iff true, when receiving a close, echo back close code/reason. Otherwise reply with *code == 1000, reason = ""* (default: *False*).
- **openHandshakeTimeout** (*float or None*) – Opening WebSocket handshake timeout, timeout in seconds or *0* to deactivate (default: *0*).
- **closeHandshakeTimeout** (*float or None*) – When we expect to receive a closing handshake reply, timeout in seconds (default: *1*).
- **tcpNoDelay** (*bool or None*) – TCP NODELAY ("Nagle") socket option (default: *True*).
- **perMessageCompressionAccept** (*callable or None*) – Acceptor function for offers.
- **autoPingInterval** (*float or None*) – Automatically send WebSocket pings every given seconds. When the peer does not respond in *autoPingTimeout*, drop the connection. Set to *0* to disable. (default: *0*).
- **autoPingTimeout** (*float or None*) – Wait this many seconds for the peer to respond to automatically sent pings. If the peer does not respond in time, drop the connection. Set to *0* to disable. (default: *0*).
- **autoPingSize** (*int or None*) – Payload size for automatic pings/pongs. Must be an integer from *[4, 125]*. (default: *4*).
- **serveFlashSocketPolicy** (*bool or None*) – Serve the Flash Socket Policy when we receive a policy file request on this protocol. (default: *False*).
- **flashSocketPolicy** (*str or None*) – The flash socket policy to be served when we are serving the Flash Socket Policy on this protocol and when Flash tried to connect to the destination port. It must end with a null character.
- **allowedOrigins** (*list or None*) – A list of allowed WebSocket origins (with '*' as a wildcard character).
- **allowNullOrigin** (*bool*) – if True, allow WebSocket connections whose *Origin:* is *"null"*.

- **maxConnections** (*int or None*) – Maximum number of concurrent connections. Set to *0* to disable (default: *0*).
- **trustXForwardedFor** (*int*) – Number of trusted web servers in front of this server that add their own X-Forwarded-For header (default: *0*)

resetProtocolOptions()
Reset all WebSocket protocol options to defaults.

class autobahn.websocket.interfaces.**IWebSocketClientChannelFactory**(*url=None*, *origin=None*, *protocols=None*, *useragent=None*, *headers=None*, *proxy=None*)
WebSocket client protocol factories implement this interface, and create protocol instances which in turn implement *autobahn.websocket.interfaces.IWebSocketChannel*.

Note that you MUST provide URL either here or set using autobahn.websocket.WebSocketClientFactory.setSessionParameters() *before* the factory is started.

Parameters

- **url** (*str*) – WebSocket URL this factory will connect to, e.g. ws://myhost.com/somepath?param1=23. For non-TCP transports like pipes or Unix domain sockets, provide None. This will use an implicit URL of ws://localhost.
- **origin** (*str*) – The origin to be sent in WebSocket opening handshake or None (default: *None*).
- **protocols** (*list of strings*) – List of subprotocols the client should announce in WebSocket opening handshake (default: *[]*).
- **useragent** (*str*) – User agent as announced in HTTP request header or None (default: *AutobahnWebSocket/?.?.?*).
- **headers** (*dict*) – An optional mapping of additional HTTP headers to send during the WebSocket opening handshake.
- **proxy** (*dict or None*) – Explicit proxy server to use; a dict with host and port keys

setSessionParameters(*url=None*, *origin=None*, *protocols=None*, *useragent=None*, *headers=None*, *proxy=None*)
Set WebSocket session parameters.

Parameters

- **url** (*str*) – WebSocket URL this factory will connect to, e.g. *ws://myhost.com/somepath?param1=23*. For non-TCP transports like pipes or Unix domain sockets, provide *None*. This will use an implicit URL of *ws://localhost*.
- **origin** (*str*) – The origin to be sent in opening handshake.
- **protocols** (*list of strings*) – List of WebSocket subprotocols the client should announce in opening handshake.
- **useragent** (*str*) – User agent as announced in HTTP request header during opening handshake.

- **headers** (*dict*) – An optional mapping of additional HTTP headers to send during the WebSocket opening handshake.
- **proxy** (*dict or None*) – (Optional) a dict with `host` and `port` keys specifying a proxy to use

setProtocolOptions(*version=None*, *utf8validateIncoming=None*, *acceptMaskedServerFrames=None*, *maskClientFrames=None*, *applyMask=None*, *maxFramePayloadSize=None*, *maxMessagePayloadSize=None*, *autoFragmentSize=None*, *failByDrop=None*, *echoCloseCodeReason=None*, *serverConnectionDropTimeout=None*, *openHandshakeTimeout=None*, *closeHandshakeTimeout=None*, *tcpNoDelay=None*, *perMessageCompressionOffers=None*, *perMessageCompressionAccept=None*, *autoPingInterval=None*, *autoPingTimeout=None*, *autoPingSize=None*)

Set WebSocket protocol options used as defaults for _new_ protocol instances.

Parameters

- **version** (*int*) – The WebSocket protocol spec (draft) version to be used (default: `autobahn.websocket.protocol.WebSocketProtocol.SUPPORTED_PROTOCOL_VERSIONS()`).
- **utf8validateIncoming** (*bool*) – Validate incoming UTF-8 in text message payloads (default: *True*).
- **acceptMaskedServerFrames** (*bool*) – Accept masked server-to-client frames (default: *False*).
- **maskClientFrames** (*bool*) – Mask client-to-server frames (default: *True*).
- **applyMask** (*bool*) – Actually apply mask to payload when mask it present. Applies for outgoing and incoming frames (default: *True*).
- **maxFramePayloadSize** (*int*) – Maximum frame payload size that will be accepted when receiving or *0* for unlimited (default: *0*).
- **maxMessagePayloadSize** (*int*) – Maximum message payload size (after reassembly of fragmented messages) that will be accepted when receiving or *0* for unlimited (default: *0*).
- **autoFragmentSize** (*int*) – Automatic fragmentation of outgoing data messages (when using the message-based API) into frames with payload length <= this size or *0* for no auto-fragmentation (default: *0*).
- **failByDrop** – Fail connections by dropping the TCP connection without performing closing handshake (default: *True*).
- **echoCloseCodeReason** (*bool*) – Iff true, when receiving a close, echo back close code/reason. Otherwise reply with *code == 1000, reason = ""* (default: *False*).
- **serverConnectionDropTimeout** (*float*) – When the client expects the server to drop the TCP, timeout in seconds (default: *1*).
- **openHandshakeTimeout** (*float*) – Opening WebSocket handshake timeout, timeout in seconds or *0* to deactivate (default: *0*).
- **closeHandshakeTimeout** (*float*) – When we expect to receive a closing handshake reply, timeout in seconds (default: *1*).
- **tcpNoDelay** (*bool*) – TCP NODELAY ("Nagle"): bool socket option (default: *True*).
- **perMessageCompressionOffers** (*list of instance of subclass of PerMessageCompressOffer*) – A list of offers to provide to the server for the

permessage-compress WebSocket extension. Must be a list of instances of subclass of PerMessageCompressOffer.

- **perMessageCompressionAccept** (*callable*) – Acceptor function for responses.
- **autoPingInterval** (*float or None*) – Automatically send WebSocket pings every given seconds. When the peer does not respond in *autoPingTimeout*, drop the connection. Set to *0* to disable. (default: *0*).
- **autoPingTimeout** (*float or None*) – Wait this many seconds for the peer to respond to automatically sent pings. If the peer does not respond in time, drop the connection. Set to *0* to disable. (default: *0*).
- **autoPingSize** (*int*) – Payload size for automatic pings/pongs. Must be an integer from *[4, 125]*. (default: *4*).

resetProtocolOptions()
Reset all WebSocket protocol options to defaults.

WebSocket Types

class autobahn.websocket.types.**ConnectionRequest**(*peer*, *headers*, *host*, *path*, *params*, *version*, *origin*, *protocols*, *extensions*)
Thin-wrapper for WebSocket connection request information provided in `autobahn.websocket.protocol.WebSocketServerProtocol.onConnect()` when a WebSocket client want to establish a connection to a WebSocket server.

Parameters

- **peer** (*str*) – Descriptor of the connecting client (e.g. IP address/port in case of TCP transports).
- **headers** (*dict*) – HTTP headers from opening handshake request.
- **host** (*str*) – Host from opening handshake HTTP header.
- **path** (*str*) – Path from requested HTTP resource URI. For example, a resource URI of `/myservice?foo=23&foo=66&bar=2` will be parsed to `/myservice`.
- **params** (*dict*) – Query parameters (if any) from requested HTTP resource URI. For example, a resource URI of `/myservice?foo=23&foo=66&bar=2` will be parsed to `{'foo': ['23', '66'], 'bar': ['2']}`.
- **version** (*int*) – The WebSocket protocol version the client announced (and will be spoken, when connection is accepted).
- **origin** (*str*) – The WebSocket origin header or None. Note that this only a reliable source of information for browser clients!
- **protocols** (*list*) – The WebSocket (sub)protocols the client announced. You must select and return one of those (or `None`) in `autobahn.websocket.WebSocketServerProtocol.onConnect()`.
- **extensions** (*list*) – The WebSocket extensions the client requested and the server accepted, and thus will be spoken, once the WebSocket connection has been fully established.

class autobahn.websocket.types.**ConnectionResponse**(*peer*, *headers*, *version*, *protocol*, *extensions*)
Thin-wrapper for WebSocket connection response information provided in `autobahn.websocket.protocol.WebSocketClientProtocol.onConnect()` when a WebSocket server has accepted a connection request by a client.

Constructor.

Parameters

- **peer** (*str*) – Descriptor of the connected server (e.g. IP address/port in case of TCP transport).
- **headers** (*dict*) – HTTP headers from opening handshake response.
- **version** (*int*) – The WebSocket protocol version that is spoken.
- **protocol** (*str*) – The WebSocket (sub)protocol in use.
- **extensions** (*list of str*) – The WebSocket extensions in use.

class autobahn.websocket.types.**ConnectionAccept**(*subprotocol=None*, *headers=None*)

Used by WebSocket servers to accept an incoming WebSocket connection. If the client announced one or multiple subprotocols, the server MUST select one of the subprotocols announced by the client.

Parameters

- **subprotocol** (*unicode or None*) – The WebSocket connection is accepted with the this WebSocket subprotocol chosen. The value must be a token as defined by RFC 2616.
- **headers** (*dict or None*) – Additional HTTP headers to send on the WebSocket opening handshake reply, e.g. cookies. The keys must be unicode, and the values either unicode or tuple/list. In the latter case a separate HTTP header line will be sent for each item in tuple/list.

exception autobahn.websocket.types.**ConnectionDeny**(*code*, *reason=None*)

Throw an instance of this class to deny a WebSocket connection during handshake in `autobahn.websocket.protocol.WebSocketServerProtocol.onConnect()`.

Parameters

- **code** (*int*) – HTTP error code.
- **reason** (*unicode*) – HTTP error reason.

WebSocket Compression

class autobahn.websocket.compress.**PerMessageDeflateOffer**(*accept_no_context_takeover=True*, *accept_max_window_bits=True*, *request_no_context_takeover=False*, *request_max_window_bits=0*)

Set of extension parameters for *permessage-deflate* WebSocket extension offered by a client to a server.

Parameters

- **accept_no_context_takeover** (*bool*) – When `True`, the client accepts the "no context takeover" feature.
- **accept_max_window_bits** (*bool*) – When `True`, the client accepts setting "max window size".
- **request_no_context_takeover** (*bool*) – When `True`, the client request the "no context takeover" feature.
- **request_max_window_bits** (*int*) – When non-zero, the client requests the given "max window size" (must be and integer from the interval `[8..15]`).

class autobahn.websocket.compress.**PerMessageDeflateOfferAccept**(*offer*, *request_no_context_takeover=False*, *request_max_window_bits=0*, *no_context_takeover=None*, *window_bits=None*, *mem_level=None*)

Set of parameters with which to accept an *permessage-deflate* offer from a client by a server.

Parameters

- **offer** (Instance of autobahn.compress.PerMessageDeflateOffer.) – The offer being accepted.
- **request_no_context_takeover** (*bool*) – When True, the server requests the "no context takeover" feature.
- **request_max_window_bits** – When non-zero, the server requests the given "max window size" (must be and integer from the interval [8..15]).
- **request_max_window_bits** – int
- **no_context_takeover** (*bool*) – Override server ("server-to-client direction") context takeover (this must be compatible with the offer).
- **window_bits** (*int*) – Override server ("server-to-client direction") window size (this must be compatible with the offer).
- **mem_level** (*int*) – Set server ("server-to-client direction") memory level.

class autobahn.websocket.compress.**PerMessageDeflateResponse**(*client_max_window_bits*, *client_no_context_takeover*, *server_max_window_bits*, *server_no_context_takeover*)

Set of parameters for *permessage-deflate* responded by server.

Parameters

- **client_max_window_bits** (*int*) – FIXME
- **client_no_context_takeover** (*bool*) – FIXME
- **server_max_window_bits** (*int*) – FIXME
- **server_no_context_takeover** (*bool*) – FIXME

class autobahn.websocket.compress.**PerMessageDeflateResponseAccept**(*response*, *no_context_takeover=None*, *window_bits=None*, *mem_level=None*)

Set of parameters with which to accept an *permessage-deflate* response from a server by a client.

Parameters

- **response** (Instance of autobahn.compress.PerMessageDeflateResponse.) – The response being accepted.
- **no_context_takeover** (*bool*) – Override client ("client-to-server direction") context takeover (this must be compatible with response).
- **window_bits** (*int*) – Override client ("client-to-server direction") window size (this must be compatible with response).

- **mem_level** (*int*) – Set client ("client-to-server direction") memory level.

WebSocket Utilities

WebSocket utilities that do not depend on the specific networking framework being used (Twisted or asyncio).

autobahn.websocket.util.**create_url**(*hostname*, *port=None*, *isSecure=False*, *path=None*, *params=None*)

Create a WebSocket URL from components.

Parameters

- **hostname** (*str*) – WebSocket server hostname.
- **port** (*int*) – WebSocket service port or None (to select default ports 80/443 depending on isSecure).
- **isSecure** (*bool*) – Set True for secure WebSocket ("wss" scheme).
- **path** (*str*) – Path component of addressed resource (will be properly URL escaped).
- **params** (*dict*) – A dictionary of key-values to construct the query component of the addressed resource (will be properly URL escaped).

Returns str – Constructed WebSocket URL.

autobahn.websocket.util.**parse_url**(*url*)

Parses as WebSocket URL into it's components and returns a tuple (isSecure, host, port, resource, path, params).

- `isSecure` is a flag which is True for wss URLs.
- `host` is the hostname or IP from the URL.
- `port` is the port from the URL or standard port derived from scheme (ws = 80, wss = 443).
- `resource` is the /resource name/ from the URL, the /path/ together with the (optional) /query/ component.
- `path` is the /path/ component properly unescaped.
- `params` is the /query/ component properly unescaped and returned as dictionary.

Parameters **url** (*str*) – A valid WebSocket URL, i.e. `ws://localhost:9000/myresource?param1=23&param2=456`

Returns tuple – A tuple (isSecure, host, port, resource, path, params)

Module autobahn.rawsocket

WAMP-RawSocket is an alternative WAMP transport that has less overhead compared to WebSocket, and is vastly simpler to implement. It can run over any stream based underlying transport, such as TCP or Unix domain socket. However, it does NOT run into the browser.

RawSocket Utilities

RawSocket utilities that do not depend on the specific networking framework being used (Twisted or asyncio).

autobahn.rawsocket.util.**create_url**(*hostname*, *port=None*, *isSecure=False*)

Create a RawSocket URL from components.

Parameters

- **hostname** (*str*) – RawSocket server hostname.
- **port** (*int*) – RawSocket service port or None (to select default ports `80` or `443` depending on `isSecure`.
- **isSecure** (*bool*) – Set `True` for secure RawSocket (`rss` scheme).

Returns Constructed RawSocket URL.

Return type str

`autobahn.rawsocket.util.`**parse_url**(*url*)

Parses as RawSocket URL into it's components and returns a tuple (isSecure, host, port).

- `isSecure` is a flag which is `True` for `rss` URLs.
- `host` is the hostname or IP from the URL.
- `port` is the port from the URL or standard port derived from scheme (`rs` => `80`, `rss` => `443`).

Parameters **url** (*str*) – A valid RawSocket URL, i.e. `rs://localhost:9000`

Returns A tuple `(isSecure, host, port)`.

Return type tuple

Module autobahn.wamp

WAMP Interfaces

class `autobahn.wamp.interfaces.`**IObjectSerializer**

Raw Python object serialization and deserialization. Object serializers are used by classes implementing WAMP serializers, that is instances of *autobahn.wamp.interfaces.ISerializer*.

BINARY

Flag (read-only) to indicate if serializer requires a binary clean transport or if UTF8 transparency is sufficient.

serialize(*obj*)

Serialize an object to a byte string.

Parameters **obj** (*any (serializable type)*) – Object to serialize.

Returns Serialized bytes.

Return type bytes

unserialize(*payload*)

Unserialize objects from a byte string.

Parameters **payload** (*bytes*) – Objects to unserialize.

Returns List of (raw) objects unserialized.

Return type list

class `autobahn.wamp.interfaces.`**ISerializer**

WAMP message serialization and deserialization.

MESSAGE_TYPE_MAP

Mapping of WAMP message type codes to WAMP message classes.

SERIALIZER_ID
The WAMP serialization format ID.

serialize(*message*)
Serializes a WAMP message to bytes for sending over a transport.

Parameters **message** (object implementing *autobahn.wamp.interfaces.IMessage*) – The WAMP message to be serialized.

Returns A pair `(payload, isBinary)`.

Return type tuple

unserialize(*payload*, *isBinary*)
Deserialize bytes from a transport and parse into WAMP messages.

Parameters

- **payload** (*bytes*) – Byte string from wire.
- **is_binary** (*bool*) – Type of payload. True if payload is a binary string, else the payload is UTF-8 encoded Unicode text.

Returns List of `a.w.m.Message` objects.

Return type list

class autobahn.wamp.interfaces.**IMessage**

MESSAGE_TYPE
WAMP message type code.

serialize(*serializer*)
Serialize this object into a wire level bytes representation and cache the resulting bytes. If the cache already contains an entry for the given serializer, return the cached representation directly.

Parameters **serializer** (object implementing *autobahn.wamp.interfaces.ISerializer*) – The wire level serializer to use.

Returns The serialized bytes.

Return type bytes

uncache()
Resets the serialization cache for this message.

class autobahn.wamp.interfaces.**ITransport**
A WAMP transport is a bidirectional, full-duplex, reliable, ordered, message-based channel.

send(*message*)
Send a WAMP message over the transport to the peer. If the transport is not open, this raises *autobahn.wamp.exception.TransportLost*. Returns a deferred/future when the message has been processed and more messages may be sent. When send() is called while a previous deferred/future has not yet fired, the send will fail immediately.

Parameters **message** (object implementing *autobahn.wamp.interfaces.IMessage*) – The WAMP message to send over the transport.

Returns obj – A Deferred/Future

isOpen()
Check if the transport is open for messaging.

Returns `True`, if the transport is open.

Return type bool

close()
Close the transport regularly. The transport will perform any closing handshake if applicable. This should be used for any application initiated closing.

abort()
Abort the transport abruptly. The transport will be destroyed as fast as possible, and without playing nice to the peer. This should only be used in case of fatal errors, protocol violations or possible detected attacks.

get_channel_id()
Return the unique channel ID of the underlying transport. This is used to mitigate credential forwarding man-in-the-middle attacks when running application level authentication (eg WAMP-cryptosign) which are decoupled from the underlying transport.

The channel ID is only available when running over TLS (either WAMP-WebSocket or WAMP-RawSocket). It is not available for non-TLS transports (plain TCP or Unix domain sockets). It is also not available for WAMP-over-HTTP/Longpoll. Further, it is currently unimplemented for asyncio (only works on Twisted).

The channel ID is computed as follows:

- for a client, the SHA256 over the "TLS Finished" message sent by the client to the server is returned.
- for a server, the SHA256 over the "TLS Finished" message the server expected the client to send

Note: this is similar to *tls-unique* as described in RFC5929, but instead of returning the raw "TLS Finished" message, it returns a SHA256 over such a message. The reason is that we use the channel ID mainly with WAMP-cryptosign, which is based on Ed25519, where keys are always 32 bytes. And having a channel ID which is always 32 bytes (independent of the TLS ciphers/hashfuns in use) allows use to easily XOR channel IDs with Ed25519 keys and WAMP-cryptosign challenges.

WARNING: For safe use of this (that is, for safely binding app level authentication to the underlying transport), you MUST use TLS, and you SHOULD deactivate both TLS session renegotiation and TLS session resumption.

References:

- https://tools.ietf.org/html/rfc5056
- https://tools.ietf.org/html/rfc5929
- http://www.pyopenssl.org/en/stable/api/ssl.html#OpenSSL.SSL.Connection.get_finished
- http://www.pyopenssl.org/en/stable/api/ssl.html#OpenSSL.SSL.Connection.get_peer_finished

Returns The channel ID (if available) of the underlying WAMP transport. The channel ID is a 32 bytes value.

Return type binary or None

class autobahn.wamp.interfaces.**ITransportHandler**

transport
When the transport this handler is attached to is currently open, this property can be read from. The property should be considered read-only. When the transport is gone, this property is set to None.

onOpen(*transport*)
Callback fired when transport is open. May run asynchronously. The transport is considered running and is_open() would return true, as soon as this callback has completed successfully.

Parameters **transport** (object implementing *autobahn.wamp.interfaces.ITransport*) – The WAMP transport.

onMessage(*message*)
Callback fired when a WAMP message was received. May run asynchronously. The callback should return or fire the returned deferred/future when it's done processing the message. In particular, an implementation of this callback must not access the message afterwards.

Parameters **message** (object implementing *autobahn.wamp.interfaces.IMessage*) – The WAMP message received.

onClose(*wasClean*)
Callback fired when the transport has been closed.

Parameters **wasClean** (*bool*) – Indicates if the transport has been closed regularly.

class autobahn.wamp.interfaces.**ISession**(*config=None*)
Interface for WAMP sessions.

Parameters **config** (instance of *autobahn.wamp.types.ComponentConfig*.) – Configuration for session.

onUserError(*fail*, *msg*)
This is called when we try to fire a callback, but get an exception from user code – for example, a registered publish callback or a registered method. By default, this prints the current stack-trace and then error-message to stdout.

ApplicationSession-derived objects may override this to provide logging if they prefer. The Twisted implemention does this. (See *autobahn.twisted.wamp.ApplicationSession*)

Parameters

- **fail** (*instance implementing txaio.IFailedFuture*) – The failure that occurred.
- **msg** (*str*) – an informative message from the library. It is suggested you log this immediately after the exception.

onConnect()
Callback fired when the transport this session will run over has been established.

join(*realm*, *authmethods=None*, *authid=None*, *authrole=None*, *authextra=None*, *resumable=None*, *resume_session=None*, *resume_token=None*)
Attach the session to the given realm. A session is open as soon as it is attached to a realm.

onChallenge(*challenge*)
Callback fired when the peer demands authentication.

May return a Deferred/Future.

Parameters **challenge** (Instance of *autobahn.wamp.types.Challenge*.) – The authentication challenge.

onJoin(*details*)
Callback fired when WAMP session has been established.

May return a Deferred/Future.

Parameters **details** (Instance of *autobahn.wamp.types.SessionDetails*.) – Session information.

leave(*reason=None*, *message=None*)
Actively close this WAMP session.

Parameters

- **reason** (*str*) – An optional URI for the closing reason. If you want to permanently log out, this should be *wamp.close.logout*
- **message** (*str*) – An optional (human readable) closing message, intended for logging purposes.

Returns may return a Future/Deferred that fires when we've disconnected

onLeave(*details*)
Callback fired when WAMP session has is closed

Parameters **details** (Instance of *autobahn.wamp.types.CloseDetails*.) – Close information.

disconnect()
Close the underlying transport.

onDisconnect()
Callback fired when underlying transport has been closed.

is_connected()
Check if the underlying transport is connected.

is_attached()
Check if the session has currently joined a realm.

set_payload_codec(*payload_codec*)
Set a payload codec on the session. To remove a previously set payload codec, set the codec to `None`.

Payload codecs are used with WAMP payload transparency mode.

Parameters **payload_codec** (object implementing *autobahn.wamp.interfaces.IPayloadCodec* or `None`) – The payload codec that should process application payload of the given encoding.

get_payload_codec()
Get the current payload codec (if any) for the session.

Payload codecs are used with WAMP payload transparency mode.

Returns The current payload codec or `None` if no codec is active.

Return type object implementing *autobahn.wamp.interfaces.IPayloadCodec* or `None`

define(*exception*, *error=None*)
Defines an exception for a WAMP error in the context of this WAMP session.

Parameters

- **exception** (A class that derives of `Exception`.) – The exception class to define an error mapping for.
- **error** (*str*) – The URI (or URI pattern) the exception class should be mapped for. Iff the `exception` class is decorated, this must be `None`.

call(*procedure*, **args*, ***kwargs*)
Call a remote procedure.

This will return a Deferred/Future, that when resolved, provides the actual result returned by the called remote procedure.

•If the result is a single positional return value, it'll be returned "as-is".

- If the result contains multiple positional return values or keyword return values, the result is wrapped in an instance of *autobahn.wamp.types.CallResult*.
- If the call fails, the returned Deferred/Future will be rejected with an instance of *autobahn.wamp.exception.ApplicationError*.

If `kwargs` contains an `options` keyword argument that is an instance of *autobahn.wamp.types.CallOptions*, this will provide specific options for the call to perform.

When the *Caller* and *Dealer* implementations support canceling of calls, the call may be canceled by canceling the returned Deferred/Future.

Parameters

- **procedure** (*unicode*) – The URI of the remote procedure to be called, e.g. `u"com.myapp.hello"`.
- **args** (*list*) – Any positional arguments for the call.
- **kwargs** (*dict*) – Any keyword arguments for the call.

Returns A Deferred/Future for the call result -

Return type instance of twisted.internet.defer.Deferred / `asyncio.Future`

register(*endpoint*, *procedure=None*, *options=None*)

Register a procedure for remote calling.

When `endpoint` is a callable (function, method or object that implements `__call__`), then `procedure` must be provided and an instance of twisted.internet.defer.Deferred (when running on **Twisted**) or an instance of `asyncio.Future` (when running on **asyncio**) is returned.

- If the registration *succeeds* the returned Deferred/Future will *resolve* to an object that implements `autobahn.wamp.interfaces.IRegistration`.
- If the registration *fails* the returned Deferred/Future will *reject* with an instance of *autobahn.wamp.exception.ApplicationError*.

When `endpoint` is an object, then each of the object's methods that is decorated with `autobahn.wamp.register()` is automatically registered and a (single) DeferredList or Future is returned that gathers all individual underlying Deferreds/Futures.

Parameters

- **endpoint** (*callable or object*) – The endpoint called under the procedure.
- **procedure** (*unicode*) – When `endpoint` is a callable, the URI (or URI pattern) of the procedure to register for. When `endpoint` is an object, the argument is ignored (and should be `None`).
- **options** (instance of *autobahn.wamp.types.RegisterOptions*.) – Options for registering.

Returns A registration or a list of registrations (or errors)

Return type instance(s) of twisted.internet.defer.Deferred / `asyncio.Future`

publish(*topic*, **args*, ***kwargs*)

Publish an event to a topic.

If `kwargs` contains an `options` keyword argument that is an instance of *autobahn.wamp.types.PublishOptions*, this will provide specific options for the publish to perform.

Note: By default, publications are non-acknowledged and the publication can fail silently, e.g. because the session is not authorized to publish to the topic.

When publication acknowledgement is requested via `options.acknowledge == True`, this function returns a Deferred/Future:

- If the publication succeeds the Deferred/Future will resolve to an object that implements `autobahn.wamp.interfaces.IPublication`.
- If the publication fails the Deferred/Future will reject with an instance of *autobahn.wamp.exception.ApplicationError*.

Parameters

- **topic** (*unicode*) – The URI of the topic to publish to, e.g. `u"com.myapp.mytopic1"`.
- **args** (*list*) – Arbitrary application payload for the event (positional arguments).
- **kwargs** (*dict*) – Arbitrary application payload for the event (keyword arguments).

Returns Acknowledgement for acknowledge publications - otherwise nothing.

Return type `None` or instance of twisted.internet.defer.Deferred / `asyncio.Future`

subscribe(*handler*, *topic=None*, *options=None*)

Subscribe to a topic for receiving events.

When `handler` is a callable (function, method or object that implements `__call__`), then *topic* must be provided and an instance of twisted.internet.defer.Deferred (when running on **Twisted**) or an instance of `asyncio.Future` (when running on **asyncio**) is returned.

- If the subscription succeeds the Deferred/Future will resolve to an object that implements `autobahn.wamp.interfaces.ISubscription`.
- If the subscription fails the Deferred/Future will reject with an instance of *autobahn.wamp.exception.ApplicationError*.

When `handler` is an object, then each of the object's methods that is decorated with `autobahn.wamp.subscribe()` is automatically subscribed as event handlers, and a list of Deferreds/Futures is returned that each resolves or rejects as above.

Parameters

- **handler** (*callable or object*) – The event handler to receive events.
- **topic** (*unicode*) – When `handler` is a callable, the URI (or URI pattern) of the topic to subscribe to. When `handler` is an object, this value is ignored (and should be `None`).
- **options** (An instance of *autobahn.wamp.types.SubscribeOptions*.) – Options for subscribing.

Returns A single Deferred/Future or a list of such objects

Return type instance(s) of twisted.internet.defer.Deferred / `asyncio.Future`

class `autobahn.wamp.interfaces.`**IPayloadCodec**

WAMP payload codecs are used with WAMP payload transparency mode.

In payload transparency mode, application payloads are transmitted "raw", as binary strings, without any processing at the WAMP router.

Payload transparency can be used eg for these use cases:

- end-to-end encryption of application payloads (WAMP-cryptobox)
- using serializers with custom user types, where the serializer and the serializer implementation has native support for serializing custom types (such as CBOR)
- transmitting MQTT payloads within WAMP, when the WAMP router is providing a MQTT-WAMP bridge

encode(*is_originating*, *uri*, *args=None*, *kwargs=None*)
Encodes application payload.

Parameters

- **is_originating** (*bool*) – Flag indicating whether the encoding is to be done from an originator (a caller or publisher).
- **uri** (*str*) – The WAMP URI associated with the WAMP message for which the payload is to be encoded (eg topic or procedure).
- **args** (*list or None*) – Positional application payload.
- **kwargs** (*dict or None*) – Keyword-based application payload.

Returns The encoded application payload or None to signal no encoding should be used.

Return type instance of *autobahn.wamp.types.EncodedPayload*

decode(*is_originating*, *uri*, *encoded_payload*)
Decode application payload.

Parameters

- **is_originating** (*bool*) – Flag indicating whether the encoding is to be done from an originator (a caller or publisher).
- **uri** (*str*) – The WAMP URI associated with the WAMP message for which the payload is to be encoded (eg topic or procedure).
- **payload** (instance of *autobahn.wamp.types.EncodedPayload*) – The encoded application payload to be decoded.

Returns A tuple with the decoded positional and keyword-based application payload: `(uri, args, kwargs)`

Return type tuple

WAMP Types

class `autobahn.wamp.types.`**ComponentConfig**(*realm=None*, *extra=None*, *keyring=None*, *controller=None*, *shared=None*)
WAMP application component configuration. An instance of this class is provided to the constructor of `autobahn.wamp.protocol.ApplicationSession`.

Parameters

- **realm** (*str*) – The realm the session would like to join or `None` to let the router auto-decide the realm (if the router is configured and allowing to do so).
- **extra** (*arbitrary*) – Optional user-supplied object with extra configuration. This can be any object you like, and is accessible in your *ApplicationSession* subclass via *self.config.extra*. *dict* is a good default choice. Important: if the component is to be hosted by Crossbar.io, the supplied value must be JSON serializable.

- **keyring** (*obj implementing IKeyRing or None*) – A mapper from WAMP URIs to "from"/"to" Ed25519 keys. When using WAMP end-to-end encryption, application payload is encrypted using a symmetric message key, which in turn is encrypted using the "to" URI (topic being published to or procedure being called) public key and the "from" URI private key. In both cases, the key for the longest matching URI is used.
- **controller** (*instance of ApplicationSession or None*) – A WAMP ApplicationSession instance that holds a session to a controlling entity. This optional feature needs to be supported by a WAMP component hosting run-time.
- **shared** (*dict or None*) – A dict object to exchange user information or hold user objects shared between components run under the same controlling entity. This optional feature needs to be supported by a WAMP component hosting run-time. Use with caution, as using this feature can introduce coupling between components. A valid use case would be to hold a shared database connection pool.

class autobahn.wamp.types.**HelloReturn**
Base class for HELLO return information.

class autobahn.wamp.types.**Accept**(*realm=None*, *authid=None*, *authrole=None*, *authmethod=None*, *authprovider=None*, *authextra=None*)
Information to accept a HELLO.

Parameters

- **realm** (*str*) – The realm the client is joined to.
- **authid** (*str*) – The authentication ID the client is assigned, e.g. "joe" or "joe@example.com".
- **authrole** (*str*) – The authentication role the client is assigned, e.g. "anonymous", "user" or "com.myapp.user".
- **authmethod** (*str*) – The authentication method that was used to authenticate the client, e.g. "cookie" or "wampcra".
- **authprovider** (*str*) – The authentication provider that was used to authenticate the client, e.g. "mozilla-persona".
- **authextra** (*dict*) – Application-specific authextra to be forwarded to the client in *WELCOME.details.authextra*.

class autobahn.wamp.types.**Deny**(*reason=u'wamp.error.not_authorized'*, *message=None*)
Information to deny a HELLO.

Parameters

- **reason** (*str*) – The reason of denying the authentication (an URI, e.g. u'wamp.error.not_authorized')
- **message** (*str*) – A human readable message (for logging purposes).

class autobahn.wamp.types.**Challenge**(*method*, *extra=None*)
Information to challenge the client upon HELLO.

Parameters

- **method** (*str*) – The authentication method for the challenge (e.g. "wampcra").
- **extra** (*dict*) – Any extra information for the authentication challenge. This is specific to the authentication method.

class autobahn.wamp.types.**HelloDetails**(*realm=None*, *authmethods=None*, *authid=None*, *authrole=None*, *authextra=None*, *session_roles=None*, *pending_session=None*, *resumable=None*, *resume_session=None*, *resume_token=None*)

Provides details of a WAMP session while still attaching.

Parameters

- **realm** (*str or None*) – The realm the client wants to join.
- **authmethods** (*list of str or None*) – The authentication methods the client is willing to perform.
- **authid** (*str or None*) – The authid the client wants to authenticate as.
- **authrole** (*str or None*) – The authrole the client wants to authenticate as.
- **authextra** (*arbitrary or None*) – Any extra information the specific authentication method requires the client to send.
- **session_roles** (*dict or None*) – The WAMP session roles and features by the connecting client.
- **pending_session** (*int or None*) – The session ID the session will get once successfully attached.
- **resumable** (*bool or None*) –
- **resume_session** (*int or None*) – The session the client would like to resume.
- **resume_token** (*str or None*) – The secure authorisation token to resume the session.

class autobahn.wamp.types.**SessionDetails**(*realm*, *session*, *authid=None*, *authrole=None*, *authmethod=None*, *authprovider=None*, *authextra=None*, *resumed=None*, *resumable=None*, *resume_token=None*)

Provides details for a WAMP session upon open.

See also:

autobahn.wamp.interfaces.ISession.onJoin()

Parameters

- **realm** (*str*) – The realm this WAMP session is attached to.
- **session** (*int*) – WAMP session ID of this session.
- **resumed** (*bool or None*) – Whether the session is a resumed one.
- **resumable** (*bool or None*) – Whether this session can be resumed later.
- **resume_token** (*str or None*) – The secure authorisation token to resume the session.

class autobahn.wamp.types.**CloseDetails**(*reason=None*, *message=None*)

Provides details for a WAMP session upon close.

See also:

autobahn.wamp.interfaces.ISession.onLeave()

Parameters

- **reason** (*str*) – The close reason (an URI, e.g. `wamp.close.normal`)
- **message** (*str*) – Closing log message.

class `autobahn.wamp.types.`**SubscribeOptions**(*match=None*, *details=None*, *details_arg=None*, *get_retained=None*)

Used to provide options for subscribing in `autobahn.wamp.interfaces.ISubscriber.subscribe()`.

Parameters

- **match** (*str*) – The topic matching method to be used for the subscription.
- **details** (*bool*) – When invoking the handler, provide event details in a keyword parameter `details`.
- **details_arg** (*str*) – DEPCREATED (use "details" flag). When invoking the handler provide event details in this keyword argument to the callable.
- **get_retained** (*bool or None*) – Whether the client wants the retained message we may have along with the subscription.

class `autobahn.wamp.types.`**EventDetails**(*subscription*, *publication*, *publisher=None*, *publisher_authid=None*, *publisher_authrole=None*, *topic=None*, *retained=None*, *enc_algo=None*)

Provides details on an event when calling an event handler previously registered.

Parameters

- **subscription** (instance of `autobahn.wamp.request.Subscription`) – The (client side) subscription object on which this event is delivered.
- **publication** (*int*) – The publication ID of the event (always present).
- **publisher** (*None or int*) – The WAMP session ID of the original publisher of this event. Only filled when publisher is disclosed.
- **publisher_authid** (*str or None*) – The WAMP authid of the original publisher of this event. Only filled when publisher is disclosed.
- **publisher_authrole** (*str or None*) – The WAMP authrole of the original publisher of this event. Only filled when publisher is disclosed.
- **topic** (*str or None*) – For pattern-based subscriptions, the actual topic URI being published to. Only filled for pattern-based subscriptions.
- **retained** (*bool or None*) – Whether the message was retained by the broker on the topic, rather than just published.
- **enc_algo** (*str or None*) – Payload encryption algorithm that was in use (currently, either `None` or `u'cryptobox'`).

class `autobahn.wamp.types.`**PublishOptions**(*acknowledge=None*, *exclude_me=None*, *exclude=None*, *exclude_authid=None*, *exclude_authrole=None*, *eligible=None*, *eligible_authid=None*, *eligible_authrole=None*, *retain=None*)

Used to provide options for subscribing in `autobahn.wamp.interfaces.IPublisher.publish()`.

Parameters

- **acknowledge** (*bool*) – If `True`, acknowledge the publication with a success or error response.

- **exclude_me** (*bool or None*) – If `True`, exclude the publisher from receiving the event, even if he is subscribed (and eligible).
- **exclude** (*int or list of int or None*) – A single WAMP session ID or a list thereof to exclude from receiving this event.
- **exclude_authid** (*str or list of str or None*) – A single WAMP authid or a list thereof to exclude from receiving this event.
- **exclude_authrole** (*list of str or None*) – A single WAMP authrole or a list thereof to exclude from receiving this event.
- **eligible** (*int or list of int or None*) – A single WAMP session ID or a list thereof eligible to receive this event.
- **eligible_authid** (*str or list of str or None*) – A single WAMP authid or a list thereof eligible to receive this event.
- **eligible_authrole** (*str or list of str or None*) – A single WAMP authrole or a list thereof eligible to receive this event.
- **retain** (*bool or None*) – If `True`, request the broker retain this event.

class `autobahn.wamp.types.`**RegisterOptions**(*match=None*, *invoke=None*, *concurrency=None*, *details_arg=None*)

Used to provide options for registering in `autobahn.wamp.interfaces.ICallee.register()`.

Parameters **details_arg** (*str*) – When invoking the endpoint, provide call details in this keyword argument to the callable.

class `autobahn.wamp.types.`**CallDetails**(*registration*, *progress=None*, *caller=None*, *caller_authid=None*, *caller_authrole=None*, *procedure=None*, *enc_algo=None*)

Provides details on a call when an endpoint previously registered is being called and opted to receive call details.

Parameters

- **registration** (instance of `autobahn.wamp.request.Registration`) – The (client side) registration object this invocation is delivered on.
- **progress** (*callable or None*) – A callable that will receive progressive call results.
- **caller** (*int or None*) – The WAMP session ID of the caller, if the latter is disclosed. Only filled when caller is disclosed.
- **caller_authid** (*str or None*) – The WAMP authid of the original caller of this event. Only filled when caller is disclosed.
- **caller_authrole** (*str or None*) – The WAMP authrole of the original caller of this event. Only filled when caller is disclosed.
- **procedure** (*str or None*) – For pattern-based registrations, the actual procedure URI being called.
- **enc_algo** (*str or None*) – Payload encryption algorithm that was in use (currently, either *None* or *"cryptobox"*).

class `autobahn.wamp.types.`**CallOptions**(*on_progress=None*, *timeout=None*)

Used to provide options for calling with `autobahn.wamp.interfaces.ICaller.call()`.

Parameters

- **on_progress** (*callable*) – A callback that will be called when the remote endpoint called yields interim call progress results.

- **timeout** (*float*) – Time in seconds after which the call should be automatically canceled.

class autobahn.wamp.types.**CallResult**(**results*, ***kwresults*)
Wrapper for remote procedure call results that contain multiple positional return values or keyword-based return values.

Parameters

- **results** (*list*) – The positional result values.
- **kwresults** (*dict*) – The keyword result values.

class autobahn.wamp.types.**EncodedPayload**(*payload*, *enc_algo*, *enc_serializer=None*, *enc_key=None*)
Wrapper holding an encoded application payload when using WAMP payload transparency.

Parameters

- **payload** (*bytes*) – The encoded application payload.
- **enc_algo** (*str*) – The payload transparency algorithm identifier to check.
- **enc_serializer** (*str*) – The payload transparency serializer identifier to check.
- **enc_key** (*str or None*) – If using payload transparency with an encryption algorithm, the payload encryption key.

WAMP Exceptions

exception autobahn.wamp.exception.**Error**
Base class for all exceptions related to WAMP.

exception autobahn.wamp.exception.**SessionNotReady**
The application tried to perform a WAMP interaction, but the session is not yet fully established.

exception autobahn.wamp.exception.**SerializationError**
Exception raised when the WAMP serializer could not serialize the application payload (`args` or `kwargs` for `CALL`, `PUBLISH`, etc).

exception autobahn.wamp.exception.**ProtocolError**
Exception raised when WAMP protocol was violated. Protocol errors are fatal and are handled by the WAMP implementation. They are not supposed to be handled at the application level.

exception autobahn.wamp.exception.**TransportLost**
Exception raised when the transport underlying the WAMP session was lost or is not connected.

exception autobahn.wamp.exception.**ApplicationError**(*error*, **args*, ***kwargs*)
Base class for all exceptions that can/may be handled at the application level.

Parameters **error** (*str*) – The URI of the error that occurred, e.g. `wamp.error.not_authorized`.

error_message()
Get the error message of this exception.

Returns The error message.

Return type str

WAMP Authentication and Encryption

autobahn.wamp.auth.**generate_totp_secret**(*length=10*)
Generates a new Base32 encoded, random secret.

See also:

http://en.wikipedia.org/wiki/Base32

Parameters **length** (*int*) – The length of the entropy used to generate the secret.

Returns The generated secret in Base32 (letters `A-Z` and digits `2-7`). The length of the generated secret is `length * 8 / 5` octets.

Return type unicode

autobahn.wamp.auth.**compute_totp**(*secret*, *offset=0*)
Computes the current TOTP code.

Parameters

- **secret** (*unicode*) – Base32 encoded secret.
- **offset** (*int*) – Time offset (in steps, use eg -1, 0, +1 for compliance with RFC6238) for which to compute TOTP.

Returns TOTP for current time (+/- offset).

Return type unicode

autobahn.wamp.auth.**pbkdf2**(*data*, *salt*, *iterations=1000*, *keylen=32*, *hashfunc=None*)
Returns a binary digest for the PBKDF2 hash algorithm of `data` with the given `salt`. It iterates `iterations` time and produces a key of `keylen` bytes. By default SHA-256 is used as hash function, a different hashlib `hashfunc` can be provided.

Parameters

- **data** (*bytes*) – The data for which to compute the PBKDF2 derived key.
- **salt** (*bytes*) – The salt to use for deriving the key.
- **iterations** (*int*) – The number of iterations to perform in PBKDF2.
- **keylen** (*int*) – The length of the cryptographic key to derive.
- **hashfunc** (*callable*) – The hash function to use, e.g. `hashlib.sha1`.

Returns The derived cryptographic key.

Return type bytes

autobahn.wamp.auth.**derive_key**(*secret*, *salt*, *iterations=1000*, *keylen=32*)
Computes a derived cryptographic key from a password according to PBKDF2.

See also:

http://en.wikipedia.org/wiki/PBKDF2

Parameters

- **secret** (*bytes or unicode*) – The secret.
- **salt** (*bytes or unicode*) – The salt to be used.
- **iterations** (*int*) – Number of iterations of derivation algorithm to run.

- **keylen** (*int*) – Length of the key to derive in bytes.

Returns The derived key in Base64 encoding.

Return type bytes

autobahn.wamp.auth.**generate_wcs**(*length=14*)

Generates a new random secret for use with WAMP-CRA.

The secret generated is a random character sequence drawn from

•upper and lower case latin letters

•digits

•

Parameters **length** (*int*) – The length of the secret to generate.

Returns The generated secret. The length of the generated is `length` octets.

Return type bytes

autobahn.wamp.auth.**compute_wcs**(*key*, *challenge*)

Compute an WAMP-CRA authentication signature from an authentication challenge and a (derived) key.

Parameters

- **key** (*bytes*) – The key derived (via PBKDF2) from the secret.
- **challenge** (*bytes*) – The authentication challenge to sign.

Returns The authentication signature.

Return type bytes

class autobahn.wamp.cryptobox.**EncodedPayload**(*payload*, *enc_algo*, *enc_serializer=None*, *enc_key=None*)

Wrapper holding an encoded application payload when using WAMP payload transparency.

Parameters

- **payload** (*bytes*) – The encoded application payload.
- **enc_algo** (*str*) – The payload transparency algorithm identifier to check.
- **enc_serializer** (*str*) – The payload transparency serializer identifier to check.
- **enc_key** (*str or None*) – If using payload transparency with an encryption algorithm, the payload encryption key.

Module autobahn.twisted

Autobahn Twisted specific classes. These are used when Twisted is run as the underlying networking framework.

WebSocket Protocols and Factories

Classes for WebSocket clients and servers using Twisted.

class autobahn.twisted.websocket.**WebSocketServerProtocol**
Base class for Twisted-based WebSocket server protocols.

Implements *autobahn.websocket.interfaces.IWebSocketChannel*.

class autobahn.twisted.websocket.**WebSocketClientProtocol**
Base class for Twisted-based WebSocket client protocols.

Implements *autobahn.websocket.interfaces.IWebSocketChannel*.

class autobahn.twisted.websocket.**WebSocketServerFactory**(**args*, ***kwargs*)
Base class for Twisted-based WebSocket server factories.

Implements *autobahn.websocket.interfaces.IWebSocketServerChannelFactory*

Note: In addition to all arguments to the constructor of *autobahn.websocket.interfaces.IWebSocketServerChannelFactory()*, you can supply a `reactor` keyword argument to specify the Twisted reactor to be used.

---

class autobahn.twisted.websocket.**WebSocketClientFactory**(**args*, ***kwargs*)
Base class for Twisted-based WebSocket client factories.

Implements *autobahn.websocket.interfaces.IWebSocketClientChannelFactory*

Note: In addition to all arguments to the constructor of *autobahn.websocket.interfaces.IWebSocketClientChannelFactory()*, you can supply a `reactor` keyword argument to specify the Twisted reactor to be used.

WAMP-over-WebSocket Protocols and Factories

Classes for WAMP-WebSocket clients and servers using Twisted.

class autobahn.twisted.websocket.**WampWebSocketServerProtocol**
Twisted-based WAMP-over-WebSocket server protocol.

Implements:

- *autobahn.wamp.interfaces.ITransport*

class autobahn.twisted.websocket.**WampWebSocketClientProtocol**
Twisted-based WAMP-over-WebSocket client protocol.

Implements:

- *autobahn.wamp.interfaces.ITransport*

class autobahn.twisted.websocket.**WampWebSocketServerFactory**(*factory*, **args*, ***kwargs*)
Twisted-based WAMP-over-WebSocket server protocol factory.

Parameters

- **factory** (*callable*) – A callable that produces instances that implement *autobahn.wamp.interfaces.ITransportHandler*
- **serializers** (list of objects implementing *autobahn.wamp.interfaces.ISerializer*) – A list of WAMP serializers to use (or `None` for all available serializers).

protocol
alias of `WampWebSocketServerProtocol`

class `autobahn.twisted.websocket.`**`WampWebSocketClientFactory`**(*factory*, **args*, ***kwargs*)
Twisted-based WAMP-over-WebSocket client protocol factory.

Parameters

- **factory** (*callable*) – A callable that produces instances that implement *autobahn.wamp.interfaces.ITransportHandler*
- **serializer** (object implementing *autobahn.wamp.interfaces.ISerializer*) – The WAMP serializer to use (or `None` for "best" serializer, chosen as the first serializer available from this list: CBOR, MessagePack, UBJSON, JSON).

protocol
alias of `WampWebSocketClientProtocol`

WAMP-over-RawSocket Protocols and Factories

Classes for WAMP-RawSocket clients and servers using Twisted.

class `autobahn.twisted.rawsocket.`**`WampRawSocketServerProtocol`**
Twisted-based WAMP-over-RawSocket server protocol.

Implements:

- *autobahn.wamp.interfaces.ITransport*

class `autobahn.twisted.rawsocket.`**`WampRawSocketClientProtocol`**
Twisted-based WAMP-over-RawSocket client protocol.

Implements:

- *autobahn.wamp.interfaces.ITransport*

class `autobahn.twisted.rawsocket.`**`WampRawSocketServerFactory`**(*factory*, *serializers=None*)
Twisted-based WAMP-over-RawSocket server protocol factory.

Parameters

- **factory** (*callable*) – A callable that produces instances that implement *autobahn.wamp.interfaces.ITransportHandler*
- **serializers** (list of objects implementing *autobahn.wamp.interfaces.ISerializer*) – A list of WAMP serializers to use (or `None` for all available serializers).

protocol
alias of `WampRawSocketServerProtocol`

class `autobahn.twisted.rawsocket.`**`WampRawSocketClientFactory`**(*factory*, *serializer=None*)
Twisted-based WAMP-over-RawSocket client protocol factory.

Parameters

- **factory** (*callable*) – A callable that produces instances that implement *autobahn.wamp.interfaces.ITransportHandler*
- **serializer** (object implementing *autobahn.wamp.interfaces.ISerializer*) – The WAMP serializer to use (or `None` for "best" serializer, chosen as the first serializer available from this list: CBOR, MessagePack, UBJSON, JSON).

protocol
alias of `WampRawSocketClientProtocol`

WAMP Sessions

Classes for WAMP sessions using Twisted.

class `autobahn.twisted.wamp.`**`ApplicationSession`**(*config=None*)
WAMP application session for Twisted-based applications.

Implements:

- *autobahn.wamp.interfaces.ITransportHandler*
- *autobahn.wamp.interfaces.ISession*

Implements *autobahn.wamp.interfaces.ISession()*

class `autobahn.twisted.wamp.`**`ApplicationRunner`**(*url*, *realm=None*, *extra=None*, *serializers=None*, *ssl=None*, *proxy=None*, *headers=None*)
This class is a convenience tool mainly for development and quick hosting of WAMP application components.

It can host a WAMP application component in a WAMP-over-WebSocket client connecting to a WAMP router.

Parameters

- **`url`** (*str*) – The WebSocket URL of the WAMP router to connect to (e.g. *ws://somehost.com:8090/somepath*)
- **`realm`** (*str*) – The WAMP realm to join the application session to.
- **`extra`** (*dict*) – Optional extra configuration to forward to the application component.
- **`serializers`** (*list*) – A list of WAMP serializers to use (or None for default serializers). Serializers must implement *autobahn.wamp.interfaces.ISerializer*.
- **`ssl`** (`twisted.internet.ssl.CertificateOptions`) – (Optional). If specified this should be an instance suitable to pass as `sslContextFactory` to `twisted.internet.endpoints.SSL4ClientEndpoint` ` such as `twisted.internet.ssl.CertificateOptions`. Leaving it as `None` will use the result of calling Twisted's `twisted.internet.ssl.platformTrust()` which tries to use your distribution's CA certificates.
- **`proxy`** (*dict or None*) – Explicit proxy server to use; a dict with `host` and `port` keys
- **`headers`** (*dict*) – Additional headers to send (only applies to WAMP-over-WebSocket).

`stop`()
Stop reconnecting, if auto-reconnecting was enabled.

`run`(*make*, *start_reactor=True*, *auto_reconnect=False*, *log_level='info'*)
Run the application component.

Parameters

- **`make`** (*callable*) – A factory that produces instances of *autobahn.twisted.wamp.ApplicationSession* when called with an instance of *autobahn.wamp.types.ComponentConfig*.
- **`start_reactor`** – When `True` (the default) this method starts the Twisted reactor and doesn't return until the reactor stops. If there are any problems starting the reactor or connect()-ing, we stop the reactor and raise the exception back to the caller.

Returns None is returned, unless you specify `start_reactor=False` in which case the Deferred that connect() returns is returned; this will callback() with an IProtocol instance, which will actually be an instance of `WampWebSocketClientProtocol`

Module `autobahn.asyncio`

Autobahn asyncio specific classes. These are used when asyncio is run as the underlying networking framework.

WebSocket Protocols and Factories

Classes for WebSocket clients and servers using asyncio.

class `autobahn.asyncio.websocket.`**`WebSocketServerProtocol`**
Base class for asyncio-based WebSocket server protocols.

Implements:

- *autobahn.websocket.interfaces.IWebSocketChannel*

class `autobahn.asyncio.websocket.`**`WebSocketClientProtocol`**
Base class for asyncio-based WebSocket client protocols.

Implements:

- *autobahn.websocket.interfaces.IWebSocketChannel*

class `autobahn.asyncio.websocket.`**`WebSocketServerFactory`**(**args*, ***kwargs*)
Base class for asyncio-based WebSocket server factories.

Implements:

- *autobahn.websocket.interfaces.IWebSocketServerChannelFactory*

Note: In addition to all arguments to the constructor of *autobahn.websocket.interfaces.IWebSocketServerChannelFactory()*, you can supply a `loop` keyword argument to specify the asyncio event loop to be used.

`protocol`
alias of `WebSocketServerProtocol`

class `autobahn.asyncio.websocket.`**`WebSocketClientFactory`**(**args*, ***kwargs*)
Base class for asyncio-based WebSocket client factories.

Implements:

- *autobahn.websocket.interfaces.IWebSocketClientChannelFactory*

Note: In addition to all arguments to the constructor of *autobahn.websocket.interfaces.IWebSocketClientChannelFactory()*, you can supply a `loop` keyword argument to specify the asyncio event loop to be used.

WAMP-over-WebSocket Protocols and Factories

Classes for WAMP-WebSocket clients and servers using asyncio.

class autobahn.asyncio.websocket.**WampWebSocketServerProtocol**
asyncio-based WAMP-over-WebSocket server protocol.

Implements:

- *autobahn.wamp.interfaces.ITransport*

class autobahn.asyncio.websocket.**WampWebSocketClientProtocol**
asyncio-based WAMP-over-WebSocket client protocols.

Implements:

- *autobahn.wamp.interfaces.ITransport*

class autobahn.asyncio.websocket.**WampWebSocketServerFactory**(*factory*, **args*, ***kwargs*)
asyncio-based WAMP-over-WebSocket server factory.

Parameters

- **factory** (*callable*) – A callable that produces instances that implement *autobahn.wamp.interfaces.ITransportHandler*
- **serializers** (list of objects implementing *autobahn.wamp.interfaces.ISerializer*) – A list of WAMP serializers to use (or None for all available serializers).

protocol
alias of WampWebSocketServerProtocol

class autobahn.asyncio.websocket.**WampWebSocketClientFactory**(*factory*, **args*, ***kwargs*)
asyncio-based WAMP-over-WebSocket client factory.

Parameters

- **factory** (*callable*) – A callable that produces instances that implement *autobahn.wamp.interfaces.ITransportHandler*
- **serializer** (object implementing *autobahn.wamp.interfaces.ISerializer*) – The WAMP serializer to use (or None for "best" serializer, chosen as the first serializer available from this list: CBOR, MessagePack, UBJSON, JSON).

protocol
alias of WampWebSocketClientProtocol

WAMP-over-RawSocket Protocols and Factories

Classes for WAMP-RawSocket clients and servers using asyncio.

class autobahn.asyncio.rawsocket.**WampRawSocketServerProtocol**
asyncio-based WAMP-over-RawSocket server protocol.

Implements:

- *autobahn.wamp.interfaces.ITransport*

class autobahn.asyncio.rawsocket.**WampRawSocketClientProtocol**
asyncio-based WAMP-over-RawSocket client protocol.

Implements:

- *autobahn.wamp.interfaces.ITransport*

class autobahn.asyncio.rawsocket.**WampRawSocketServerFactory**(*factory*, *serializers=None*)

asyncio-based WAMP-over-RawSocket server protocol factory.

Parameters

- **factory** (*callable*) – A callable that produces instances that implement *autobahn.wamp.interfaces.ITransportHandler*
- **serializers** (list of objects implementing *autobahn.wamp.interfaces.ISerializer*) – A list of WAMP serializers to use (or None for all available serializers).

protocol

alias of WampRawSocketServerProtocol

class autobahn.asyncio.rawsocket.**WampRawSocketClientFactory**(*factory*, *serializer=None*)

asyncio-based WAMP-over-RawSocket client factory.

Parameters

- **factory** (*callable*) – A callable that produces instances that implement *autobahn.wamp.interfaces.ITransportHandler*
- **serializer** (object implementing *autobahn.wamp.interfaces.ISerializer*) – The WAMP serializer to use (or None for "best" serializer, chosen as the first serializer available from this list: CBOR, MessagePack, UBJSON, JSON).

protocol

alias of WampRawSocketClientProtocol

WAMP Sessions

Classes for WAMP sessions using asyncio.

class autobahn.asyncio.wamp.**ApplicationSession**(*config=None*)

WAMP application session for asyncio-based applications.

Implements:

- *autobahn.wamp.interfaces.ITransportHandler*
- *autobahn.wamp.interfaces.ISession*

Implements *autobahn.wamp.interfaces.ISession()*

class autobahn.asyncio.wamp.**ApplicationRunner**(*url*, *realm=None*, *extra=None*, *serializers=None*, *ssl=None*, *proxy=None*, *headers=None*)

This class is a convenience tool mainly for development and quick hosting of WAMP application components.

It can host a WAMP application component in a WAMP-over-WebSocket client connecting to a WAMP router.

Parameters

- **url** (*str*) – The WebSocket URL of the WAMP router to connect to (e.g. *ws://somehost.com:8090/somepath*)
- **realm** (*str*) – The WAMP realm to join the application session to.
- **extra** (*dict*) – Optional extra configuration to forward to the application component.

- **serializers** (*list*) – A list of WAMP serializers to use (or None for default serializers). Serializers must implement *autobahn.wamp.interfaces.ISerializer*.
- **ssl** (ssl.SSLContext or bool) – An (optional) SSL context instance or a bool. See the documentation for the *loop.create_connection* asyncio method, to which this value is passed as the ssl keyword parameter.
- **proxy** (*dict or None*) – Explicit proxy server to use; a dict with host and port keys
- **headers** (*dict*) – Additional headers to send (only applies to WAMP-over-WebSocket).

stop()
Stop reconnecting, if auto-reconnecting was enabled.

run(*make*, *start_loop=True*, *log_level='info'*)
Run the application component. Under the hood, this runs the event loop (unless *start_loop=False* is passed) so won't return until the program is done.

Parameters

- **make** (*callable*) – A factory that produces instances of *autobahn.asyncio.wamp.ApplicationSession* when called with an instance of *autobahn.wamp.types.ComponentConfig*.
- **start_loop** (*bool*) – When True (the default) this method start a new asyncio loop.

Returns None is returned, unless you specify *start_loop=False* in which case the coroutine from calling *loop.create_connection()* is returned. This will yield the (transport, protocol) pair.

CHAPTER 9

Changelog

17.5.1

Published 2017-05-01

- new: switched to calendar-based release/version numbering
- new: WAMP event retention example and docs
- new: WAMP subscribe/register options on WAMP decorators
- fix: require all TLS dependencies on extra_require_encryption setuptools
- new: support for X-Forwarded-For HTTP header
- fix: ABC interface definitions where missing “self”

0.18.2

Published 2017-04-14

- new: payload codec API
- fix: make WAMP-cryptobox use new payload codec API
- fix: automatic binary conversation for JSON
- new: improvements to experimental component API

0.18.1

Published 2017-03-28

- fix: errback all user handlers for all WAMP requests still outstanding when session/transport is closed/lost

- fix: allow WebSocketServerProtocol.onConnect to return a Future/Deferred
- new: allow configuration of RawSocket serializer
- new: test all examples on both WebSocket and RawSocket
- fix: revert to default arg for Deny reason
- new: WAMP-RawSocket and WebSocket default settings for asyncio
- new: experimental component based API and new WAMP Session class

0.18.0

Published 2017-03-26

- fix: big docs cleanup and polish
- fix: docs for publisher black-/whitelisting based on authid/authrole
- fix: serialization for publisher black-/whitelisting based on authid/authrole
- new: allow to stop auto-reconnecting for Twisted ApplicationRunner
- fix: allow empty realms (router decides) for asyncio ApplicationRunner

0.17.2

Published 2017-02-25

- new: WAMP-cryptosign elliptic curve based authentication support for asyncio
- new: CI testing on Twisted 17.1
- new: controller/shared attributes on ComponentConfig

0.17.1

Published 2016-12-29

- new: demo MQTT and WAMP clients interoperating via Crossbar.io
- new: WAMP message attributes for message resumption
- new: improvements to experimental WAMP components API
- fix: Python 3.4.4+ when using asyncio

0.17.0

Published 2016-11-30

- new: WAMP PubSub event retention
- new: WAMP PubSub last will / testament
- new: WAMP PubSub acknowledged delivery

- fix: WAMP Session lifecycle - properly handle asynchronous *ApplicationSession.onConnect* for asyncio

0.16.1

Published 2016-11-07

- fix: inconsistency between *PublishOptions* and *Publish* message
- new: improve logging with dropped connections (eg due to timeouts)
- fix: various smaller asyncio fixes
- new: rewrite all examples for new Python 3.5 async/await syntax
- fix: copyrights transferred from Tavendo GmbH to Crossbar.io Technologies GmbH

0.16.0

Published 2016-08-14

- new: new *autobahn.wamp.component* API in experimental stage
- new: Ed25519 OpenSSH and OpenBSD signify key support
- fix: allow Py2 and async user code in *onConnect* callback of asyncio

0.15.0

Published 2016-07-19

- new: WAMP AP option: register with maximum concurrency
- new: automatic reconnect for WAMP clients ApplicationRunner on Twisted
- new: RawSocket support in WAMP clients using ApplicationRunner on Twisted
- new: Set WebSocket production settings on WAMP clients using ApplicationRunner on Twisted
- fix: #715 Py2/Py3 issue with WebSocket traffic logging
- new: allow WAMP factories to take classes OR instances of ApplicationSession
- fix: make WebSocketResource working on Twisted 16.3
- fix: remove some minified AutobahnJS from examples (makes distro packagers happy)
- new: WAMP-RawSocket transport for asyncio
- fix: #691 (**security**) If the *allowedOrigins* websocket option was set, the resulting matching was insufficient and would allow more origins than intended

0.14.1

Published 2016-05-26

- fix: unpinned Twisted version again

- fix: remove X-Powered-By header
- fix: removed decrecated args to ApplicationRunner

0.14.0

Published 2016-05-01

- new: use of batched/chunked timers to massively reduce CPU load with WebSocket auto-ping/pong
- new: support new UBJSON WAMP serialization format
- new: publish universal wheels
- fix: replaced *msgpack-python* with *u-msgpack-python*
- fix: some glitches with *eligible / exlude* when used with *authid / authrole*
- fix: some logging glitches
- fix: pin Twisted at 16.1.1 (for now)

0.13.1

Published 2016-04-09

- moved helper funs for WebSocket URL handling to `autobahn.websocket.util`
- fix: marshal WAMP options only when needed
- fix: various smallish examples fixes

0.13.0

Published 2016-03-15

- fix: better traceback logging (#613)
- fix: unicode handling in debug messages (#606)
- fix: return Deferred from `run()` (#603).
- fix: more debug logging improvements
- fix: more *Pattern* tests, fix edge case (#592).
- fix: better logging from `asyncio` ApplicationRunner
- new: `disclose` becomes a strict router-side feature (#586).
- new: subscriber black/whitelisting using authid/authrole
- new: asyncio websocket testee
- new: refine Observable API (#593).

0.12.1

Published 2016-01-30

- new: support CBOR serialization in WAMP
- new: support WAMP payload transparency
- new: beta version of WAMP-cryptosign authentication method
- new: alpha version of WAMP-cryptobox end-to-end encryption
- new: support user provided authextra data in WAMP authentication
- new: support WAMP channel binding
- new: WAMP authentication util functions for TOTP
- fix: support skewed time leniency for TOTP
- fix: use the new logging system in WAMP implementation
- fix: some remaining Python 3 issues
- fix: allow WAMP prefix matching register/subscribe with dot at end of URI

0.11.0

Published 2015-12-09

0.10.9

Published 2015-09-15

- fixes regression #500 introduced with commit 9f68749

0.10.8

Published 2015-09-13

- maintenance release with some issues fixed

0.10.7

Published 2015-09-06

- fixes a regression in 0.10.6

0.10.6

Published 2015-09-05

- maintenance release with nearly two dozen fixes
- improved Python 3, error logging, WAMP connection mgmt, ..

0.10.5

Published 2015-08-06

- maintenance release with lots of smaller bug fixes

0.10.4

Published 2015-05-08

- maintenance release with some smaller bug fixes

0.10.3

Published 2015-04-14

- new: using txaio package
- new: revised WAMP-over-RawSocket specification implemented
- fix: ignore unknown attributes in WAMP Options/Details

0.10.2

Published 2015-03-19

- fix: Twisted 11 lacks IPv6 address class
- new: various improvements handling errors from user code
- new: add parameter to limit max connections on WebSocket servers
- new: use new-style classes everywhere
- new: moved package content to repo root
- new: implement router revocation signaling for registrations/subscriptions
- new: a whole bunch of more unit tests / coverage
- new: provide reason/message when transport is lost
- fix: send WAMP errors upon serialization errors

0.10.1

Published 2015-03-01

- support for pattern-based subscriptions and registrations
- support for shared registrations
- fix: HEARTBEAT removed

0.10.0

Published 2015-02-19

- Change license from Apache 2.0 to MIT
- fix file line endings
- add setuptools test target
- fix Python 2.6

0.9.6

Published 2015-02-13

- PEP8 code conformance
- PyFlakes code quality
- fix: warning for xrange on Python 3
- fix: parsing of IPv6 host headers
- add WAMP/Twisted service
- fix: handle connect error in ApplicationRunner (on Twisted)

0.9.5

Published 2015-01-11

- do not try to fire onClose on a session that never existed in the first place (fixes #316)
- various doc fixes
- fix URI decorator component handling (PR #309)
- fix “standalone” argument to ApplicationRunner

0.9.4

Published 2014-12-15

- refactor router code to Crossbar.io

- fix: catch error when Nagle cannot be set on stream transport (UDS)
- fix: spelling in doc strings / docs
- fix: WAMP JSON serialization of Unicode for ujson
- fix: Twisted plugins issue

0.9.3-2

Published 2014-11-15

- maintenance release with some smaller bug fixes
- use ujson for WAMP when available
- reduce WAMP ID space to [0, 2**31-1]
- deactivate Twisted plugin cache recaching in *setup.py*

0.9.3

Published 2014-11-10

- feature: WebSocket origin checking
- feature: allow to disclose caller transport level info
- fix: Python 2.6 compatibility
- fix: handling of WebSocket close frame in a corner-case

0.9.2

Published 2014-10-17

- fix: permessage-deflate "client_max_window_bits" parameter handling
- fix: cancel opening handshake timeouts also for WebSocket clients
- feature: add more control parameters to Flash policy file factory
- feature: update AutobahnJS in examples
- feature: allow to set WebSocket HTTP headers via dict
- fix: ayncio imports for Python 3.4.2
- feature: added reconnecting WebSocket client example

0.9.1

Published 2014-09-22

- maintenance release with some smaller bug fixes

0.9.0

Published 2014-09-02

- all WAMP v1 code removed
- migrated various WAMP examples to WAMP v2
- improved unicode/bytes handling
- lots of code quality polishment
- more unit test coverage

0.8.15

Published 2014-08-23

- docs polishing
- small fixes (unicode handling and such)

0.8.14

Published 2014-08-14

- add automatic WebSocket ping/pong (#24)
- WAMP-CRA client side (beta!)

0.8.13

Published 2014-08-05

- fix Application class (#240)
- support WSS for Application class
- remove implicit dependency on bzip2 (#244)

0.8.12

Published 2014-07-23

- WAMP application payload validation hooks
- added Tox based testing for multiple platforms
- code quality fixes

0.8.11

Published

- hooks and infrastructure for WAMP2 authorization
- new examples: Twisted Klein, Crochet, wxPython
- improved WAMP long-poll transport
- improved stats tracker

0.8.10

Published

- WAMP-over-Long-poll (preliminary)
- WAMP Authentication methods CR, Ticket, TOTP (preliminary)
- WAMP App object (preliminary)
- various fixes

0.8.9

Published

- maintenance release

0.8.8

Published

- initial support for WAMP on asyncio
- new WAMP examples
- WAMP ApplicationRunner

0.8.7

Published

- maintenance release

0.8.6

Published

- started reworking docs

- allow factories to operate without WS URL
- fix behavior on second protocol violation

0.8.5

Published

- support WAMP endpoint/handler decorators
- new examples for endpoint/handler decorators
- fix excludeMe pubsub option

0.8.4

Published

- initial support for WAMP v2 authentication
- various fixes/improvements to WAMP v2 implementation
- new example: WebSocket authentication with Mozilla Persona
- polish up documentation

0.8.3

Published

- fix bug with closing router app sessions

0.8.2

Published

- compatibility with latest WAMP v2 spec ("RC-2, 2014/02/22")
- various smaller fixes

0.8.1

Published

- WAMP v2 basic router (broker + dealer) implementation
- WAMP v2 example set
- WAMP v2: decouple transports, sessions and routers
- support explicit (binary) subprotocol name for wrapping WebSocket factory
- fix dependency on MsgPack

0.8.0

Published

- new: complete WAMP v2 protocol implementation and API layer
- new: basic WAMP v2 router implementation
- existing WAMP v1 implementation renamed

0.7.4

Published

- fix WebSocket server HTML status page
- fix close reason string handling
- new “slowsquare” example
- Python 2.6 fixes

0.7.3

Published

- support asyncio on Python 2 (via “Trollius” backport)

0.7.2

Published

- really fix setup/packaging

0.7.1

Published

- setup fixes
- fixes for Python2.6

0.7.0

Published

- asyncio support
- Python 3 support
- support WebSocket (and WAMP) over Twisted stream endpoints

- support Twisted stream endpoints over WebSocket
- twistd stream endpoint forwarding plugin
- various new examples
- fix Flash policy factory

0.6.5

Published

- Twisted reactor is no longer imported on module level (but lazy)
- optimize pure Python UTF8 validator (10-20% speedup on PyPy)
- opening handshake traffic stats (per-open stats)
- add multi-core echo example
- fixes with examples of streaming mode
- fix zero payload in streaming mode

0.6.4

Published

- support latest *permessage-deflate* draft
- allow controlling memory level for *zlib* / *permessage-deflate*
- updated reference, moved docs to "Read the Docs"
- fixes #157 (a WAMP-CRA timing attack very, very unlikely to be exploitable, but anyway)

0.6.3

Published

- symmetric RPCs
- WebSocket compression: client and server, *permessage-deflate*, *permessage-bzip2* and *permessage-snappy*
- *onConnect* is allowed to return Deferreds now
- custom publication and subscription handler are allowed to return Deferreds now
- support for explicit proxies
- default protocol version now is RFC6455
- option to use salted passwords for authentication with WAMP-CRA
- automatically use *ultrajson* acceleration package for JSON processing when available
- automatically use *wsaccel* acceleration package for WebSocket masking and UTF8 validation when available
- allow setting and getting of custom HTTP headers in WebSocket opening handshake
- various new code examples

- various documentation fixes and improvements

0.5.14

Published

- base version when we started to maintain a changelog

Python Module Index

a

Index

S

T

U

W

X