autobahn

Release 20.2.1

Feb 19, 2020

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)

for Python 3.5+ running on Twisted **or** asyncio.

Note: **Autobahn|Python** up to version v19.11.2 also supported Python 2 and 3.4+.

1.1 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
- 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 ;)

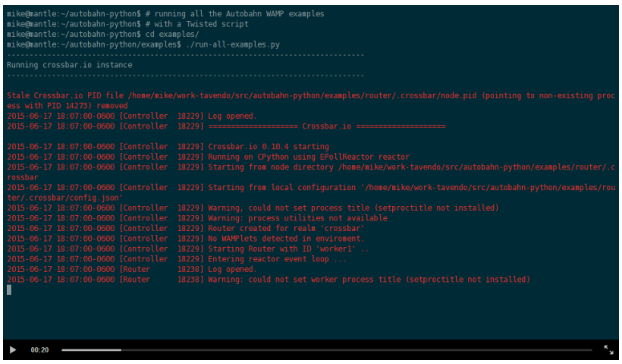
For (hopefully) current test reports from the Testsuite see

- 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.

1.2 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?

1.3 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.component import Component
# or: from autobahn.asyncio.component import Component

demo = Component(
```

(continues on next page)

(continued from previous page)

```
    transports=[u"wss://demo.crossbar.io/ws"],
)

# 1. subscribe to a topic
@demo.subscribe(u'com.myapp.hello')
def hello(msg):
    print("Got hello: {}".format(msg))

# 2. register a procedure for remote calling
@demo.register(u'com.myapp.add2')
def add2(x, y):
    return x + y

# 3. after we've authenticated, run some code
@demo.on_join
async def joined(session, details):
    # publish an event (won't go to "this" session by default)
    await session.publish('com.myapp.hello', 'Hello, world!')

    # 4. call a remote procedure
    result = await session.call('com.myapp.add2', 2, 3)
    print("com.myapp.add2(2, 3) = {}".format(result))

if __name__ == "__main__":
    run([demo])
```

Complete example code:

- Twisted Example
- asyncio Example

Introduction to WAMP Programming with Autobahn:

- *WAMP Programming*

1.4 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.

1.5 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.

1.6 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.

1.7 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

1.8 Sitemap

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

CHAPTER 2

Installation

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

2.1 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 3.5+.

2.1.1 Supported Configurations

Here are the configurations supported by Autobahn:

Python	Twisted	asyncio	Notes
CPython 3.5+	yes	yes	asyncio in the standard library
PyPy 3	yes	yes	asyncio in the standard library

2.1.2 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.

2.2 Installing Autobahn

2.2.1 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 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 CPython or PyPy.

There are the flavors which are based on the official CPython and PyPy images, plus 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.)

2.2.2 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]
```

2.2.3 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]
```

2.2.4 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]
```

2.2.5 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 3.7 32-Bit
2. Add `C:\Python37;C:\Python37\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]`

2.3 Check the Installation

To check the installation, fire up the Python and run

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

2.4 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

3.1 Introduction

3.1.1 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 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.

3.1.2 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

3.1.3 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.

3.2 Resources

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

3.2.1 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.

3.2.2 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:

3.3 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.

3.3.1 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.

3.3.2 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)
```

(continues on next page)

(continued from previous page)

```
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*

4.1 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.

4.1.1 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.

4.1.2 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.

4.1.3 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.

4.1.4 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__':

   import 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)

4.2 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.onConnecting()`
- `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 four callbacks above each only fires once for a given connection.

4.2.1 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))
```

For a WebSocket client protocol, `onConnecting()` is called immediately before the handshake to the server starts. It is called with some details about the underlying transport. This may return `None` (the default) to get default values for several options (which are gotten from the Factory) or it may return a `autobahn.websocket.types.ConnectingRequest` instance to indicate options for this handshake. This allows using different options on each request (as opposed to using a static set of options in the Factory).

Then, once the server has responded, a WebSocket client protocol will fire `onConnect()` with a `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`.

4.2.2 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.")
```

4.2.3 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.

4.2.4 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.

4.3 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*.

4.3.1 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*

4.3.2 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__':
   import 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)

4.4 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.

4.4.1 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)

4.4.2 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

4.4.3 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:

4.5 Upgrading

4.5.1 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.

5.1 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 routed 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.

5.1.1 Creating Components

There are two ways to create components using Autobahn. One is based on deriving from a particular class and overriding methods and the other is based on functions and decorators. The latter is the recommended approach (but note that many examples and existing code use the subclassing approach). Both are fine and end up calling the same code under the hood.

For both approaches you get to decide if you prefer to use **Twisted** or **asyncio** and express this through `import`. This is `autobahn.twisted.*` versus `autobahn.asyncio.*`

When using **Twisted** you import from `autobahn.twisted.component`:

```
from autobahn.twisted.component import Component

comp = Component(...)

@comp.on_join
def joined(session, details):
    print("session ready")
```

whereas when you are using **asyncio**:

```
from autobahn.asyncio.component import Component

comp = Component(...)

@comp.on_join
def joined(session, 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. For Twisted, you can use `@inlineCallbacks` or return `Deferred` from methods decorated with `on_join`; in Python 3 (with asyncio or Twisted) you would use coroutines (`async def`).

There are four "life cycle" events that Autobahn will trigger on your components: `connect`, `join`, `leave`, and `disconnect`. These all have corresponding decorators (or you can use code like `comp.on('join', the_callback)` if you prefer). We go over these events later.

5.1.2 Running Components

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

```
from autobahn.twisted.component import Component
from autobahn.twisted.component import run

comp = Component(
    transports=u"ws://localhost:8080/ws",
    realm=u"realm1",
)

@comp.on_join
def joined(session, details):
    print("session ready")

if __name__ == "__main__":
    run([comp])
```

and with **asyncio**:

```
from autobahn.asyncio.component import Component
from autobahn.asyncio.component import run

comp = Component(
    transports=u"ws://localhost:8080/ws",
    realm=u"realm1",
)

@comp.on_join
async def joined(session, details):
    print("session ready")

if __name__ == "__main__":
    run([comp])
```

As can be seen, the only difference between Twisted and asyncio is the import (line 1 and 2). The rest of the code is identical.

The configuration of the component is specified when you construct it; the above is the bare minimum – you can specify many transports (which will be tried and re-tried in order) as well as authentication options, the realm to join, re-connection parameters, etcetera. See *Component Configuration Options* for details. A single Python program can run many different `Component` instances at once and you can interconnect these as you see fit – so a single program can have multiple WAMP connections (e.g. to different Realms or different WAMP routers) at once.

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.

5.1.3 Running Subclass-Style Components

You can use the same "component" APIs to run a component based on subclassing *ApplicationSession*. In older code it's common to see `autobahn.twisted.wamp.ApplicationRunner` or `autobahn.asyncio.wamp.ApplicationRunner`. This runner lacks many of the options of the `autobahn.twisted.component.`

`run()` or `autobahn.asyncio.component.run()` functions, so although it can still be useful you likely want to upgrade to `run()`.

All you need to do is set the *session_factory* of a `autobahn.twisted.component.Component` instance to your `autobahn.twisted.wamp.ApplicationSession` subclass (or pass it as a `kwarg` when creating the `Component`)

```
comp = Component(
    session_factory=MyApplicationSession,
)
```

5.1.4 Patterns for More Complicated Applications

Many of the examples in this documentation use a decorator style with fixed, static WAMP URIs for registrations and subscriptions. If you have a more complex application, you might want to create URIs at run-time or link several `Component` instances together.

It is important to remember that `Component` handles re-connection – this implies there are times when your component is **not** connected. The *on_join* handlers are run whenever a fresh WAMP session is started, so this is the appropriate way to hook in "initialization"-style code (*on_leave* is where "un-initialization" code goes). Note that each new WAMP session will use a new instance of `ApplicationSession`.

Here's a slightly more complex example that is a small Klein Web application that publishes to a WAMP session when a certian URL is requested (note that the Crossbario.io router supports various REST-style integrations already). Using a similar pattern, you could tie together two or more `Component` instances (even connecting to two or more *different* WAMP routers).

```
from autobahn.twisted.component import Component
from twisted.internet.defer import inlineCallbacks, Deferred
from twisted.internet.endpoints import TCP4ServerEndpoint
from twisted.web.server import Site
from twisted.internet.task import react


# pip install klein
from klein import Klein


class WebApplication(object):
    """
    A simple Web application that publishes an event every time the
    url "/" is visited.
    """
    def __init__(self, app, wamp_comp):
        self._app = app
        self._wamp = wamp_comp
        self._session = None  # "None" while we're disconnected from WAMP router

        # associate ourselves with WAMP session lifecycle
        self._wamp.on('join', self._initialize)
        self._wamp.on('leave', self._uninitialize)
        # hook up Klein routes
        self._app.route("/", branch=True)(self._render_slash)

    def _initialize(self, session, details):
        print("Connected to WAMP router")
```

(continues on next page)

(continued from previous page)

```
        self._session = session

    def _uninitialize(self, session, reason):
        print(session, reason)
        print("Lost WAMP connection")
        self._session = None

    def _render_slash(self, request):
        if self._session is None:
            request.setResponseCode(500)
            return b"No WAMP session\n"
        self._session.publish("com.myapp.request_served")
        return b"Published to 'com.myapp.request_served'\n"


@inlineCallbacks
def main(reactor):
    component = Component(
        transports="ws://localhost:8080/ws",
        realm="crossbardemo",
    )
    app = Klein()
    webapp = WebApplication(app, component)

    # have our Web site listen on 8090
    site = Site(app.resource())
    server_ep = TCP4ServerEndpoint(reactor, 8090)
    port = yield server_ep.listen(site)
    print("Web application on {}".format(port))

    # we don't *have* to hand over control of the reactor to
    # component.run -- if we don't want to, we call .start()
    # The Deferred it returns fires when the component is "completed"
    # (or errbacks on any problems).
    comp_d = component.start(reactor)

    # When not using run() we also must start logging ourselves.
    import txaio
    txaio.start_logging(level='info')

    # If the Component raises an exception we want to exit. Note that
    # things like failing to connect will be swallowed by the
    # re-connection mechanisms already so won't reach here.

    def _failed(f):
        print("Component failed: {}".format(f))
        done.errback(f)
    comp_d.addErrback(_failed)

    # wait forever (unless the Component raises an error)
    done = Deferred()
    yield done


if __name__ == '__main__':
    react(main)
```

5.1.5 Longer Example

Here is a more-complete example showing some of the options you can pass when setting up a *Component*. This example can be run against the Crossbar.io router configuration that comes with Autobahn – just run *crossbar start* in *examples/router/* in your clone.

Twisted:

```
from autobahn.twisted.component import Component, run
from autobahn.twisted.util import sleep
from autobahn.wamp.types import RegisterOptions
from twisted.internet.defer import inlineCallbacks, returnValue

# to see how this works on the Crossbar.io side, see the example
# router configuration in:
# https://github.com/crossbario/autobahn-python/blob/master/examples/router/.crossbar/
↪config.json

component = Component(
    # you can configure multiple transports; here we use two different
    # transports which both exist in the demo router
    transports=[
        {
            "type": "websocket",
            "url": "ws://localhost:8080/auth_ws",
            "endpoint": {
                "type": "tcp",
                "host": "localhost",
                "port": 8080,
            },
            # you can set various websocket options here if you want
            "options": {
                "open_handshake_timeout": 100,
            }
        },
    ],
    # authentication can also be configured (this will only work on
    # the demo router on the first transport above)
    authentication={
        "cryptosign": {
            'authid': 'alice',
            # this key should be loaded from disk, database etc never burned into␣
↪code like this...
            'privkey':
↪'6e3a302aa67d55ffc2059efeb5cf679470b37a26ae9ac18693b56ea3d0cd331c',
        }
    },
    # must provide a realm
    realm="crossbardemo",
)


@component.on_join
@inlineCallbacks
def join(session, details):
    print("joined {}: {}".format(session, details))
    yield sleep(1)
    print("Calling 'com.example'")
```

(continues on next page)

(continued from previous page)

```
    res = yield session.call("example.foo", 42, something="nothing")
    print("Result: {}".format(res))
    yield session.leave()


@component.register(
    "example.foo",
    options=RegisterOptions(details_arg='details'),
)
@inlineCallbacks
def foo(*args, **kw):
    print("foo called: {}, {}".format(args, kw))
    for x in range(5, 0, -1):
        print("  returning in {}".format(x))
        yield sleep(1)
    print("returning '42'")
    returnValue(42)


if __name__ == "__main__":
    run([component])
```

The Python3 / asyncio version of the same example is nearly identical except for some imports (and the use of *async def* instead of Twisted's decorators):

asyncio:

```
from autobahn.asyncio.component import Component, run
from asyncio import sleep
from autobahn.wamp.types import RegisterOptions

# to see how this works on the Crossbar.io side, see the example
# router configuration in:
# https://github.com/crossbario/autobahn-python/blob/master/examples/router/.crossbar/
↪config.json

component = Component(
    # you can configure multiple transports; here we use two different
    # transports which both exist in the demo router
    transports=[
        {
            "type": "websocket",
            "url": "ws://localhost:8080/auth_ws",
            "endpoint": {
                "type": "tcp",
                "host": "localhost",
                "port": 8080,
            },
            # you can set various websocket options here if you want
            "options": {
                "open_handshake_timeout": 100,
            }
        },
    ],
    # authentication can also be configured (this will only work on
    # the demo router on the first transport above)
    authentication={
```

(continues on next page)

(continued from previous page)

```
        "cryptosign": {
            'authid': 'alice',
            # this key should be loaded from disk, database etc never burned into␣
↪code like this...
            'privkey':
↪'6e3a302aa67d55ffc2059efeb5cf679470b37a26ae9ac18693b56ea3d0cd331c',
        }
    },
    # must provide a realm
    realm="crossbardemo",
)


@component.on_join
async def join(session, details):
    print("joined {}: {}".format(session, details))
    await sleep(1)
    print("Calling 'com.example'")
    res = await session.call("example.foo", 42, something="nothing")
    print("Result: {}".format(res))
    await session.leave()


@component.register(
    "example.foo",
    options=RegisterOptions(details_arg='details'),
)
async def foo(*args, **kw):
    print("foo called: {}, {}".format(args, kw))
    for x in range(5, 0, -1):
        print("  returning in {}".format(x))
        await sleep(1)
    print("returning '42'")
    return 42


if __name__ == "__main__":
    run([component])
```

5.2 Component Configuration Options

Most of the arguments given when creating a new `Component` are a series of `dict` instances containing "configuration"-style information. These are documented in `autobahn.wamp.component.Component` so we go through the most important ones here:

5.2.1 transports=

You may define any number of transports; these are tried in round-robin order when doing connections (and subsequent re-connections). If the `is_fatal=` predicate is used and returns `True` for any errors, that transport won't be used any more (and when no transports remain, the `Component` has "failed").

Each transport is defined similarly to "connecting transports" in Crossbar.io but as a simplification a plain unicode URI may be used, for example `transports=u"ws://example.com/ws"` or `transports=[u"ws:/`

/example.com/ws"]. If using a dict instead of a string you can specify the following keys:

- type: "websocket" (default) or "rawsocket"
- url: the URL of the router to connect to (very often, this will be the same as the "endpoint" host but not always)
- endpoint: (optional; can be inferred from above) - type: "tcp" or "unix" - host, port: only for type="tcp" - path: only for type="unix" - tls: bool (advanced Twisted users can pass CertificateOptions); this is also inferred from a wss: scheme.

In addition, each transport may have some options related to re-connections:

- max_retries: (default -1, "try forever") or a hard limit.
- max_retry_delay: (default 300)
- initial_retry_delay: (default 1.5) how long we wait to re-connect the first time
- retry_delay_growth: (default 1.5) a multiplier expanding our delay each try (so the second re-connect we wait retry_delay_growth * initial_retry_delay seconds).
- retry_delay_jitter: (default 0.1) percent of total retry delay to add/subtract as jitter

After a successful connection, all re-connection values are set back to their original values.

5.2.2 realm=

Each WAMP Session is associated with precisely one realm, and so is each *Component*. A "realm" is a logically separated WAMP URI space (and is isolated from all other realms that may exist on a WAMP router). You must pass a unicode string here.

5.2.3 session_factory=

Leaving this as None should be fine for most users. You may pass an ApplicationSession subclass here (or even a callable that takes a single config argument and returns an instance implementing IApplicationSession) to create new session objects. This can be used by users of the "subclass"-style API who still want to take advantage of the configuration of Component and run(). The session argument passed in many of the callbacks will be an instance of this (see also *Session Lifecycle*).

5.2.4 authentication=

This contains a dict mapping an authenticator name to its configuration. You do not have to have any authentication information, in which case anonymous will be used. Currently valid authenticators are: anonymous, ticket, wampcra, cryptosign (experimental) and scram (experimental).

Typically the administrator of your WAMP router will decide which authentication methods are allowed. See for example Crossbar.io's authentication documentation for some discussion of the various methods.

anonymous accepts no options. Most methods accept options for:

- authextra: application-specific information
- authid: unicode username
- authrole: the desired role inside the realm

The other authentication methods take additional options as indicated below:

- **wampcra**: also accepts secret (the password)

- **cryptosign** (experimental): also accepts `privkey`, the hex-encoded ed25519 private key
- **scram** (experimental): also requires `nonce` (hex-encoded), `kdf` (`"argon2id-13"` or `"pbkdf2"`), `salt` (hex-encoded), `iterations` (integer) and optionally `memory` (integer) and `channel_binding` (currently ignored).
- **ticket**: accepts only the `ticket` option

5.3 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 `crossbardemo`.

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, run the example configuration from `examples/router` in your Autobahn clone. If you want to start fresh, you can instead do this:

```
crossbar init
```

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

```
crossbar start
```

Note: The defaults in the above will not work with the examples in the repository nor this documentation; please use the example router configuration that ships with Autobahn (in `examples/router/.crossbar/`).

5.4 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*.

5.4.1 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**; note that we've eliminated the configuration of the `Component` for clarity; see above for full example.

```
1  from autobahn.twisted.component import Component, run
2
3  component = Component(...)
4
5  @component.on_join
6  @inlineCallbacks
7  def joined(session, details):
8      print("session ready")
9
10     def add2(x, y):
11         return x + y
12
13     try:
14         yield session.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 `on_join` callback which fires when the session has connected to a *Router* and joined a *Realm*. Another way to arrange for procedures to be registered is with the `@register` decorator:

```
1  from autobahn.twisted.component import Component, run
2
3  component = Component(...)
4
5  @component.register
6  def add2(x, y):
7      return x + y
```

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 identical except for the imports (note that `add` could be `async def` here if it needed to do other work).

```
1  from autobahn.asyncio.component import Component, run
2
3  component = Component(...)
4
5  @component.register
6  def add2(x, y):
7      return x + y
```

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`

5.4.2 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.component import Component, run
from twisted.internet.defer import inlineCallbacks


component = Component(...)

@component.on_join
@inlineCallbacks
def joined(session, details):
    print("session ready")
    try:
        res = yield session.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.component import Component, run


component = Component(...)

@component.on_join
async def joined(session, details):
    print("session ready")
    try:
        res = await session.call(u'com.myapp.add2', 2, 3)
        print("call result: {}".format(res))
    except Exception as e:
        print("call error: {0}".format(e))
```

5.5 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*.

5.5.1 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:

```
1  from autobahn.twisted.component import Component
2  from twisted.internet.defer import inlineCallbacks
3
4
5  component = Component(...)
6
7  @component.on_join
8  @inlineCallbacks
9  def joined(session, details):
10     print("session ready")
11
12     def oncounter(count):
13         print("event received: {0}", count)
14
15     try:
16         yield session.subscribe(oncounter, u'com.myapp.oncounter')
17         print("subscribed to topic")
18     except Exception as e:
19         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

```
1  from autobahn.twisted.component import Component
2
3
4  component = Component(...)
5
6  @component.on_join
7  async def joined(session, details):
8      print("session ready")
9
10     def oncounter(count):
11         print("event received: {0}", count)
12
13     try:
14         yield session.subscribe(oncounter, u'com.myapp.oncounter')
15         print("subscribed to topic")
```

(continues on next page)

(continued from previous page)

```
    except Exception as e:
        print("could not subscribe to topic: {0}".format(e))
```

Again, nearly identical to Twisted. Note that when using the `Component` APIs we can use a shortcut to the above (e.g. perhaps there's nothing else to do in `on_join`). This shortcut works similarly for Twisted, so we only show an **asyncio** example:

```
from autobahn.twisted.component import Component


component = Component(...)

@component.subscribe(u"com.myapp.oncounter")
def oncounter(count):
    print("event received: {0}", count)
```

5.5.2 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.component import Component
from autobahn.twisted.util import sleep
from twisted.internet.defer import inlineCallbacks


component = Component(...)


@component.on_join
@inlineCallbacks
def joined(session, details):
    print("session ready")

    counter = 0
    while True:
        # publish() only returns a Deferred if we asked for an acknowledgement
        session.publish(u'com.myapp.oncounter', counter)
        counter += 1
        yield sleep(1)
```

The corresponding **asyncio** code looks like this

```
from autobahn.asyncio.component import Component
from asyncio import sleep


component = Component(...)


@component.on_join
```

(continues on next page)

(continued from previous page)

```
async def joined(session, details):
    print("session ready")

    counter = 0
    while True:
        # publish() is only async if we asked for an acknowledgement
        session.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) and causes `publish()` to return a Future/Deferred.

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`.

5.6 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)

In the `Component` API, there are similar corresponding events. The biggest difference is the lack of "challenge" events (you pass authentication configuration instead) and the addition of a "ready" event. You can subscribe to these events directly using a "listener" style API or via decorators. The events are:

1. "connect": transport connected
2. "join": session has successfully joined a realm
3. "ready": indicates that the realm has been joined **and** all "join" handlers have completed (including async ones)
4. "leave": session has left a realm

5. "disconnect": transport has disconnected

You can use the method `autobahn.wamp.component.Component.on()` to subscribe directly to events with a listener-function. For example, `component.on('ready', my_ready_listener)`. Note that on a single `Component` instance these callbacks *can* happen multiple times (e.g. if the component is disconnected and then reconnects, its `connect` message will fire again after the `disconnect`). However, they will always be in order (i.e. you can't `join` until after a `connect` and `ready` always comes after `join`).

There is also still the older "subclassing" based API, which is still supported and can be used if you prefer. This API involves subclassing `ApplicationSession` and overriding methods corresponding to the events (see `ISession` for more information):

```
class CustomSession(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")
```

5.7 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're using the high-level `autobahn.twisted.component.run()` or `autobahn.asyncio.component.run()` APIs, logging will be started for you.

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.

5.8 Upgrading

5.8.1 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.

5.8.2 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

Asyncio REPL

```
pip install autobahn[asyncio,serialization,encryption]
python -m asyncio
```

```
import txaio
txaio.use_asyncio()

from autobahn.wamp.types import SubscribeOptions, PublishOptions
from autobahn.asyncio.component import Component, run
from autobahn.asyncio.util import sleep

comp = Component(
    transports='ws://localhost:8080/ws',
    realm='realm1'
)

counter = 0

async def on_event(details=None):
    global counter
    print('Event received', details)
    counter += 1

@comp.on_join
async def joined(session, details):
    global counter
    print('Session joined', details)
    sub = await session.subscribe(on_event, "io.crossbar.example",
                                  options=SubscribeOptions(details=True))
    print('Session subscribed', sub)
    while counter < 10:
        pub = await session.publish("io.crossbar.example",
                                    options=PublishOptions(acknowledge=True, exclude_
↪me=False))
        print('Event published', pub)
```

(continues on next page)

(continued from previous page)

```
        await sleep(1)
    session.leave()

run([comp], start_loop=False)
```

```
(cpy380_1) oberstet@intel-nuci7:~/scm$ python -m asyncio
asyncio REPL 3.8.0 (default, Oct 18 2019, 13:51:54)
[GCC 7.4.0] on linux
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more information.
>>> import asyncio
>>> import txaio
>>> txaio.use_asyncio()
>>>
>>> from autobahn.wamp.types import SubscribeOptions, PublishOptions
>>> from autobahn.asyncio.component import Component, run
>>> from autobahn.asyncio.util import sleep
>>>
>>> comp = Component(
...     transports='ws://localhost:8080/ws',
...     realm='realm1'
... )
>>>
>>> async def on_event(details=None):
...     print('Event received', details)
...
>>> @comp.on_join
... async def joined(session, details):
...     print('Session joined', details)
...     sub = await session.subscribe(on_event, "io.crossbar.example",␣
↪options=SubscribeOptions(details=True))
...     print('Session subscribed', sub)
...     while True:
...         pub = await session.publish("io.crossbar.example",␣
↪options=PublishOptions(acknowledge=True, exclude_me=False))
...         print('Event published', pub)
...         await sleep(1)
...
>>> run([comp], start_loop=False)
2019-10-23T09:51:39 connecting once using transport type "websocket" over endpoint
↪"tcp"
>>> Session joined
SessionDetails(realm=<realm1>,
            session=6184116312079408,
            authid=<F7FK-QQV7-F4W5-CNGX-QYHN-PLPV>,
            authrole=<anonymous>,
            authmethod=anonymous,
            authprovider=static,
            authextra={'x_cb_node_id': None, 'x_cb_peer': 'tcp4:127.0.0.1:27192', 'x_
↪cb_pid': 21008},
            serializer=<cbor>,
            resumed=None,
            resumable=None,
            resume_token=None)
Session subscribed Subscription(id=5518068544367384, is_active=True)
Event published Publication(id=4151012151142491, was_encrypted=False)
```

(continues on next page)

(continued from previous page)

```
Event received EventDetails(subscription=Subscription(id=5518068544367384, is_
↪active=True), publication=4151012151142491, publisher=None, publisher_authid=None,␣
↪publisher_authrole=None, topic=<io.crossbar.example>, retained=None, enc_algo=None,␣
↪forward_for=None)
Event published Publication(id=3676179573074954, was_encrypted=False)
Event received EventDetails(subscription=Subscription(id=5518068544367384, is_
↪active=True), publication=3676179573074954, publisher=None, publisher_authid=None,␣
↪publisher_authrole=None, topic=<io.crossbar.example>, retained=None, enc_algo=None,␣
↪forward_for=None)
Event published Publication(id=1831205249541796, was_encrypted=False)
Event received EventDetails(subscription=Subscription(id=5518068544367384, is_
↪active=True), publication=1831205249541796, publisher=None, publisher_authid=None,␣
↪publisher_authrole=None, topic=<io.crossbar.example>, retained=None, enc_algo=None,␣
↪forward_for=None)
Event published Publication(id=6028323371359219, was_encrypted=False)
Event received EventDetails(subscription=Subscription(id=5518068544367384, is_
↪active=True), publication=6028323371359219, publisher=None, publisher_authid=None,␣
↪publisher_authrole=None, topic=<io.crossbar.example>, retained=None, enc_algo=None,␣
↪forward_for=None)
Event published Publication(id=211622895505210, was_encrypted=False)
Event received EventDetails(subscription=Subscription(id=5518068544367384, is_
↪active=True), publication=211622895505210, publisher=None, publisher_authid=None,␣
↪publisher_authrole=None, topic=<io.crossbar.example>, retained=None, enc_algo=None,␣
↪forward_for=None)
Event published Publication(id=6235103334995396, was_encrypted=False)
Event received EventDetails(subscription=Subscription(id=5518068544367384, is_
↪active=True), publication=6235103334995396, publisher=None, publisher_authid=None,␣
↪publisher_authrole=None, topic=<io.crossbar.example>, retained=None, enc_algo=None,␣
↪forward_for=None)
Event published Publication(id=2482469817470784, was_encrypted=False)
Event received EventDetails(subscription=Subscription(id=5518068544367384, is_
↪active=True), publication=2482469817470784, publisher=None, publisher_authid=None,␣
↪publisher_authrole=None, topic=<io.crossbar.example>, retained=None, enc_algo=None,␣
↪forward_for=None)
2019-10-23T09:51:46 Shutting down due to SIGINT
(cpy380_1) oberstet@intel-nuci7:~/scm$
```

CHAPTER 7

XBR Programming

Autobahn comes with built-in support for XBR. This chapter contains documentation of writing XBR buyers and sellers in Python using Autobahn.

7.1 On-chain XBR smart contracts

The root anchor of the XBR network is a set of smart contracts on the Ethereum blockchain. The contracts are written in Solidity and published as open-source on GitHub.

To talk to the XBR smart contracts on the blockchain, you need two things:

1. the XBR smart contract addresses and
2. the XBR smart contract ABI files (JSON)

Both of which are built into the Autobahn library.

7.1.1 SimpleBlockchain

While you *can* use just the raw addresses and ABI blobs contained in Autobahn, and use whatever way and form to talk directly to the blockchain fully on your own, Autobahn *also* provides a convenient blockchain client with specific functions directly supporting XBR:

7.1.2 SimpleBlockchain Example

Here is a complete example blockchain client:

```
import argparse
from binascii import a2b_hex, b2a_hex
from autobahn import xbr
from twisted.internet.task import react
from twisted.internet.defer import inlineCallbacks


@inlineCallbacks
def main(reactor, gateway, adr):
    sbc = xbr.SimpleBlockchain(gateway)
    yield sbc.start()

    print('status for address 0x{}:'.format(b2a_hex(adr).decode()))

    # get ETH and XBR account balances for address
    balances = yield sbc.get_balances(adr)
    print('balances: {}'.format(balances))

    # get XBR network membership status for address
    member_status = yield sbc.get_member_status(adr)
    print('member status: {}'.format(member_status))


if __name__ == '__main__':
    parser = argparse.ArgumentParser()

    parser.add_argument('--gateway',
                        dest='gateway',
                        type=str,
                        default=None,
                        help='Ethereum HTTP gateway URL or None for auto-select.')

    parser.add_argument('--adr',
```

(continues on next page)

(continued from previous page)

```
                        dest='adr',
                        type=str,
                        default=None,
                        help='Ethereum address to lookup.')

    args = parser.parse_args()

    react(main, (args.gateway, a2b_hex(args.adr[2:],)))
```

Here is example output of above program with two different addresses, only one being a member, and with different balances of ETH and XBR.

```
connected to network 5777 at provider "http://localhost:1545"
status for address 0x90f8bf6a479f320ead074411a4b0e7944ea8c9c1:
balances: {'ETH': 999986687100000000000, 'XBR': 100000000000000000000000000}
member status: None

connected to network 5777 at provider "http://localhost:1545"
status for address 0xffcf8fdee72ac11b5c542428b35eef5769c409f0:
balances: {'ETH': 9999999999999999625429, 'XBR': 0}
member status: {'eula': 'QmU7Gizbre17x6V2VR1Q2GJEjz6m8S1bXmBtVxS2vmvb81', 'profile':␣
↪None}
```

7.1.3 Using the ABI files

To directly use the embedded ABI files:

```
import json
import pkg_resources
from pprint import pprint

with open(pkg_resources.resource_filename('xbr', 'contracts/XBRToken.json')) as f:
    data = json.loads(f.read())
    abi = data['abi']
    pprint(abi)
```

7.1.4 Data stored on-chain

See the XBRNetwork contract:

```
/// Current XBR Network members ("member directory").
mapping(address => Member) private members;

/// Current XBR Domains ("domain directory")
mapping(bytes16 => Domain) private domains;

/// Current XBR Nodes ("node directory");
mapping(bytes16 => Node) private nodes;

/// Index: node public key => (market ID, node ID)
mapping(bytes32 => bytes16) private nodesByKey;

/// Current XBR Markets ("market directory")
```

(continues on next page)

(continued from previous page)

```
mapping(bytes16 => Market) private markets;

/// Index: maker address => market ID
mapping(address => bytes16) private marketsByMaker;
```

7.2 Off-chain XBR market maker

7.2.1 SimpleBuyer

Autobahn includes a "simple buyer" for use in buyer delegate user services which is able to automatically quote and buy data encryption keys via the market maker from sellers in a market.

The simple buyer operates in the background and automatically buys keys on demand, as the user calls "unwrap(ciphertext)" on received XBR encrypted application payload:

7.2.2 SimpleBuyer Example

Here is a complete example buyer:

```
import binascii
import os

from autobahn.twisted.component import Component, run
from autobahn.xbr import SimpleBuyer
from autobahn.wamp.types import SubscribeOptions

comp = Component(
    transports=os.environ.get('XBR_INSTANCE', 'wss://continental2.crossbario.com/ws'),
    realm=os.environ.get('XBR_REALM', 'realm1'),
    extra={
        'market_maker_adr': os.environ.get('XBR_MARKET_MAKER_ADR',
            '0xff035c911accf7c7154c51cb62460b50f43ea54f'),
        'buyer_privkey': os.environ.get('XBR_BUYER_PRIVKEY',
            '646f1ce2fdad0e6deeeb5c7e8e5543bdde65e86029e2fd9fc169899c440a7913'),
    }
)


@comp.on_join
async def joined(session, details):
    print('Buyer session joined', details)

    market_maker_adr = binascii.a2b_hex(session.config.extra['market_maker_adr'][2:])
    print('Using market maker adr:', session.config.extra['market_maker_adr'])

    buyer_privkey = binascii.a2b_hex(session.config.extra['buyer_privkey'])

    buyer = SimpleBuyer(market_maker_adr, buyer_privkey, 100)
    balance = await buyer.start(session, details.authid)
    print("Remaining balance={}".format(balance))

    async def on_event(key_id, enc_ser, ciphertext, details=None):
```

(continues on next page)

(continued from previous page)

```
        payload = await buyer.unwrap(key_id, enc_ser, ciphertext)
        print('Received event {}:'.format(details.publication), payload)

    await session.subscribe(on_event, "io.crossbar.example",
        options=SubscribeOptions(details=True))


if __name__ == '__main__':
    run([comp])
```

7.2.3 SimpleSeller

Autobahn includes a "simple seller" for use in seller delegate user services which is able to automatically offer and sell data encryption keys via the market maker to buyers in a market:

7.2.4 SimpleSeller Example

Here is a complete example seller:

```
import binascii
import os
from time import sleep
from uuid import UUID

from autobahn.twisted.component import Component, run
from autobahn.twisted.util import sleep
from autobahn.wamp.types import PublishOptions
from autobahn.xbr import SimpleSeller

comp = Component(
    transports=os.environ.get('XBR_INSTANCE', 'wss://continental2.crossbario.com/ws'),
    realm=os.environ.get('XBR_REALM', 'realm1'),
    extra={
        'market_maker_adr': os.environ.get('XBR_MARKET_MAKER_ADR',
            '0xff035c911accf7c7154c51cb62460b50f43ea54f'),
        'seller_privkey': os.environ.get('XBR_SELLER_PRIVKEY',
            '646f1ce2fdad0e6deeeb5c7e8e5543bdde65e86029e2fd9fc169899c440a7913'),
    }
)

running = False


@comp.on_join
async def joined(session, details):
    print('Seller session joined', details)
    global running
    running = True

    market_maker_adr = binascii.a2b_hex(session.config.extra['market_maker_adr'][2:])
    print('Using market maker adr:', session.config.extra['market_maker_adr'])

    seller_privkey = binascii.a2b_hex(session.config.extra['seller_privkey'])
```

(continues on next page)

(continued from previous page)

```
    api_id = UUID('627f1b5c-58c2-43b1-8422-a34f7d3f5a04').bytes
    topic = 'io.crossbar.example'
    counter = 1

    seller = SimpleSeller(market_maker_adr, seller_privkey)
    seller.add(api_id, topic, 35, 10, None)
    await seller.start(session)

    print('Seller has started')

    while running:
        payload = {'data': 'py-seller', 'counter': counter}
        key_id, enc_ser, ciphertext = await seller.wrap(api_id,
                                                        topic,
                                                        payload)
        pub = await session.publish(topic, key_id, enc_ser, ciphertext,
                                    options=PublishOptions(acknowledge=True))

        print('Published event {}: {}'.format(pub.id, payload))

        counter += 1
        await sleep(1)


@comp.on_leave
def left(session, details):
    print('Seller session left', details)
    global running
    running = False


if __name__ == '__main__':
    run([comp])
```

7.2.5 KeySeries

Helper class used in SimpleSeller to create an auto-rotating (time-interval based) data encryption key series:

7.3 Interface Reference

7.3.1 IMarketMaker

7.3.2 IProvider

7.3.3 IConsumer

7.3.4 ISeller

7.3.5 IBuyer

CHAPTER 8

WebSocket Examples

8.1 Basic Examples

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

8.1.1 Echo

Twisted / asyncio

A simple WebSocket echo server and client.

8.1.2 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.

8.1.3 Testee

Twisted / asyncio

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

8.2 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**.

8.2.1 Secure WebSocket

Twisted

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

8.2.2 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.

8.2.3 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.

8.2.4 Secure WebSocket and Twisted Web

Twisted

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

8.2.5 WebSocket Ping-Pong

Twisted

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

8.2.6 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 9

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.

9.1 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.

9.2 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):

9.3 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.

9.4 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.

9.5 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 10

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.

10.1 Module autobahn.util

autobahn.util.**public**(*obj*)

The public user API of Autobahn is marked using this decorator. Everything that is not decorated @public is library internal, can change at any time and should not be used in user program code.

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.**utcnow**()
Get current time in UTC as ISO 8601 string.

Returns Current time as string in ISO 8601 format.

Return type str

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.**id**()
Generate a new random integer ID from range **[0, 2**53]**.

The generated ID is based on a pseudo-random number generator (Mersenne Twister, which has a period of 2**19937-1). It is NOT cryptographically strong, and hence NOT suitable to generate e.g. secret keys or access tokens.

The upper bound **2**53** is chosen since it is the maximum integer that can be represented as a IEEE double such that all smaller integers are representable as well.

Hence, IDs can be safely used with languages that use IEEE double as their main (or only) number type (JavaScript, Lua, etc).

Returns A random integer ID.

Return type int

autobahn.util.**rid**()
Generate a new random integer ID from range **[0, 2**53]**.

The generated ID is uniformly distributed over the whole range, doesn't have a period (no pseudo-random generator is used) and cryptographically strong.

The upper bound **2**53** is chosen since it is the maximum integer that can be represented as a IEEE double such that all smaller integers are representable as well.

Hence, IDs can be safely used with languages that use IEEE double as their main (or only) number type (JavaScript, Lua, etc).

Returns A random integer ID.

Return type int

autobahn.util.**newid**(*length=16*)
Generate a new random string ID.

The generated ID is uniformly distributed and cryptographically strong. It is hence usable for things like secret keys and access tokens.

Parameters **length** (*int*) – The length (in chars) of the ID to generate.

Returns A random string ID.

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

class autobahn.util.**Stopwatch**(*start=True*)
Stopwatch based on walltime.

This can be used to do code timing and uses the most precise walltime measurement available on the platform. This is a very light-weight object, so create/dispose is very cheap.

Parameters **start** (*bool*) – If `True`, immediately start the stopwatch.

elapsed()
Return total time elapsed in seconds during which the stopwatch was running.

Returns The elapsed time in seconds.

Return type float

pause()
Pauses the stopwatch and returns total time elapsed in seconds during which the stopwatch was running.

Returns The elapsed time in seconds.

Return type float

resume()
Resumes a paused stopwatch and returns total elapsed time in seconds during which the stopwatch was running.

Returns The elapsed time in seconds.

Return type float

stop()
Stops the stopwatch and returns total time elapsed in seconds during which the stopwatch was (previously) running.

Returns The elapsed time in seconds.

Return type float

class autobahn.util.**Tracker**(*tracker*, *tracked*)
A key-based statistics tracker.

absolute(*key*)
Return the UTC wall-clock time at which a tracked event occurred.

Parameters **key** (*str*) – The key

Returns Timezone-naive datetime.

Return type instance of `datetime.datetime`

diff(*start_key*, *end_key*, *formatted=True*)
Get elapsed difference between two previously tracked keys.

Parameters

- **start_key** (*str*) – First key for interval (older timestamp).
- **end_key** (*str*) – Second key for interval (younger timestamp).
- **formatted** (*bool*) – If `True`, format computed time period and return string.

Returns Computed time period in seconds (or formatted string).

Return type float or str

track(*key*)
Track elapsed for key.

Parameters **key** (*str*) – Key under which to track the timing.

class autobahn.util.**EqualityMixin**
Mixing to add equality comparison operators to a class.

Two objects are identical under this mixin, if and only if:

1. both object have the same class
2. all non-private object attributes are equal

class autobahn.util.**ObservableMixin**
Internal utility for enabling event-listeners on particular objects

fire(*event*, **args*, ***kwargs*)
Fire a particular event.

Parameters **event** – the event to fire. All other args and kwargs are passed on to the handler(s) for the event.

Returns a Deferred/Future gathering all async results from all handlers and/or parent handlers.

off(*event=None*, *handler=None*)
Stop listening for a single event, or all events.

Parameters

- **event** – if None, remove all listeners. Otherwise, remove listeners for the single named event.
- **handler** – if None, remove all handlers for the named event; otherwise remove just the given handler.

on(*event*, *handler*)
Add a handler for an event.

Parameters

- **event** – the name of the event
- **handler** – a callable thats invoked when .fire() is called for this events. Arguments will be whatever are given to .fire()

set_valid_events(*valid_events=None*)

Parameters **valid_events** – if non-None, .on() or .fire() with an event not listed in valid_events raises an exception.

class autobahn.util.**IdGenerator**

ID generator for WAMP request IDs.

WAMP request IDs are sequential per WAMP session, starting at 1 and wrapping around at 2**53 (both value are inclusive [1, 2**53]).

The upper bound **2**53** is chosen since it is the maximum integer that can be represented as a IEEE double such that all smaller integers are representable as well.

Hence, IDs can be safely used with languages that use IEEE double as their main (or only) number type (JavaScript, Lua, etc).

See https://github.com/wamp-proto/wamp-proto/blob/master/spec/basic.md#ids

next()

Returns next ID.

Returns The next ID.

Return type int

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 `'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_serial_number**()
Generate a globally unique serial / product code of the form `'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.**generate_user_password**()
Generate a secure, random user password of the form `'kgojzi61dn5dtb6d'`. The generated value is cryptographically strong and has (at least) 76 bits of entropy.

Returns The generated password.

Return type str

10.2 Module autobahn.websocket

10.2.1 WebSocket Interfaces

10.2.2 WebSocket Types

10.2.3 WebSocket Compression

10.2.4 WebSocket Utilities

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

10.3 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.

10.3.1 RawSocket Utilities

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

10.4 Module autobahn.wamp

10.4.1 WAMP Interfaces

10.4.2 WAMP Types

10.4.3 WAMP Exceptions

10.4.4 WAMP Authentication and Encryption

10.4.5 WAMP Serializer

10.4.6 WAMP Messages

10.5 Module autobahn.wamp.component

10.5.1 Component

This is common code for both Twisted and asyncio components; see either `autobahn.twisted.component.Component` or `autobahn.asyncio.component.Component` for the concrete implementations.

10.6 Module autobahn.twisted

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

10.6.1 Component

The component API provides a high-level funcional style method of defining and running WAMP components including authentication configuration

10.6.2 WebSocket Protocols and Factories

Classes for WebSocket clients and servers using Twisted.

10.6.3 WAMP-over-WebSocket Protocols and Factories

Classes for WAMP-WebSocket clients and servers using Twisted.

10.6.4 WAMP-over-RawSocket Protocols and Factories

Classes for WAMP-RawSocket clients and servers using Twisted.

10.6.5 WAMP Sessions

Classes for WAMP sessions using Twisted.

10.7 Module autobahn.asyncio

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

10.7.1 Component

The component API provides a high-level funcional style method of defining and running WAMP components including authentication configuration

10.7.2 WebSocket Protocols and Factories

Classes for WebSocket clients and servers using asyncio.

10.7.3 WAMP-over-WebSocket Protocols and Factories

Classes for WAMP-WebSocket clients and servers using asyncio.

10.7.4 WAMP-over-RawSocket Protocols and Factories

Classes for WAMP-RawSocket clients and servers using asyncio.

10.7.5 WAMP Sessions

Classes for WAMP sessions using asyncio.

CHAPTER 11

Changelog

11.1 20.2.1

- fix: update XBR ABI files to XBR release v20.2.1
- fix: add AuthAnonymous to __all__ (#1303)

11.2 20.1.3

- fix: CI building (caching?) issue "corrupt ZIP file"
- fix: update docker image build scripts and add ARM64/PyPy
- fix: update XBR ABI files
- fix: use `txaio.time_ns()` and drop deprecated `autobahn.util.time_ns()`
- fix: update project README and docs for supported python versions (#1296)
- fix: WebSocket protocol instances now raise *autobahn.exception.Disconnected* when sending on a closed connection (#1002)
- fix: version conflict in xbr downstream application dependency (crossbarfx) (#1295)

11.3 20.1.2

- fix: add *python_requires>=3.5* to prevent installation on python 2 (#1293)

11.4 20.1.1

- IMPORTANT: beginning release v20.1.1, Autobahn|Python only supports Python 3.5 or later.

- fix: first part of cleaning up code, dropping Python 2 support (#1282).

11.5 19.11.2

- IMPORTANT: release v19.11.2 will be the last release supporting Python 2. We will support Python 3.5 and later beginning with Autobahn v20.1.1.
- fix: add docs for parameters to component.py (#1276)
- new: statistics tracking on WAMP serializers `autobahn.wamp.serializer.Serializer`
- new: helper `autobahn.util.time_ns()`

11.6 19.11.1

- fix: argument type check for fragmentSize in sendMessage
- new: start_loop option for WAMP components
- new: ethereum bip39/32 helpers
- new: enable XBR in Docker image build scripts

11.7 19.10.1

- new: updated docker image scripts
- new: add WAMP serializer in use to SessionDetails
- fix: partial support for xb buyers/sellers in pypy
- fix: remove dependency on "ethereum" package (part of pypy support)

11.8 19.9.3

- new: XBR - update XBR for new contract ABIs
- new: XBR - payment channel close
- new: XBR - implement EIP712 signing of messages in endpoints

11.9 19.9.2

- new: XBR - update XBR for new contract ABIs

11.10 19.9.1

- new: XBR - update XBR for new contract ABIs

11.11 19.8.1

- new: implement XBR off-chain delegate transaction signing and verification (#1202)
- new: update XBR for new contract ABIs

11.12 19.7.2

- fix: monkey patch re-add removed helper functions removed in eth-abi
- new: simple blockchain (XBR) client
- new: update XBR ABI files
- new: XBR endpoint transaction signing
- new: client side catching of WAMP URI errors in *session.call|register|publish|subscribe*

11.13 19.7.1

- fix: implement client side payload exceed max size; improve max size exceeded handling
- fix: detect when our transport is "already" closed at connect time (#1215)
- fix: XBR examples

11.14 19.6.2

- fix: add forgotten cryptography dependency (#1205)

11.15 19.6.1

- new: XBR client library integrated (#1201)
- new: add entropy depletion unit tests
- fix: make CLI tool python2 compatible (#1197)
- fix: use cryptography pbkdf2 instead of custom (#1198)
- fix: include tests for packaging (#1194)

11.16 19.5.1

- fix: authextra merging (#1191)
- fix: set default retry_delay_jitter (#1190)
- new: add rawsocket + twisted example (#1189)
- new: WebSocket testing support, via Agent-style interface (#1186)

- new: decorator for on_connectfailure
- fix: delayed call leakage (#1152)
- new: CLI client (#1150)
- fix: set up TLS over proxy properly (#1149)
- new: expose ser modules (#1148)
- fix: base64 encodings, add hex encoding (#1146)
- new: onConnecting callback (with TransportDetails and ConnectingRequest). **Note**: if you've implemented a pure *IWebSocketChannel* without inheriting from Autobahn base classes, you'll need to add an *onConnecting()* method that just does *return None*.

11.17 19.3.3

- fix: RegisterOptions should have details|bool parameter (#1143)
- new: WAMP callee disclosure
- new: WAMP forward_for in more message types; expose forward_for in options/details types
- new: expose underlying serializer modules on WAMP object serializers
- fix: WAMP-cryptosign fix base64 encodings, add hex encoding (#1146)

11.18 19.3.2

- fix: import guards for flatbuffers (missed in CI as we run with "all deps installed" there)

11.19 19.3.1

- new: add experimental support for WAMP-FlatBuffers serializer: EVENT and PUBLISH messages for now only
- new: add FlatBuffers schema for WAMP messages
- fix: improve serializer package preference behavior depending on CPy vs PyPy
- fix: relax protocol violations: ignore unknown INTERRUPT and GOODBYE already sent; reduce log noise
- fix: skipping Yield message if transport gets closed before success callback is called (#1119)
- fix: integer division in logging in py3 (#1120)
- fix: Await tasks after they've been cancelled in *autobahn.asycio.component.nicely_exit* (#1116)

11.20 19.2.1

- fix: set announced roles on appsession object (#1109)
- new: lower log noise on ApplicationErrors (#1107)
- new: allow explicit passing of tx endpoint and reactor (#1103)

- new: add attribute to forward applicationrunner to applicationsession via componentconfig

11.21 19.1.1

- new: adding marshal on SessionDetails

11.22 18.12.1

- fix: return the wrapped function from component decorators (#1093)
- new: add proxy= support for Component transports (#1091)
- fix: Ticket1077 stop start (#1090)
- fix: cleanup cancel handling (#1087)

11.23 18.11.2

- fix: asyncio unregisterProducer raises exception (#1079)
- fix: URL is not required in RawSocket configuration items with WAMP component API
- fix: revert PR https://github.com/crossbario/autobahn-python/pull/1075

11.24 18.11.1

- new: forward_for WAMP message attribute (for Crossbar.io Router-to-Router federation)
- new: support RawSocket URLs (eg "rs://localhost:5000" or "rs://unix:/tmp/file.sock")
- new: support WAMP-over-Unix sockets for WAMP components ("new API")
- fix: use same WAMP serializer construction code for WAMP components ("new API") and ApplicationSession/Runner
- fix: memory leak with Twisted/WebSocket, dropConnection and producer

11.25 18.10.1

- Don't eat Component.stop() request when crossbar not connected (#1066)
- handle async on_progress callbacks properly (#1061)
- fix attribute error when ConnectionResetError does not contain "reason" attribute (#1059)
- infer rawsocket host, port from URL (#1056)
- fix error on connection lost if no reason (reason = None) (#1055)
- fixed typo on class name (#1054)

11.26 18.9.2

- fix: TLS error logging (#1052)

11.27 18.9.1

- new: Interrupt has Options.reason to signal detailed origin of call cancelation (active cancel vs passive timeout)
- fix: Cancel and Interrupt gets `"killnowait"` mode
- new: Cancel and Interrupt no longer have `ABORT`/`"abort"`

11.28 18.8.2

- new: WAMP call cancel support
- fix: getting started documentation and general docs improvements
- fix: WebSocket auto-reconnect on opening handshake failure
- fix: more Python 3.7 compatibility and CI
- fix: Docker image building using multi-arch, size optimizations and more
- fix: asyncio failed to re-connect under some circumstances (#1040, #1041, #1010, #1030)

11.29 18.8.1

- fix: Python 3.7 compatibility
- fix: remove Python 2.6 support leftovers
- new: getting started docker-based examples in matching with docs

11.30 18.7.1

- new: Python 3.7 supported and integrated into CI
- new: WAMP-SCRAM examples
- fix: glitches in WAMP-SCRAM

11.31 18.6.1

- fix: implement abort argument for asyncio in WebSocketAdapterProtocol._closeConnection (#1012)

11.32 18.5.2

- fix: security (DoS amplification): a WebSocket server with permessage-deflate turned on could be induced to waste extra memory through a "zip-bomb" style attack. Setting a max-message-size will now stop deflating compressed data when the max is reached (instead of consuming all compressed data first). This could be used by a malicious client to make the server waste much more memory than the bandwidth the client uses.

11.33 18.5.1

- fix: asyncio/rawsocket buffer processing
- fix: example failures due to pypy longer startup time (#996)
- fix: add on_welcome for AuthWampCra (#992)
- fix: make run() of multiple components work on Windows (#986)
- new: *max_retries* now defaults to -1 ("try forever")

11.34 18.4.1

- new: WAMP-SCRAM authentication
- new: native vector extensions (NVX)
- fix: improve choosereactor (#965, #963)
- new: lots of new and improved documentation, component API and more
- new: Docker image tooling now in this repo
- fix: "fatal errors" in Component (#977)
- fix: AIO/Component: create a new loop if already closed
- fix: kwarg keys sometimes are bytes on Python2 (#980)
- fix: various improvements to new component API

11.35 18.3.1

- fix: endpoint configuration error messages (#942)
- fix: various improvements to the new components API (including retries)
- fix: pass *unregisterProducer* through to twisted to complement *WebSocketAdapterProtocol.registerProducer* (#875)

11.36 17.10.1

- fix: proxy support (#918)
- fix: ensure that a future is not done before rejecting it (#919)
- fix: don't try to reject cancelled futures within pending requests when closing the session

11.37 17.9.3

Published 2017-09-23

- new: user configurable backoff policy
- fix: close aio loop on exit
- fix: some component API cleanups
- fix: cryptosign on py2
- new: allow setting correlation_is_last message marker in WAMP messages from user code

11.38 17.9.2

Published 2017-09-12

- new: allow setting correlation URI and anchor flag in WAMP messages from user code
- fix: WebSocket proxy connect on Python 3 (unicode vs bytes bug)

11.39 17.9.1

Published 2017-09-04

- new: allow setting correlation ID in WAMP messages from user code
- fix: distribute LICENSE file in all distribution formats (using setup.cfg metadata)

11.40 17.8.1

Published 2017-08-15

- new: prefix= kwarg now available on ApplicationSession.register for runtime method names
- new: @wamp.register(None) will use the function-name as the URI
- new: correlation and uri attributes for WAMP message tracing

11.41 17.7.1

Published 2017-07-21

- new: lots of improvements of components API, including asyncio support

11.42 17.6.2

Published 2017-06-24

- new: force register option when joining realms
- fix: TLS options in components API

11.43 17.6.1

Published 2017-06-07

- new: allow components to pass WebSocket/RawSocket options
- fix: register/subscribe decorators support different URI syntax from what session.register and session.subscribe support
- new: allow for standard Crossbar a.c..d style pattern URIs to be used with Pattern
- new: dynamic authorizer example
- new: configurable log level in *ApplicationRunner.run* for asyncio
- fix: forward reason of hard dropping WebSocket connection in *wasNotCleanReason*

11.44 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"

11.45 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

11.46 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

11.47 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

11.48 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

11.49 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

11.50 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

11.51 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

11.52 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

11.53 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

11.54 0.14.1

Published 2016-05-26

- fix: unpinned Twisted version again
- fix: remove X-Powered-By header
- fix: removed decrecated args to ApplicationRunner

11.55 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)

11.56 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

11.57 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).

11.58 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

11.59 0.11.0

Published 2015-12-09

11.60 0.10.9

Published 2015-09-15

- fixes regression #500 introduced with commit 9f68749

11.61 0.10.8

Published 2015-09-13

- maintenance release with some issues fixed

11.62 0.10.7

Published 2015-09-06

- fixes a regression in 0.10.6

11.63 0.10.6

Published 2015-09-05

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

11.64 0.10.5

Published 2015-08-06

- maintenance release with lots of smaller bug fixes

11.65 0.10.4

Published 2015-05-08

- maintenance release with some smaller bug fixes

11.66 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

11.67 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

11.68 0.10.1

Published 2015-03-01

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

11.69 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

11.70 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)

11.71 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

11.72 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

11.73 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*

11.74 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

11.75 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

11.76 0.9.1

Published 2014-09-22

- maintenance release with some smaller bug fixes

11.77 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

11.78 0.8.15

Published 2014-08-23

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

11.79 0.8.14

Published 2014-08-14

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

11.80 0.8.13

Published 2014-08-05

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

11.81 0.8.12

Published 2014-07-23

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

11.82 0.8.11

Published

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

11.83 0.8.10

Published

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

11.84 0.8.9

Published

- maintenance release

11.85 0.8.8

Published

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

11.86 0.8.7

Published

- maintenance release

11.87 0.8.6

Published

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

11.88 0.8.5

Published

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

11.89 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

11.90 0.8.3

Published

- fix bug with closing router app sessions

11.91 0.8.2

Published

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

11.92 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

11.93 0.8.0

Published

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

11.94 0.7.4

Published

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

11.95 0.7.3

Published

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

11.96 0.7.2

Published

- really fix setup/packaging

11.97 0.7.1

Published

- setup fixes
- fixes for Python2.6

11.98 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

11.99 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

11.100 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)

11.101 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

11.102 0.5.14

Published

- base version when we started to maintain a changelog

Python Module Index

a

Index