# txaio

*Release 18.8.1*

**Jan 21, 2020**

# Contents

CHAPTER 1

# txaio

**txaio** is a helper library for writing code that runs unmodified on both Twisted and asyncio / Trollius.

This is like six, but for wrapping over differences between Twisted and asyncio so one can write code that runs unmodified on both (aka *source code compatibility*). In other words: your *users* can choose if they want asyncio **or** Twisted as a dependency.

Note that, with this approach, user code **runs under the native event loop of either Twisted or asyncio**. This is different from attaching either one's event loop to the other using some event loop adapter.

## 1.1 Platform support

**txaio** runs on CPython 2.7/3.3+ and PyPy 2/3, on top of Twisted or asyncio. Specifically, **txaio** is tested on the following platforms:

**Python 2:**

- CPython 2.7 on Twisted 12.1, 13.2, 15.4, 16.5, trunk and on Trollius 2.0
- PyPy 2 on Twisted 12.1, 13.2, 15.4, 16.5, trunk and on Trollius 2.0

**Python 3:**

- CPython 3.4 on Twisted 15.4, 16.5, trunk and on asyncio (stdlib)
- CPython 3.5 on Twisted 15.4, 16.5, trunk and on asyncio (stdlib)
- CPython 3.6 on Twisted 15.4, 16.5, trunk and on asyncio (stdlib)
- PyPy 3 on Twisted 15.4, 16.5, trunk and on asyncio (stdlib)

## 1.2 How it works

Instead of directly importing, instantiating and using `Deferred` (for Twisted) or `Future` (for asyncio) objects, **txaio** provides helper-functions to do that for you, as well as associated things like adding callbacks or errbacks.

This obviously changes the style of your code, but then you can choose at runtime (or import time) which underlying event-loop to use. This means you can write **one** code-base that can run on Twisted *or* asyncio (without a Twisted dependency) as you or your users see fit.

Code like the following can then run on *either* system:

```
import txaio
txaio.use_twisted()  # or .use_asyncio()

f0 = txaio.create_future()
f1 = txaio.as_future(some_func, 1, 2, key='word')
txaio.add_callbacks(f0, callback, errback)
txaio.add_callbacks(f1, callback, errback)
# ...
txaio.resolve(f0, "value")
txaio.reject(f1, RuntimeError("it failed"))
```

Please refer to the documentation for description and usage of the library features.

CHAPTER 2

# Overview

## 2.1 Brief History

This library has been factored out of the Autobahn|Python WAMP client library. The `ApplicationSession` object from that project therefore serves as a good example of how to use this library in a complex use-case.

We are releasing it in the hopes these utilities are useful on their own to other projects using event-based Python. Only authors of "library style" code are likely to be interested in this – new application code should use your favourite Python asynchronous I/O platform.

## 2.2 Overview by Example

The simplest way to use **txaio** is to `import txaio` and use the helper functions directly. You must select the framework you wish to use by calling `txaio.use_twisted()` or `txaio.use_asyncio()` (which means asyncio, or trollius/tuplip if asyncio import fails).

Note that to use this library successfully you *shouldn't* call methods on futures – use *only* **txaio** methods to operate on them.

```
import txaio
txaio.use_twisted()  # or .use_asyncio()

def cb(value):
    print("Callback:", value)

def eb(fail):
    # fail will implement txaio.IFailedFuture
    print("Errback:", txaio.failure_message(fail))
    print(txaio.failure_formatted_traceback(fail))

f = txaio.create_future()
txaio.add_callbacks(f, cb, eb)
```

(continues on next page)

(continued from previous page)

```
# ...other things happen...

try:
    answer = do_something()
    fail = None
except Exception:
    fail = txaio.create_failure()

# the point here is that you "somehow" arrange to call either
# reject() or resolve() on every future you've created.

if fail:
    txaio.reject(f, fail)
else:
    txaio.resolve(f, answer)
```

## 2.3 Restrictions and Caveats

**txaio** is not a new event-based programming solution. It is not a complete box-set of asynchronous programming tools.

It is **one piece** that *can* help **library authors** to write cross-event-loop asynchronous code. For example, you'll note that there's no way to run "the event loop" – that's up to you.

There is **no support for @coroutine or @inlineCallbacks** decorators. This is not possible, as asyncio under Python3 introduced a new syntax (`yield from`) to call into other co-routines. So, you are stuck with "callback style" code for your cross-platform library. (Note that *users* of your library can of course use new Python3 features like `yield from`, `async` and `await` in their own code – but they do so by explicitly choosing "Python3 and asyncio" as their platform).

`txaio` is basically a "lowest common denominator" tool. There is a minimum of wrapping, etcetera but the library author doesn't get to use fancy features (e.g. `@inlineCallbacks`, mutation of returns, `@coroutine`) of the underlying async platforms.

## 2.4 Futures and Deferreds

In most cases asyncio is trying to be "as thin as possible" wrapper around the different APIs. So, there's nothing wrapping Future or Deferred – you get the bare objects. This means that *`txaio.create_future()`* returns you the native object, which you then pass to *`txaio.add_callbacks()`*

Similarly, *`txaio.call_later()`* returns the underlying object (`IDelayedCall` in Twisted or a `Handle` in asyncio). These both have a `cancel()` method, but little else in common.

## 2.5 Callbacks and Errbacks

Twisted and asyncio have made different design-decisions. One that stands out is callbacks, and callback chaining. In Twisted, the return value from an earlier callback is what gets passed to the next callback. Similarly, errbacks in Twisted can cancel the error. There are not equivalent facilities in `asyncio`: if you add multiple callbacks, they all get the same value (or exception).

When using **txaio**, **don't depend on chaining**. This means that your `callback` and `errback` methods must **always return their input argument** so that Twisted works if you add multiple callbacks or errbacks (and doesn't unexpectedly cancel errors).

**txaio** does add the concept of an `errback` for handling errors (a concept asyncio does not have) and therefore adds one helper to encapsulate exceptions (similar to Twisted's Failure object) which only exists in the asyncio implementation.

There is no `inlineCallbacks` or `coroutine` decorator support. Don't use these.

## 2.6 Error Handling

In your `errback`, you will receive a single arg which is an instance conforming to `IFailedFuture`. This interface has only a single attribute: `.value`, which is the Exception instance which caused the error. You can also use `txaio.failure_*` methods to operate on an `IFailedFuture`:

- txaio.failure_message: returns a unicode error-message
- txaio.failure_traceback: returns a `traceback` object
- txaio.failure_formatted_traceback: returns a unicode formatted stack-trace

You should **not** depend on *any* other attributes or methods of the instance you're given.

## 2.7 Real Examples

You are encouraged to look at Autobahn|Python for an example of a system that can run on both Twisted and asyncio. In particular, look at the difference between `autobahn/twisted/websocket.py` and `autobahn/asyncio/websocket.py` and the compatibility super-class in `autobahn/wamp/protocol.py` which is the piece that uses **txaio** to provide an event-loop agnostic implementation that both the Twisted and asyncio concrete `ApplicationSession` objects inherit from.

`autobahn.wamp.protocol.ApplicationSession` is glued to a particular event-loop via `autobahn.twisted.wamp.ApplicationSession` which takes advantage of `txaio.tx.LoopMixin` to provide the helpers-methods attached to `self`.

In this manner, code in the generic implementation simply always calls **txaio** methods via `self.create_future()` or similar and users of Autobahn|Python can choose between asyncio and Twisted as they prefer by either `from autobahn.twisted.wamp import ApplicationSession` or `from autobahn.asyncio.wamp import ApplicationSession`

## 2.8 Cross-API Magic

If you wish to write Twisted-like code that uses `asyncio` as its event-loop, you should look at txtulip. I do not know of a project that lets you write asyncio-like code that runs on Twisted's event-loop.

# CHAPTER 3

# Programming Guide

This section is a work in progress and suggestions are welcome.

## 3.1 Explict Event Loops

Twisted has a single, global reactor (for now). As such, txaio was built with a single, global (but configurable) event-loop. However, asyncio supports multiple event-loops.

After version 2.7.0 it is possible to use txaio with multiple event-loops, and thereby offer asyncio users the chance to pass one. Of course, it's still not possible to use multiple event-loops at once with Twisted.

To start using multiple event-loops with txaio, use `txaio.with_config()` to return a new "instance" of the txaio API with the given config (the only thing you can configure currently is the event-loop). On Twisted, it's an error if you try to use a different reactor.

The object returned by `txaio.with_config()` is a drop-in replacement for every *txaio.** call, so you can go from code like this:

```
import txaio
f = txaio.create_future()
```

. . . and instead make your code do look like this:

```
import asyncio
import txaio
txa = txaio.with_config(loop=asyncio.new_event_loop())
f = txa.create_future()
```

If you're doing this inside a class, you could use `self._txa` or similar instead. This gives you an easy path to opt-in to this multiple event-loop API:

- replace all `txaio.*` calls to use an object, like `self._txa`.
- assign this to the txaio module (`self._txa = txaio`) or use the new API right away (`self._txa = txaio.with_config()`)

- add a public API to your library to pass in an event loop
- when this is used, you set `self._txa = txaio.with_config(loop=loop)`

See the example in `examples/multiloop.py`.

## 3.2 Logging

If you are developing a new application, you can take advantage of more structured logging by using txaio's APIs throughout. This API is similar to Twisted's logging in many ways, but not identical. If you're integrating txaio into existing code, it should "play nicely" with the `logging` module, Twisted's newest logger, and the pre-15.2.0 "legacy" Twisted logger.

To create an object suitable for logging, call *`txaio.make_logger()`*. This will return an instance which has a series of methods indicating the "severity" or "level" of the log – see *`txaio.interfaces.ILogger`* for an example and more details.

So, given some code like:

```
import txaio
txaio.use_twisted()

class Bunny(object):
    log = txaio.make_logger()

    def hop(self, times=1):
        self.log.trace("Bunny.hop(times={times})", times=times)
        self.log.debug("Hopping {times} times.", times=times)
        try:
            1 / 0
        except Exception:
            fail = txaio.create_failure()
            self.log.critical(txaio.failure_format_traceback(fail))

print("output before start_logging")
txaio.start_logging(level='debug')
print("output after start_logging")
jack = Bunny()
jack.hop(42)
```

Then you should see output approximately like this:

```
output before start_logging
2016-01-21T01:02:03-0100 output after start_logging
2016-01-21T01:02:03-0100 Hopping 42 times.
2016-01-21T01:02:03-0100 Traceback (most recent call last):
  File "logging-example.py", line 21, in <module>
    jack.hop(42)
--- <exception caught here> ---
  File "logging-example.py", line 12, in hop
    raise RuntimeError("Fox spotted!")
exceptions.RuntimeError: Fox spotted!
```

Note that the `trace`-level message wasn't logged. If you don't like to see full tracebacks except with debugging, you can use this idiom:

```
self.log.critical(txaio.failure_message(fail))
self.log.debug(txaio.failure_format_traceback(fail))
```

It's worth noting the code doesn't change at all if you do `.use_asyncio()` at the top instead – of course this is the whole point of `txaio`!

## 3.3 Logging Interoperability

When you're using libraries that are already doing logging, but not using the `txaio` APIs, you shouldn't need to do anything. For example:

```
import txaio
txaio.use_twisted()


def existing_code():
    from twisted.python import log
    log.msg("A legacy Twisted logger message")

txaio.start_logging(level='debug')
existing_code()
```

If you're using `asyncio` (or just built-in Python logging), it could look like this:

```
import txaio
txaio.use_asyncio()


def existing_code():
    import logging
    log = logging.getLogger("roy")
    log.info("Python stdlib message: %s", "txaio was here")

txaio.start_logging(level='debug')
existing_code()
```

## 3.4 Starting Logging Yourself

If you are already starting your favourite logging system yourself (be that Twiste'd logger via `globalLogBeginner` or Python stdlib logging), any library using txaio's logging should play nicely with it. **Not** ever calling `txaio.start_logging()` has a slight drawback, however: as part of setting up logging, we re-bind all the "unused" logging methods to do-nothing. For example, if the log level is set to `'info'` than the `.debug` method on all txaio-created logger instances becomes a no-op.

For fully-worked examples of this, look in `examples/log_interop_stdlib.py` and `examples/log_interop_twisted.py`.

# CHAPTER 4

# txio releases

## 4.1 master

- ...

## 4.2 18.8.1

- add API to support cancellation; this means passing a 1-argument callable to `create_future` and `txaio.cancel` to actually cancel a future
- support Python 3.7 (CI / testing added)

## 4.3 18.7.1

- move to calver
- deprecate Python 3.3 support and CI testing

## 4.4 2.10.0

- the asyncio version of `make_logger` now deduces a proper namespace instead of using the root (thanks spr0cketeer)

## 4.5 2.9.0

- March 2, 2018

## 4.6 2.8.2

- September 4, 2017
- fix: no longer install LICENSE file into installation directory (conflicts!)

## 4.7 2.8.1

- July 21, 2017
- fix: the asyncio version of sleep() correctly returns a Future instance

## 4.8 2.8.0

- June 8, 2017
- fix: asyncio - remove the hacks for "simulating" chained futures (no longer works - cpy36 has native code for future)
- new: run CI on Python 3.5 and 3.6

## 4.9 2.7.1

- May 1, 2017
- asyncio: example and docs for running multiple loops
- asyncio: log exception tracebacks when they're available for error-message

## 4.10 2.7.0

- April 15, 2017
- allow alternate asyncio loops
- new future creation API for alternate loops

## 4.11 2.6.1

- February 9, 2017
- added inline sleep helper (Twisted only for now)

## 4.12 2.6.0

- December 29, 2016
- avoid giving negative times to *callLater* with batched timers (issue #81)

## 4.13 2.5.2

- November 6, 2016
- fix pytest3/2
- fix Sphinx 1.4+ doc building
- Copyrights transferred from Tavendo to Crossbar.io Technologies

## 4.14 2.5.1

- April 28, 2016
- Bug with `make_batched_timer` remembering (via a closure) the reactor/event-loop too persistantly

## 4.15 2.5.0

- April 28, 2016
- Document that `@coroutine` and `@inlineCallbacks` are not supported
- Objects returned from the *txaio.make_batched_timer()* API now have millisecond resolution and spread out their notifications over the entire range of the bucket.

## 4.16 2.4.0

- April 22, 2016
- Added *txaio.make_batched_timer()* API. The main use-case for this is when you have lots of of timers, but their exact resolution isn't important; batching them into buckets causes far fewer delayed call instances to be outstanding in the underlying event-loop/reactor.

## 4.17 2.3.1

- April 10, 2016
- added universal wheels

## 4.18 2.3.0

- April 9, 2016
- More logging infrastructure and APIs to support moving all of Crossbar.io's logging to txaio.

## 4.19 previous releases

- We didn't produce any release notes prior to 2.4.0
- Instead of making up summaries of all previous releases after the fact, you will have to do something like `git log v1.1.0..v2.0.0` to see what changed between releases. If you **do** make a summary, pull-requests are welcome!

CHAPTER 5

# API

The API is identical whether you're using Twisted or asyncio under the hood. Two `bool` variables are available if you need to know which framework is in use, and two helpers to enforce one or the other framework.

## 5.1 Explicitly Selecting a Framework

Until you explicitly select a framework, all txaio API methods just throw a usage error. So, you must call `.use_twisted()` or `.use_asyncio()` as appropriate. These will fail with `ImportError` if you don't have the correct dependencies.

```
import txaio
txaio.use_twisted()
txaio.use_asyncio()
```

## 5.2 Set an Event Loop / Reactor

You can set `txaio.config.loop` to either an EventLoop instance (if using asyncio) or an explicit reactor (if using Twisted). By default, `reactor` is imported from `twisted.internet` on the first `call_later` invocation. For asyncio, `asyncio.get_event_loop()` is called at import time.

If you've installed your reactor before `import txaio` you shouldn't need to do anything.

Note that under Twisted, only the IReactorTime interface is required.

## 5.3 Test Helpers

Test utilities are in `txaio.testutil`. There is a context-manager for testing delayed calls; see `test_call_later.py` for an example.

txaio.testutil.**replace_loop**(*new_loop*)
This is a context-manager that sets the txaio event-loop to the one supplied temporarily. It's up to you to ensure you pass an event_loop or a reactor instance depending upon asyncio/Twisted.

Use like so:

```
from twisted.internet import task
with replace_loop(task.Clock()) as fake_reactor:
    f = txaio.call_later(5, foo)
    fake_reactor.advance(10)
    # ...etc
```

## 5.4 txaio module

txaio.**using_twisted**
`True` only if we're using Twisted as our underlying event framework

txaio.**using_asyncio**
`True` only if we're using asyncio as our underlying event framework

txaio.**use_asyncio**()
Select `asyncio` framework (uses trollius/tulip on Pythons that lack asyncio).

txaio.**use_twisted**()
Select the Twisted framework (will fail if Twisted is not installed).

txaio.**create_future**(*result=None*, *error=None*, *canceller=None*)
Create and return a new framework-specific future object. On asyncio this returns a Future, on Twisted it returns a Deferred.

**Parameters**

- **result** – if not `None`, the future is already fulfilled, with the given result.
- **error** (class:*IFailedFuture* or Exception) – if not `None` then the future is already failed, with the given error.
- **canceller** – a single-argument callable which is invoked if this future is cancelled (the single argument is the future object which has been cancelled)

**Raises** **ValueError** – if both `value` and `error` are provided.

**Returns** under Twisted a Deferred, under asyncio a Future

txaio.**as_future**(*func*, **args*, ***kwargs*)
Call `func` with the provided arguments and keyword arguments, and always return a Future/Deferred. If `func` itself returns a future, that is directly returned. If it immediately succeed or failed then an already-resolved Future/Deferred is returned instead.

This allows you to write code that calls functions (e.g. possibly provided from user-code) and treat them uniformly. For example:

```
p = txaio.as_future(some_function, 1, 2, key='word')
txaio.add_callbacks(p, do_something, it_failed)
```

You therefore don't have to worry if the underlying function was itself asynchronous or not – your code always treats it as asynchronous.

txaio.**reject**(*future*, *error=None*)
Resolve the given future as failed. This will call any errbacks registered against this Future/Deferred. On Twisted, the errback is called with a bare Failure instance; on asyncio we provide an object that implements `IFailedFuture` because there is no equivalent in asyncio (this mimics part of the Failure API).

**Parameters**

- **future** – an unresolved Deferred/Future as returned by *create_future()*
- **error** (`IFailedFuture` or `Exception`) – The error to fail the Deferred/Future with. If this is `None`, `sys.exc_info()` is used to create an `txaio.IFailedFuture` (or Failure) wrapping the current exception (so in this case it must be called inside an `except:` clause).

txaio.**cancel**(*future*)
Cancel the given future. If a `canceller` was registered, it is invoked now. It is invalid to `resolve` or `reject` the future after cancelling it.

**Parameters** **future** – an unresolved Deferred/Future as returned by *create_future()*

txaio.**resolve**(*future*, *value*)
Resolve the given future with the provided value. This triggers any callbacks registered against this Future/Deferred.

txaio.**add_callbacks**(*future*, *callback*, *errback*)
Adds the provided callback and/or errback to the given future. To add multiple callbacks, call this method multiple times. For example, to add just an errback, call `add_callbacks(p, None, my_errback)`

Note that `txaio` doesn't do anything special with regards to callback or errback chaining – it is highly recommended that you always return the incoming argument unmodified in your callback/errback so that Twisted and asyncio behave the same. For example:

```
def callback_or_errback(value):
    # other code
    return value
```

**Raises** **ValueError** – if both callback and errback are None

txaio.**failure_message**(*fail*)
Takes an `txaio.IFailedFuture` instance and returns a formatted message suitable to show to a user. This will be a `str` with no newlines for the form: `{exception_name}: {error_message}` where `error_message` is the result of running `str()` on the exception instance (under asyncio) or the result of `.getErrorMessage()` on the Failure under Twisted.

txaio.**failure_traceback**(*fail*)
Take an `txaio.IFailedFuture` instance and returns the Python `traceback` instance associated with the failure.

**failure_format_traceback(fail):**
Take an `txaio.IFailedFuture` instance and returns a formatted string showing the traceback. Typically, this will have many newlines in it and look like a "normal" Python traceback.

txaio.**call_later**(*delay*, *func*, **args*, ***kwargs*)
This calls the function `func` with the given parameters at the specified time in the future. Although asyncio doesn't directly support kwargs with `loop.call_later` we wrap it in a `functools.partial`, as asyncio documentation suggests.

Note: see *txaio.make_batched_timer()* if you may have a lot of timers, and their absolute accuracy isn't very important.

**Parameters** **delay** – how many seconds in the future to make the call

**Returns** The underlying library object, which will at least have a `.cancel()` method on it. It's really IDelayedCall in Twisted and a Handle in asyncio.

`txaio.`**`make_batched_timer`**(*seconds_per_bucket*, *chunk_size*)

This returns an object implementing `txaio.IBatchedTimer` such that any `.call_later` calls done through it (instead of via *`txaio.call_later()`*) will be "quantized" into buckets and processed in `chunk_size` batches "near" the time they are supposed to fire. `seconds_per_bucket` is only accurate to "milliseconds".

When there are "tens of thousands" of outstanding timers, CPU usage can become a problem – if the accuracy of the timers isn't very important, using "batched" timers can greatly reduce the number of "real" delayed calls in the event loop.

For example, Autobahn uses this feature for auto-ping timeouts, where the exact time of the event isn't extremely important – but there are 2 outstanding calls per connection.

`txaio.`**`gather`**(*futures*, *consume_exceptions=True*)

Returns a new Future that waits for the results from all the futures provided.

The Future/Deferred returned will callback with a list the same length as `futures` containing either the return value from each future, or an `IFailedFuture`/Failure instance if it failed.

Note that on Twisted, we use DeferredList which usually returns a list of 2-tuples of `(status, value)`. We do inject a callback that unpacks this list to be just the value (or Failure) so that your callback can be identical on Twisted and asyncio.

`txaio.`**`make_logger`**()

Creates and returns an instance of `ILogger`. This can pick up context from where it's instantiated (e.g. the containing class or module) so the best way to use this is to create a logger for each class that produces logs; see the example in `ILogger` 's documentation

**class** `txaio.interfaces.`**`ILogger`**

Bases: `object`

This defines the methods you can call on the object returned from *`txaio.make_logger()`* – although the actual object may have additional methods, you should *only* call the methods listed here.

All the log methods have the same signature, they just differ in what "log level" they represent to the handlers/emitters. The `message` argument is a format string using PEP3101-style references to things from the `kwargs`. Note that there are also the following keys added to the `kwargs`: `log_time` and `log_level`.

For example:

```
class MyThing(object):
    log = txaio.make_logger()

    def something_interesting(self, things=dict(one=1, two=2)):
        try:
            self.log.debug("Called with {things[one]}", things=things)
            result = self._method_call()
            self.log.info("Got '{result}'.", result=result)
        except Exception:
            fail = txaio.create_failure()
            self.log.critical(txaio.failure_format_traceback(fail))
```

The philsophy behind txaio's interface is fairly similar to Twisted's logging APIs after version 15. See Twisted's documentation for details.

**`critical`**(*message*, ***kwargs*)

log a critical-level message

**error**(*message*, ***kwargs*)
log a error-level message

**warn**(*message*, ***kwargs*)
log a error-level message

**info**(*message*, ***kwargs*)
log an info-level message

**debug**(*message*, ***kwargs*)
log an debug-level message

**trace**(*message*, ***kwargs*)
log a trace-level message

**class** txaio.interfaces.**IFailedFuture**
Bases: object

This defines the interface for a common object encapsulating a failure from either an asyncio task/coroutine or a Twisted Deferred.

An instance implementing this interface is given to any errback callables you provide via *txaio.add_callbacks()*

In your errback you can extract information from an IFailedFuture with *txaio.failure_message()* and *txaio.failure_traceback()* or use .value to get the Exception instance.

Depending on other details or methods will probably cause incompatibilities between asyncio and Twisted.

**value**
An actual Exception instance. Same as the second item returned from sys.exc_info()

# Python Module Index

## t

# Index