Tornado Documentation

Release 4.4.2

The Tornado Authors

October 01, 2016

Contents

Tornado is a Python web framework and asynchronous networking library, originally developed at FriendFeed. By using non-blocking network I/O, Tornado can scale to tens of thousands of open connections, making it ideal for long polling, WebSockets, and other applications that require a long-lived connection to each user.

CHAPTER 1

Quick links

- Download version 4.4.2: tornado-4.4.2.tar.gz (release notes)
- Source (github)
- Mailing lists: discussion and announcements
- Stack Overflow
- Wiki

CHAPTER 2

Hello, world

Here is a simple "Hello, world" example web app for Tornado:

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

def make_app():
    return tornado.web.Application([
        (r"/", MainHandler),
    ])

if __name__ == "__main__":
    app = make_app()
    app.listen(8888)
    tornado.ioloop.IOLoop.current().start()
```

This example does not use any of Tornado's asynchronous features; for that see this simple chat room.

CHAPTER 3

Installation

Automatic installation:

```
pip install tornado
```

Tornado is listed in PyPI and can be installed with `pip` or `easy_install`. Note that the source distribution includes demo applications that are not present when Tornado is installed in this way, so you may wish to download a copy of the source tarball as well.

Manual installation: Download tornado-4.4.2.tar.gz:

```
tar xvzf tornado-4.4.2.tar.gz
cd tornado-4.4.2
python setup.py build
sudo python setup.py install
```

The Tornado source code is hosted on GitHub.

Prerequisites: Tornado 4.3 runs on Python 2.7, and 3.3+ For Python 2, version 2.7.9 or newer is *strongly* recommended for the improved SSL support. In addition to the requirements which will be installed automatically by `pip` or `setup.py install`, the following optional packages may be useful:

- concurrent.futures is the recommended thread pool for use with Tornado and enables the use of *`ThreadedResolver`*. It is needed only on Python 2; Python 3 includes this package in the standard library.
- pycurl is used by the optional `tornado.curl_httpclient`. Libcurl version 7.19.3.1 or higher is required; version 7.21.1 or higher is recommended.
- Twisted may be used with the classes in *`tornado.platform.twisted`*.
- pycares is an alternative non-blocking DNS resolver that can be used when threads are not appropriate.
- monotonic or Monotime add support for a monotonic clock, which improves reliability in environments where clock adjustements are frequent. No longer needed in Python 3.3.

Platforms: Tornado should run on any Unix-like platform, although for the best performance and scalability only Linux (with `epoll`) and BSD (with `kqueue`) are recommended for production deployment (even though Mac OS X is derived from BSD and supports kqueue, its networking performance is generally poor so it is recommended only for development use). Tornado will also run on Windows, although this configuration is not officially supported and is recommended only for development use.

CHAPTER 4

Documentation

This documentation is also available in PDF and Epub formats.

4.1 User's guide

4.1.1 Introduction

Tornado is a Python web framework and asynchronous networking library, originally developed at FriendFeed. By using non-blocking network I/O, Tornado can scale to tens of thousands of open connections, making it ideal for long polling, WebSockets, and other applications that require a long-lived connection to each user.

Tornado can be roughly divided into four major components:

- A web framework (including `RequestHandler` which is subclassed to create web applications, and various supporting classes).
- Client- and server-side implementions of HTTP (`HTTPServer` and `AsyncHTTPClient`).
- An asynchronous networking library including the classes `IOLoop` and `IOStream`, which serve as the building blocks for the HTTP components and can also be used to implement other protocols.
- A coroutine library (`tornado.gen`) which allows asynchronous code to be written in a more straightforward way than chaining callbacks.

The Tornado web framework and HTTP server together offer a full-stack alternative to WSGI. While it is possible to use the Tornado web framework in a WSGI container (`WSGIAdapter`), or use the Tornado HTTP server as a container for other WSGI frameworks (`WSGIContainer`), each of these combinations has limitations and to take full advantage of Tornado you will need to use the Tornado's web framework and HTTP server together.

4.1.2 Asynchronous and non-Blocking I/O

Real-time web features require a long-lived mostly-idle connection per user. In a traditional synchronous web server, this implies devoting one thread to each user, which can be very expensive.

To minimize the cost of concurrent connections, Tornado uses a single-threaded event loop. This means that all application code should aim to be asynchronous and non-blocking because only one operation can be active at a time.

The terms asynchronous and non-blocking are closely related and are often used interchangeably, but they are not quite the same thing.

Blocking

A function **blocks** when it waits for something to happen before returning. A function may block for many reasons: network I/O, disk I/O, mutexes, etc. In fact, *every* function blocks, at least a little bit, while it is running and using the CPU (for an extreme example that demonstrates why CPU blocking must be taken as seriously as other kinds of blocking, consider password hashing functions like bcrypt, which by design use hundreds of milliseconds of CPU time, far more than a typical network or disk access).

A function can be blocking in some respects and non-blocking in others. For example, *tornado.httpclient* in the default configuration blocks on DNS resolution but not on other network access (to mitigate this use *ThreadedResolver* or a `tornado.curl_httpclient` with a properly-configured build of `libcurl`). In the context of Tornado we generally talk about blocking in the context of network I/O, although all kinds of blocking are to be minimized.

Asynchronous

An **asynchronous** function returns before it is finished, and generally causes some work to happen in the background before triggering some future action in the application (as opposed to normal **synchronous** functions, which do everything they are going to do before returning). There are many styles of asynchronous interfaces:

- Callback argument
- Return a placeholder (*Future*, `Promise`, `Deferred`)
- Deliver to a queue
- Callback registry (e.g. POSIX signals)

Regardless of which type of interface is used, asynchronous functions *by definition* interact differently with their callers; there is no free way to make a synchronous function asynchronous in a way that is transparent to its callers (systems like gevent use lightweight threads to offer performance comparable to asynchronous systems, but they do not actually make things asynchronous).

Examples

Here is a sample synchronous function:

```
from tornado.httpclient import HTTPClient

def synchronous_fetch(url):
    http_client = HTTPClient()
    response = http_client.fetch(url)
    return response.body
```

And here is the same function rewritten to be asynchronous with a callback argument:

```
from tornado.httpclient import AsyncHTTPClient

def asynchronous_fetch(url, callback):
    http_client = AsyncHTTPClient()
    def handle_response(response):
        callback(response.body)
    http_client.fetch(url, callback=handle_response)
```

And again with a *Future* instead of a callback:

```
from tornado.concurrent import Future

def async_fetch_future(url):
    http_client = AsyncHTTPClient()
    my_future = Future()
    fetch_future = http_client.fetch(url)
    fetch_future.add_done_callback(
        lambda f: my_future.set_result(f.result()))
    return my_future
```

The raw *Future* version is more complex, but `Futures` are nonetheless recommended practice in Tornado because they have two major advantages. Error handling is more consistent since the *Future.result* method can simply raise an exception (as opposed to the ad-hoc error handling common in callback-oriented interfaces), and `Futures` lend themselves well to use with coroutines. Coroutines will be discussed in depth in the next section of this guide. Here is the coroutine version of our sample function, which is very similar to the original synchronous version:

```
from tornado import gen

@gen.coroutine
def fetch_coroutine(url):
    http_client = AsyncHTTPClient()
    response = yield http_client.fetch(url)
    raise gen.Return(response.body)
```

The statement `raise gen.Return(response.body)` is an artifact of Python 2, in which generators aren't allowed to return values. To overcome this, Tornado coroutines raise a special kind of exception called a *Return*. The coroutine catches this exception and treats it like a returned value. In Python 3.3 and later, a `return response.body` achieves the same result.

4.1.3 Coroutines

Coroutines are the recommended way to write asynchronous code in Tornado. Coroutines use the Python `yield` keyword to suspend and resume execution instead of a chain of callbacks (cooperative lightweight threads as seen in frameworks like gevent are sometimes called coroutines as well, but in Tornado all coroutines use explicit context switches and are called as asynchronous functions).

Coroutines are almost as simple as synchronous code, but without the expense of a thread. They also make concurrency easier to reason about by reducing the number of places where a context switch can happen.

Example:

```
from tornado import gen

@gen.coroutine
def fetch_coroutine(url):
    http_client = AsyncHTTPClient()
    response = yield http_client.fetch(url)
    # In Python versions prior to 3.3, returning a value from
    # a generator is not allowed and you must use
    #   raise gen.Return(response.body)
    # instead.
    return response.body
```

Python 3.5: `async` and `await`

Python 3.5 introduces the `async` and `await` keywords (functions using these keywords are also called "native coroutines"). Starting in Tornado 4.3, you can use them in place of `yield`-based coroutines. Simply use `async def foo()` in place of a function definition with the `@gen.coroutine` decorator, and `await` in place of yield. The rest of this document still uses the `yield` style for compatibility with older versions of Python, but `async` and `await` will run faster when they are available:

```
async def fetch_coroutine(url):
    http_client = AsyncHTTPClient()
    response = await http_client.fetch(url)
    return response.body
```

The `await` keyword is less versatile than the `yield` keyword. For example, in a `yield`-based coroutine you can yield a list of `Futures`, while in a native coroutine you must wrap the list in *tornado.gen.multi*. You can also use *tornado.gen.convert_yielded* to convert anything that would work with `yield` into a form that will work with `await`.

While native coroutines are not visibly tied to a particular framework (i.e. they do not use a decorator like *tornado.gen.coroutine* or `asyncio.coroutine`), not all coroutines are compatible with each other. There is a *coroutine runner* which is selected by the first coroutine to be called, and then shared by all coroutines which are called directly with `await`. The Tornado coroutine runner is designed to be versatile and accept awaitable objects from any framework; other coroutine runners may be more limited (for example, the `asyncio` coroutine runner does not accept coroutines from other frameworks). For this reason, it is recommended to use the Tornado coroutine runner for any application which combines multiple frameworks. To call a coroutine using the Tornado runner from within a coroutine that is already using the asyncio runner, use the *tornado.platform.asyncio.to_asyncio_future* adapter.

How it works

A function containing `yield` is a **generator**. All generators are asynchronous; when called they return a generator object instead of running to completion. The `@gen.coroutine` decorator communicates with the generator via the `yield` expressions, and with the coroutine's caller by returning a *Future*.

Here is a simplified version of the coroutine decorator's inner loop:

```
# Simplified inner loop of tornado.gen.Runner
def run(self):
    # send(x) makes the current yield return x.
    # It returns when the next yield is reached
    future = self.gen.send(self.next)
    def callback(f):
        self.next = f.result()
        self.run()
    future.add_done_callback(callback)
```

The decorator receives a *Future* from the generator, waits (without blocking) for that *Future* to complete, then "unwraps" the *Future* and sends the result back into the generator as the result of the `yield` expression. Most asynchronous code never touches the *Future* class directly except to immediately pass the *Future* returned by an asynchronous function to a `yield` expression.

How to call a coroutine

Coroutines do not raise exceptions in the normal way: any exception they raise will be trapped in the *Future* until it is yielded. This means it is important to call coroutines in the right way, or you may have errors that go unnoticed:

```
@gen.coroutine
def divide(x, y):
    return x / y

def bad_call():
    # This should raise a ZeroDivisionError, but it won't because
    # the coroutine is called incorrectly.
    divide(1, 0)
```

In nearly all cases, any function that calls a coroutine must be a coroutine itself, and use the `yield` keyword in the call. When you are overriding a method defined in a superclass, consult the documentation to see if coroutines are allowed (the documentation should say that the method "may be a coroutine" or "may return a *Future*"):

```
@gen.coroutine
def good_call():
    # yield will unwrap the Future returned by divide() and raise
    # the exception.
    yield divide(1, 0)
```

Sometimes you may want to "fire and forget" a coroutine without waiting for its result. In this case it is recommended to use *IOLoop.spawn_callback*, which makes the *IOLoop* responsible for the call. If it fails, the *IOLoop* will log a stack trace:

```
# The IOLoop will catch the exception and print a stack trace in
# the logs. Note that this doesn't look like a normal call, since
# we pass the function object to be called by the IOLoop.
IOLoop.current().spawn_callback(divide, 1, 0)
```

Finally, at the top level of a program, *if the 'IOLoop' is not yet running,* you can start the *IOLoop*, run the coroutine, and then stop the *IOLoop* with the *IOLoop.run_sync* method. This is often used to start the `main` function of a batch-oriented program:

```
# run_sync() doesn't take arguments, so we must wrap the
# call in a lambda.
IOLoop.current().run_sync(lambda: divide(1, 0))
```

Coroutine patterns

Interaction with callbacks

To interact with asynchronous code that uses callbacks instead of *Future*, wrap the call in a *Task*. This will add the callback argument for you and return a *Future* which you can yield:

```
@gen.coroutine
def call_task():
    # Note that there are no parens on some_function.
    # This will be translated by Task into
    #   some_function(other_args, callback=callback)
    yield gen.Task(some_function, other_args)
```

Calling blocking functions

The simplest way to call a blocking function from a coroutine is to use a `ThreadPoolExecutor`, which returns `Futures` that are compatible with coroutines:

```
thread_pool = ThreadPoolExecutor(4)

@gen.coroutine
def call_blocking():
    yield thread_pool.submit(blocking_func, args)
```

Parallelism

The coroutine decorator recognizes lists and dicts whose values are `Futures`, and waits for all of those `Futures` in parallel:

```
@gen.coroutine
def parallel_fetch(url1, url2):
    resp1, resp2 = yield [http_client.fetch(url1),
                          http_client.fetch(url2)]

@gen.coroutine
def parallel_fetch_many(urls):
    responses = yield [http_client.fetch(url) for url in urls]
    # responses is a list of HTTPResponses in the same order

@gen.coroutine
def parallel_fetch_dict(urls):
    responses = yield {url: http_client.fetch(url)
                       for url in urls}
    # responses is a dict {url: HTTPResponse}
```

Interleaving

Sometimes it is useful to save a *`Future`* instead of yielding it immediately, so you can start another operation before waiting:

```
@gen.coroutine
def get(self):
    fetch_future = self.fetch_next_chunk()
    while True:
        chunk = yield fetch_future
        if chunk is None: break
        self.write(chunk)
        fetch_future = self.fetch_next_chunk()
        yield self.flush()
```

Looping

Looping is tricky with coroutines since there is no way in Python to `yield` on every iteration of a `for` or `while` loop and capture the result of the yield. Instead, you'll need to separate the loop condition from accessing the results, as in this example from Motor:

```
import motor
db = motor.MotorClient().test

@gen.coroutine
def loop_example(collection):
    cursor = db.collection.find()
```

```
    while (yield cursor.fetch_next):
        doc = cursor.next_object()
```

Running in the background

PeriodicCallback is not normally used with coroutines. Instead, a coroutine can contain a `while True:` loop and use *tornado.gen.sleep*:

```
@gen.coroutine
def minute_loop():
    while True:
        yield do_something()
        yield gen.sleep(60)

# Coroutines that loop forever are generally started with
# spawn_callback().
IOLoop.current().spawn_callback(minute_loop)
```

Sometimes a more complicated loop may be desirable. For example, the previous loop runs every `60+N` seconds, where `N` is the running time of `do_something()`. To run exactly every 60 seconds, use the interleaving pattern from above:

```
@gen.coroutine
def minute_loop2():
    while True:
        nxt = gen.sleep(60)   # Start the clock.
        yield do_something()  # Run while the clock is ticking.
        yield nxt             # Wait for the timer to run out.
```

4.1.4 `Queue` example - a concurrent web spider

Tornado's *tornado.queues* module implements an asynchronous producer / consumer pattern for coroutines, analogous to the pattern implemented for threads by the Python standard library's `queue` module.

A coroutine that yields *Queue.get* pauses until there is an item in the queue. If the queue has a maximum size set, a coroutine that yields *Queue.put* pauses until there is room for another item.

A *Queue* maintains a count of unfinished tasks, which begins at zero. *put* increments the count; *task_done* decrements it.

In the web-spider example here, the queue begins containing only base_url. When a worker fetches a page it parses the links and puts new ones in the queue, then calls *task_done* to decrement the counter once. Eventually, a worker fetches a page whose URLs have all been seen before, and there is also no work left in the queue. Thus that worker's call to *task_done* decrements the counter to zero. The main coroutine, which is waiting for *join*, is unpaused and finishes.

```
#!/usr/bin/env python

import time
from datetime import timedelta

try:
    from HTMLParser import HTMLParser
    from urlparse import urljoin, urldefrag
except ImportError:
    from html.parser import HTMLParser
```

```
    from urllib.parse import urljoin, urldefrag

from tornado import httpclient, gen, ioloop, queues

base_url = 'http://www.tornadoweb.org/en/stable/'
concurrency = 10


@gen.coroutine
def get_links_from_url(url):
    """Download the page at `url` and parse it for links.

    Returned links have had the fragment after `#` removed, and have been made
    absolute so, e.g. the URL 'gen.html#tornado.gen.coroutine' becomes
    'http://www.tornadoweb.org/en/stable/gen.html'.
    """
    try:
        response = yield httpclient.AsyncHTTPClient().fetch(url)
        print('fetched %s' % url)

        html = response.body if isinstance(response.body, str) \
            else response.body.decode()
        urls = [urljoin(url, remove_fragment(new_url))
                for new_url in get_links(html)]
    except Exception as e:
        print('Exception: %s %s' % (e, url))
        raise gen.Return([])

    raise gen.Return(urls)


def remove_fragment(url):
    pure_url, frag = urldefrag(url)
    return pure_url


def get_links(html):
    class URLSeeker(HTMLParser):
        def __init__(self):
            HTMLParser.__init__(self)
            self.urls = []

        def handle_starttag(self, tag, attrs):
            href = dict(attrs).get('href')
            if href and tag == 'a':
                self.urls.append(href)

    url_seeker = URLSeeker()
    url_seeker.feed(html)
    return url_seeker.urls


@gen.coroutine
def main():
    q = queues.Queue()
    start = time.time()
    fetching, fetched = set(), set()
```

```
    @gen.coroutine
    def fetch_url():
        current_url = yield q.get()
        try:
            if current_url in fetching:
                return

            print('fetching %s' % current_url)
            fetching.add(current_url)
            urls = yield get_links_from_url(current_url)
            fetched.add(current_url)

            for new_url in urls:
                # Only follow links beneath the base URL
                if new_url.startswith(base_url):
                    yield q.put(new_url)

        finally:
            q.task_done()

    @gen.coroutine
    def worker():
        while True:
            yield fetch_url()

    q.put(base_url)

    # Start workers, then wait for the work queue to be empty.
    for _ in range(concurrency):
        worker()
    yield q.join(timeout=timedelta(seconds=300))
    assert fetching == fetched
    print('Done in %d seconds, fetched %s URLs.' % (
        time.time() - start, len(fetched)))


if __name__ == '__main__':
    import logging
    logging.basicConfig()
    io_loop = ioloop.IOLoop.current()
    io_loop.run_sync(main)
```

4.1.5 Structure of a Tornado web application

A Tornado web application generally consists of one or more *RequestHandler* subclasses, an *Application* object which routes incoming requests to handlers, and a `main()` function to start the server.

A minimal "hello world" example looks something like this:

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

def make_app():
```

```
    return tornado.web.Application([
        (r"/", MainHandler),
    ])

if __name__ == "__main__":
    app = make_app()
    app.listen(8888)
    tornado.ioloop.IOLoop.current().start()
```

The `Application` object

The *`Application`* object is responsible for global configuration, including the routing table that maps requests to handlers.

The routing table is a list of *`URLSpec`* objects (or tuples), each of which contains (at least) a regular expression and a handler class. Order matters; the first matching rule is used. If the regular expression contains capturing groups, these groups are the *path arguments* and will be passed to the handler's HTTP method. If a dictionary is passed as the third element of the *`URLSpec`*, it supplies the *initialization arguments* which will be passed to *`RequestHandler.initialize`*. Finally, the *`URLSpec`* may have a name, which will allow it to be used with *`RequestHandler.reverse_url`*.

For example, in this fragment the root URL `/` is mapped to `MainHandler` and URLs of the form `/story/` followed by a number are mapped to `StoryHandler`. That number is passed (as a string) to `StoryHandler.get`.

```
class MainHandler(RequestHandler):
    def get(self):
        self.write('<a href="%s">link to story 1</a>' %
                   self.reverse_url("story", "1"))

class StoryHandler(RequestHandler):
    def initialize(self, db):
        self.db = db

    def get(self, story_id):
        self.write("this is story %s" % story_id)

app = Application([
    url(r"/", MainHandler),
    url(r"/story/([0-9]+)", StoryHandler, dict(db=db), name="story")
    ])
```

The *`Application`* constructor takes many keyword arguments that can be used to customize the behavior of the application and enable optional features; see *`Application.settings`* for the complete list.

Subclassing `RequestHandler`

Most of the work of a Tornado web application is done in subclasses of *`RequestHandler`*. The main entry point for a handler subclass is a method named after the HTTP method being handled: `get()`, `post()`, etc. Each handler may define one or more of these methods to handle different HTTP actions. As described above, these methods will be called with arguments corresponding to the capturing groups of the routing rule that matched.

Within a handler, call methods such as *`RequestHandler.render`* or *`RequestHandler.write`* to produce a response. `render()` loads a *`Template`* by name and renders it with the given arguments. `write()` is used for non-template-based output; it accepts strings, bytes, and dictionaries (dicts will be encoded as JSON).

Many methods in *RequestHandler* are designed to be overridden in subclasses and be used throughout the application. It is common to define a `BaseHandler` class that overrides methods such as *write_error* and *get_current_user* and then subclass your own `BaseHandler` instead of *RequestHandler* for all your specific handlers.

Handling request input

The request handler can access the object representing the current request with `self.request`. See the class definition for *HTTPServerRequest* for a complete list of attributes.

Request data in the formats used by HTML forms will be parsed for you and is made available in methods like *get_query_argument* and *get_body_argument*.

```
class MyFormHandler(tornado.web.RequestHandler):
    def get(self):
        self.write('<html><body><form action="/myform" method="POST">'
                   '<input type="text" name="message">'
                   '<input type="submit" value="Submit">'
                   '</form></body></html>')

    def post(self):
        self.set_header("Content-Type", "text/plain")
        self.write("You wrote " + self.get_body_argument("message"))
```

Since the HTML form encoding is ambiguous as to whether an argument is a single value or a list with one element, *RequestHandler* has distinct methods to allow the application to indicate whether or not it expects a list. For lists, use *get_query_arguments* and *get_body_arguments* instead of their singular counterparts.

Files uploaded via a form are available in `self.request.files`, which maps names (the name of the HTML `<input type="file">` element) to a list of files. Each file is a dictionary of the form `{"filename":..., "content_type":..., "body":...}`. The `files` object is only present if the files were uploaded with a form wrapper (i.e. a `multipart/form-data` Content-Type); if this format was not used the raw uploaded data is available in `self.request.body`. By default uploaded files are fully buffered in memory; if you need to handle files that are too large to comfortably keep in memory see the *stream_request_body* class decorator.

Due to the quirks of the HTML form encoding (e.g. the ambiguity around singular versus plural arguments), Tornado does not attempt to unify form arguments with other types of input. In particular, we do not parse JSON request bodies. Applications that wish to use JSON instead of form-encoding may override *prepare* to parse their requests:

```
def prepare(self):
    if self.request.headers["Content-Type"].startswith("application/json"):
        self.json_args = json.loads(self.request.body)
    else:
        self.json_args = None
```

Overriding RequestHandler methods

In addition to `get()`/`post()`/etc, certain other methods in *RequestHandler* are designed to be overridden by subclasses when necessary. On every request, the following sequence of calls takes place:

1. A new *RequestHandler* object is created on each request
2. *initialize()* is called with the initialization arguments from the *Application* configuration. `initialize` should typically just save the arguments passed into member variables; it may not produce any output or call methods like *send_error*.

3. *prepare()* is called. This is most useful in a base class shared by all of your handler subclasses, as `prepare` is called no matter which HTTP method is used. `prepare` may produce output; if it calls *finish* (or `redirect`, etc), processing stops here.
4. One of the HTTP methods is called: `get()`, `post()`, `put()`, etc. If the URL regular expression contains capturing groups, they are passed as arguments to this method.
5. When the request is finished, *on_finish()* is called. For synchronous handlers this is immediately after `get()` (etc) return; for asynchronous handlers it is after the call to *finish()*.

All methods designed to be overridden are noted as such in the *RequestHandler* documentation. Some of the most commonly overridden methods include:

- *write_error* - outputs HTML for use on error pages.
- *on_connection_close* - called when the client disconnects; applications may choose to detect this case and halt further processing. Note that there is no guarantee that a closed connection can be detected promptly.
- *get_current_user* - see *User authentication*
- *get_user_locale* - returns *Locale* object to use for the current user
- *set_default_headers* - may be used to set additional headers on the response (such as a custom `Server` header)

Error Handling

If a handler raises an exception, Tornado will call *RequestHandler.write_error* to generate an error page. *tornado.web.HTTPError* can be used to generate a specified status code; all other exceptions return a 500 status.

The default error page includes a stack trace in debug mode and a one-line description of the error (e.g. "500: Internal Server Error") otherwise. To produce a custom error page, override *RequestHandler.write_error* (probably in a base class shared by all your handlers). This method may produce output normally via methods such as *write* and *render*. If the error was caused by an exception, an `exc_info` triple will be passed as a keyword argument (note that this exception is not guaranteed to be the current exception in `sys.exc_info`, so `write_error` must use e.g. `traceback.format_exception` instead of `traceback.format_exc`).

It is also possible to generate an error page from regular handler methods instead of `write_error` by calling *set_status*, writing a response, and returning. The special exception *tornado.web.Finish* may be raised to terminate the handler without calling `write_error` in situations where simply returning is not convenient.

For 404 errors, use the `default_handler_class` *Application setting*. This handler should override *prepare* instead of a more specific method like `get()` so it works with any HTTP method. It should produce its error page as described above: either by raising a `HTTPError(404)` and overriding `write_error`, or calling `self.set_status(404)` and producing the response directly in `prepare()`.

Redirection

There are two main ways you can redirect requests in Tornado: *RequestHandler.redirect* and with the *RedirectHandler*.

You can use `self.redirect()` within a *RequestHandler* method to redirect users elsewhere. There is also an optional parameter `permanent` which you can use to indicate that the redirection is considered permanent. The default value of `permanent` is `False`, which generates a `302 Found` HTTP response code and is appropriate for things like redirecting users after successful `POST` requests. If `permanent` is true, the `301 Moved Permanently` HTTP response code is used, which is useful for e.g. redirecting to a canonical URL for a page in an SEO-friendly manner.

RedirectHandler lets you configure redirects directly in your *Application* routing table. For example, to configure a single static redirect:

```
app = tornado.web.Application([
    url(r"/app", tornado.web.RedirectHandler,
        dict(url="http://itunes.apple.com/my-app-id")),
    ])
```

RedirectHandler also supports regular expression substitutions. The following rule redirects all requests beginning with `/pictures/` to the prefix `/photos/` instead:

```
app = tornado.web.Application([
    url(r"/photos/(.*)", MyPhotoHandler),
    url(r"/pictures/(.*)", tornado.web.RedirectHandler,
        dict(url=r"/photos/\1")),
    ])
```

Unlike *RequestHandler.redirect*, *RedirectHandler* uses permanent redirects by default. This is because the routing table does not change at runtime and is presumed to be permanent, while redirects found in handlers are likely to be the result of other logic that may change. To send a temporary redirect with a *RedirectHandler*, add `permanent=False` to the *RedirectHandler* initialization arguments.

Asynchronous handlers

Tornado handlers are synchronous by default: when the `get()`/`post()` method returns, the request is considered finished and the response is sent. Since all other requests are blocked while one handler is running, any long-running handler should be made asynchronous so it can call its slow operations in a non-blocking way. This topic is covered in more detail in Asynchronous and non-Blocking I/O; this section is about the particulars of asynchronous techniques in *RequestHandler* subclasses.

The simplest way to make a handler asynchronous is to use the *coroutine* decorator. This allows you to perform non-blocking I/O with the `yield` keyword, and no response will be sent until the coroutine has returned. See Coroutines for more details.

In some cases, coroutines may be less convenient than a callback-oriented style, in which case the *tornado.web.asynchronous* decorator can be used instead. When this decorator is used the response is not automatically sent; instead the request will be kept open until some callback calls *RequestHandler.finish*. It is up to the application to ensure that this method is called, or else the user's browser will simply hang.

Here is an example that makes a call to the FriendFeed API using Tornado's built-in *AsyncHTTPClient*:

```
class MainHandler(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    def get(self):
        http = tornado.httpclient.AsyncHTTPClient()
        http.fetch("http://friendfeed-api.com/v2/feed/bret",
                   callback=self.on_response)

    def on_response(self, response):
        if response.error: raise tornado.web.HTTPError(500)
        json = tornado.escape.json_decode(response.body)
        self.write("Fetched " + str(len(json["entries"])) + " entries "
                   "from the FriendFeed API")
        self.finish()
```

When `get()` returns, the request has not finished. When the HTTP client eventually calls `on_response()`, the request is still open, and the response is finally flushed to the client with the call to `self.finish()`.

For comparison, here is the same example using a coroutine:

```
class MainHandler(tornado.web.RequestHandler):
    @tornado.gen.coroutine
    def get(self):
        http = tornado.httpclient.AsyncHTTPClient()
        response = yield http.fetch("http://friendfeed-api.com/v2/feed/bret")
        json = tornado.escape.json_decode(response.body)
        self.write("Fetched " + str(len(json["entries"])) + " entries "
                   "from the FriendFeed API")
```

For a more advanced asynchronous example, take a look at the chat example application, which implements an AJAX chat room using long polling. Users of long polling may want to override `on_connection_close()` to clean up after the client closes the connection (but see that method's docstring for caveats).

4.1.6 Templates and UI

Tornado includes a simple, fast, and flexible templating language. This section describes that language as well as related issues such as internationalization.

Tornado can also be used with any other Python template language, although there is no provision for integrating these systems into *`RequestHandler.render`*. Simply render the template to a string and pass it to *`RequestHandler.write`*

Configuring templates

By default, Tornado looks for template files in the same directory as the `.py` files that refer to them. To put your template files in a different directory, use the `template_path` *`Application setting`* (or override *`RequestHandler.get_template_path`* if you have different template paths for different handlers).

To load templates from a non-filesystem location, subclass *`tornado.template.BaseLoader`* and pass an instance as the `template_loader` application setting.

Compiled templates are cached by default; to turn off this caching and reload templates so changes to the underlying files are always visible, use the application settings `compiled_template_cache=False` or `debug=True`.

Template syntax

A Tornado template is just HTML (or any other text-based format) with Python control sequences and expressions embedded within the markup:

```
<html>
   <head>
      <title>{{ title }}</title>
   </head>
   <body>
     <ul>
       {% for item in items %}
         <li>{{ escape(item) }}</li>
       {% end %}
     </ul>
   </body>
 </html>
```

If you saved this template as "template.html" and put it in the same directory as your Python file, you could render this template with:

```
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        items = ["Item 1", "Item 2", "Item 3"]
        self.render("template.html", title="My title", items=items)
```

Tornado templates support *control statements* and *expressions*. Control statements are surrounded by `{%` and `%}`, e.g., `{% if len(items) > 2 %}`. Expressions are surrounded by `{{` and `}}`, e.g., `{{ items[0] }}`.

Control statements more or less map exactly to Python statements. We support `if`, `for`, `while`, and `try`, all of which are terminated with `{% end %}`. We also support *template inheritance* using the `extends` and `block` statements, which are described in detail in the documentation for the *tornado.template*.

Expressions can be any Python expression, including function calls. Template code is executed in a namespace that includes the following objects and functions (Note that this list applies to templates rendered using *RequestHandler.render* and *render_string*. If you're using the *tornado.template* module directly outside of a *RequestHandler* many of these entries are not present).

- `escape`: alias for *tornado.escape.xhtml_escape*
- `xhtml_escape`: alias for *tornado.escape.xhtml_escape*
- `url_escape`: alias for *tornado.escape.url_escape*
- `json_encode`: alias for *tornado.escape.json_encode*
- `squeeze`: alias for *tornado.escape.squeeze*
- `linkify`: alias for *tornado.escape.linkify*
- `datetime`: the Python `datetime` module
- `handler`: the current *RequestHandler* object
- `request`: alias for *handler.request*
- `current_user`: alias for *handler.current_user*
- `locale`: alias for *handler.locale*
- _: alias for *handler.locale.translate*
- `static_url`: alias for *handler.static_url*
- `xsrf_form_html`: alias for *handler.xsrf_form_html*
- `reverse_url`: alias for *Application.reverse_url*
- All entries from the `ui_methods` and `ui_modules` `Application` settings
- Any keyword arguments passed to *render* or *render_string*

When you are building a real application, you are going to want to use all of the features of Tornado templates, especially template inheritance. Read all about those features in the *tornado.template* section (some features, including `UIModules` are implemented in the *tornado.web* module)

Under the hood, Tornado templates are translated directly to Python. The expressions you include in your template are copied verbatim into a Python function representing your template. We don't try to prevent anything in the template language; we created it explicitly to provide the flexibility that other, stricter templating systems prevent. Consequently, if you write random stuff inside of your template expressions, you will get random Python errors when you execute the template.

All template output is escaped by default, using the *tornado.escape.xhtml_escape* function. This behavior can be changed globally by passing `autoescape=None` to the *Application* or *tornado.template.Loader* constructors, for a template file with the `{% autoescape None %}` directive,

or for a single expression by replacing `{{ ... }}` with `{% raw ...%}`. Additionally, in each of these places the name of an alternative escaping function may be used instead of `None`.

Note that while Tornado's automatic escaping is helpful in avoiding XSS vulnerabilities, it is not sufficient in all cases. Expressions that appear in certain locations, such as in Javascript or CSS, may need additional escaping. Additionally, either care must be taken to always use double quotes and *xhtml_escape* in HTML attributes that may contain untrusted content, or a separate escaping function must be used for attributes (see e.g. http://wonko.com/post/html-escaping)

Internationalization

The locale of the current user (whether they are logged in or not) is always available as `self.locale` in the request handler and as `locale` in templates. The name of the locale (e.g., `en_US`) is available as `locale.name`, and you can translate strings with the *Locale.translate* method. Templates also have the global function call `_()` available for string translation. The translate function has two forms:

```
_("Translate this string")
```

which translates the string directly based on the current locale, and:

```
_("A person liked this", "%(num)d people liked this",
  len(people)) % {"num": len(people)}
```

which translates a string that can be singular or plural based on the value of the third argument. In the example above, a translation of the first string will be returned if `len(people)` is `1`, or a translation of the second string will be returned otherwise.

The most common pattern for translations is to use Python named placeholders for variables (the `%(num)d` in the example above) since placeholders can move around on translation.

Here is a properly internationalized template:

```
<html>
   <head>
      <title>FriendFeed - {{ _("Sign in") }}</title>
   </head>
   <body>
     <form action="{{ request.path }}" method="post">
       <div>{{ _("Username") }} <input type="text" name="username"/></div>
       <div>{{ _("Password") }} <input type="password" name="password"/></div>
       <div><input type="submit" value="{{ _("Sign in") }}"/></div>
       {% module xsrf_form_html() %}
     </form>
   </body>
 </html>
```

By default, we detect the user's locale using the `Accept-Language` header sent by the user's browser. We choose `en_US` if we can't find an appropriate `Accept-Language` value. If you let user's set their locale as a preference, you can override this default locale selection by overriding *RequestHandler.get_user_locale*:

```
class BaseHandler(tornado.web.RequestHandler):
    def get_current_user(self):
        user_id = self.get_secure_cookie("user")
        if not user_id: return None
        return self.backend.get_user_by_id(user_id)

    def get_user_locale(self):
        if "locale" not in self.current_user.prefs:
            # Use the Accept-Language header
```

```
            return None
        return self.current_user.prefs["locale"]
```

If `get_user_locale` returns `None`, we fall back on the `Accept-Language` header.

The *tornado.locale* module supports loading translations in two formats: the `.mo` format used by `gettext` and related tools, and a simple `.csv` format. An application will generally call either *tornado.locale.load_translations* or *tornado.locale.load_gettext_translations* once at startup; see those methods for more details on the supported formats..

You can get the list of supported locales in your application with *tornado.locale.get_supported_locales()*. The user's locale is chosen to be the closest match based on the supported locales. For example, if the user's locale is `es_GT`, and the `es` locale is supported, `self.locale` will be `es` for that request. We fall back on `en_US` if no close match can be found.

UI modules

Tornado supports *UI modules* to make it easy to support standard, reusable UI widgets across your application. UI modules are like special function calls to render components of your page, and they can come packaged with their own CSS and JavaScript.

For example, if you are implementing a blog, and you want to have blog entries appear on both the blog home page and on each blog entry page, you can make an `Entry` module to render them on both pages. First, create a Python module for your UI modules, e.g., `uimodules.py`:

```
class Entry(tornado.web.UIModule):
    def render(self, entry, show_comments=False):
        return self.render_string(
            "module-entry.html", entry=entry, show_comments=show_comments)
```

Tell Tornado to use `uimodules.py` using the `ui_modules` setting in your application:

```
from . import uimodules

class HomeHandler(tornado.web.RequestHandler):
    def get(self):
        entries = self.db.query("SELECT * FROM entries ORDER BY date DESC")
        self.render("home.html", entries=entries)

class EntryHandler(tornado.web.RequestHandler):
    def get(self, entry_id):
        entry = self.db.get("SELECT * FROM entries WHERE id = %s", entry_id)
        if not entry: raise tornado.web.HTTPError(404)
        self.render("entry.html", entry=entry)

settings = {
    "ui_modules": uimodules,
}
application = tornado.web.Application([
    (r"/", HomeHandler),
    (r"/entry/([0-9]+)", EntryHandler),
], **settings)
```

Within a template, you can call a module with the `{% module %}` statement. For example, you could call the `Entry` module from both `home.html`:

```
{% for entry in entries %}
  {% module Entry(entry) %}
{% end %}
```

and `entry.html`:

```
{% module Entry(entry, show_comments=True) %}
```

Modules can include custom CSS and JavaScript functions by overriding the `embedded_css`, `embedded_javascript`, `javascript_files`, or `css_files` methods:

```
class Entry(tornado.web.UIModule):
    def embedded_css(self):
        return ".entry { margin-bottom: 1em; }"

    def render(self, entry, show_comments=False):
        return self.render_string(
            "module-entry.html", show_comments=show_comments)
```

Module CSS and JavaScript will be included once no matter how many times a module is used on a page. CSS is always included in the `<head>` of the page, and JavaScript is always included just before the `</body>` tag at the end of the page.

When additional Python code is not required, a template file itself may be used as a module. For example, the preceding example could be rewritten to put the following in `module-entry.html`:

```
{{ set_resources(embedded_css=".entry { margin-bottom: 1em; }") }}
<!-- more template html... -->
```

This revised template module would be invoked with:

```
{% module Template("module-entry.html", show_comments=True) %}
```

The `set_resources` function is only available in templates invoked via `{% module Template(...) %}`. Unlike the `{% include ... %}` directive, template modules have a distinct namespace from their containing template - they can only see the global template namespace and their own keyword arguments.

4.1.7 Authentication and security

Cookies and secure cookies

You can set cookies in the user's browser with the `set_cookie` method:

```
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        if not self.get_cookie("mycookie"):
            self.set_cookie("mycookie", "myvalue")
            self.write("Your cookie was not set yet!")
        else:
            self.write("Your cookie was set!")
```

Cookies are not secure and can easily be modified by clients. If you need to set cookies to, e.g., identify the currently logged in user, you need to sign your cookies to prevent forgery. Tornado supports signed cookies with the *`set_secure_cookie`* and *`get_secure_cookie`* methods. To use these methods, you need to specify a secret key named `cookie_secret` when you create your application. You can pass in application settings as keyword arguments to your application:

```
application = tornado.web.Application([
    (r"/", MainHandler),
], cookie_secret="__TODO:_GENERATE_YOUR_OWN_RANDOM_VALUE_HERE__")
```

Signed cookies contain the encoded value of the cookie in addition to a timestamp and an HMAC signature. If the cookie is old or if the signature doesn't match, `get_secure_cookie` will return `None` just as if the cookie isn't set. The secure version of the example above:

```
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        if not self.get_secure_cookie("mycookie"):
            self.set_secure_cookie("mycookie", "myvalue")
            self.write("Your cookie was not set yet!")
        else:
            self.write("Your cookie was set!")
```

Tornado's secure cookies guarantee integrity but not confidentiality. That is, the cookie cannot be modified but its contents can be seen by the user. The `cookie_secret` is a symmetric key and must be kept secret – anyone who obtains the value of this key could produce their own signed cookies.

By default, Tornado's secure cookies expire after 30 days. To change this, use the `expires_days` keyword argument to `set_secure_cookie` *and* the `max_age_days` argument to `get_secure_cookie`. These two values are passed separately so that you may e.g. have a cookie that is valid for 30 days for most purposes, but for certain sensitive actions (such as changing billing information) you use a smaller `max_age_days` when reading the cookie.

Tornado also supports multiple signing keys to enable signing key rotation. `cookie_secret` then must be a dict with integer key versions as keys and the corresponding secrets as values. The currently used signing key must then be set as `key_version` application setting but all other keys in the dict are allowed for cookie signature validation, if the correct key version is set in the cookie. To implement cookie updates, the current signing key version can be queried via *`get_secure_cookie_key_version`*.

User authentication

The currently authenticated user is available in every request handler as *`self.current_user`*, and in every template as `current_user`. By default, `current_user` is `None`.

To implement user authentication in your application, you need to override the `get_current_user()` method in your request handlers to determine the current user based on, e.g., the value of a cookie. Here is an example that lets users log into the application simply by specifying a nickname, which is then saved in a cookie:

```
class BaseHandler(tornado.web.RequestHandler):
    def get_current_user(self):
        return self.get_secure_cookie("user")

class MainHandler(BaseHandler):
    def get(self):
        if not self.current_user:
            self.redirect("/login")
            return
        name = tornado.escape.xhtml_escape(self.current_user)
        self.write("Hello, " + name)

class LoginHandler(BaseHandler):
    def get(self):
        self.write('<html><body><form action="/login" method="post">'
                   'Name: <input type="text" name="name">'
                   '<input type="submit" value="Sign in">'
```

```
                    '</form></body></html>')

    def post(self):
        self.set_secure_cookie("user", self.get_argument("name"))
        self.redirect("/")

application = tornado.web.Application([
    (r"/", MainHandler),
    (r"/login", LoginHandler),
], cookie_secret="__TODO:_GENERATE_YOUR_OWN_RANDOM_VALUE_HERE__")
```

You can require that the user be logged in using the Python decorator *tornado.web.authenticated*. If a request goes to a method with this decorator, and the user is not logged in, they will be redirected to `login_url` (another application setting). The example above could be rewritten:

```
class MainHandler(BaseHandler):
    @tornado.web.authenticated
    def get(self):
        name = tornado.escape.xhtml_escape(self.current_user)
        self.write("Hello, " + name)

settings = {
    "cookie_secret": "__TODO:_GENERATE_YOUR_OWN_RANDOM_VALUE_HERE__",
    "login_url": "/login",
}
application = tornado.web.Application([
    (r"/", MainHandler),
    (r"/login", LoginHandler),
], **settings)
```

If you decorate `post()` methods with the `authenticated` decorator, and the user is not logged in, the server will send a `403` response. The `@authenticated` decorator is simply shorthand for `if not self.current_user: self.redirect()` and may not be appropriate for non-browser-based login schemes.

Check out the Tornado Blog example application for a complete example that uses authentication (and stores user data in a MySQL database).

Third party authentication

The *tornado.auth* module implements the authentication and authorization protocols for a number of the most popular sites on the web, including Google/Gmail, Facebook, Twitter, and FriendFeed. The module includes methods to log users in via these sites and, where applicable, methods to authorize access to the service so you can, e.g., download a user's address book or publish a Twitter message on their behalf.

Here is an example handler that uses Google for authentication, saving the Google credentials in a cookie for later access:

```
class GoogleOAuth2LoginHandler(tornado.web.RequestHandler,
                               tornado.auth.GoogleOAuth2Mixin):
    @tornado.gen.coroutine
    def get(self):
        if self.get_argument('code', False):
            user = yield self.get_authenticated_user(
                redirect_uri='http://your.site.com/auth/google',
                code=self.get_argument('code'))
            # Save the user with e.g. set_secure_cookie
```

```
        else:
            yield self.authorize_redirect(
                redirect_uri='http://your.site.com/auth/google',
                client_id=self.settings['google_oauth']['key'],
                scope=['profile', 'email'],
                response_type='code',
                extra_params={'approval_prompt': 'auto'})
```

See the *tornado.auth* module documentation for more details.

Cross-site request forgery protection

Cross-site request forgery, or XSRF, is a common problem for personalized web applications. See the Wikipedia article for more information on how XSRF works.

The generally accepted solution to prevent XSRF is to cookie every user with an unpredictable value and include that value as an additional argument with every form submission on your site. If the cookie and the value in the form submission do not match, then the request is likely forged.

Tornado comes with built-in XSRF protection. To include it in your site, include the application setting `xsrf_cookies`:

```
settings = {
    "cookie_secret": "__TODO:_GENERATE_YOUR_OWN_RANDOM_VALUE_HERE__",
    "login_url": "/login",
    "xsrf_cookies": True,
}
application = tornado.web.Application([
    (r"/", MainHandler),
    (r"/login", LoginHandler),
], **settings)
```

If `xsrf_cookies` is set, the Tornado web application will set the `_xsrf` cookie for all users and reject all `POST`, `PUT`, and `DELETE` requests that do not contain a correct `_xsrf` value. If you turn this setting on, you need to instrument all forms that submit via `POST` to contain this field. You can do this with the special *UIModule* `xsrf_form_html()`, available in all templates:

```
<form action="/new_message" method="post">
  {% module xsrf_form_html() %}
  <input type="text" name="message"/>
  <input type="submit" value="Post"/>
</form>
```

If you submit AJAX `POST` requests, you will also need to instrument your JavaScript to include the `_xsrf` value with each request. This is the jQuery function we use at FriendFeed for AJAX `POST` requests that automatically adds the `_xsrf` value to all requests:

```
function getCookie(name) {
    var r = document.cookie.match("\\b" + name + "=([^;]*)\\b");
    return r ? r[1] : undefined;
}

jQuery.postJSON = function(url, args, callback) {
    args._xsrf = getCookie("_xsrf");
    $.ajax({url: url, data: $.param(args), dataType: "text", type: "POST",
        success: function(response) {
        callback(eval("(" + response + ")"));
```

```
    }});
};
```

For `PUT` and `DELETE` requests (as well as `POST` requests that do not use form-encoded arguments), the XSRF token may also be passed via an HTTP header named `X-XSRFToken`. The XSRF cookie is normally set when `xsrf_form_html` is used, but in a pure-Javascript application that does not use any regular forms you may need to access `self.xsrf_token` manually (just reading the property is enough to set the cookie as a side effect).

If you need to customize XSRF behavior on a per-handler basis, you can override *RequestHandler.check_xsrf_cookie()*. For example, if you have an API whose authentication does not use cookies, you may want to disable XSRF protection by making `check_xsrf_cookie()` do nothing. However, if you support both cookie and non-cookie-based authentication, it is important that XSRF protection be used whenever the current request is authenticated with a cookie.

4.1.8 Running and deploying

Since Tornado supplies its own HTTPServer, running and deploying it is a little different from other Python web frameworks. Instead of configuring a WSGI container to find your application, you write a `main()` function that starts the server:

```
def main():
    app = make_app()
    app.listen(8888)
    IOLoop.current().start()

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

Configure your operating system or process manager to run this program to start the server. Please note that it may be necessary to increase the number of open files per process (to avoid "Too many open files"-Error). To raise this limit (setting it to 50000 for example) you can use the ulimit command, modify /etc/security/limits.conf or setting `minfds` in your supervisord config.

Processes and ports

Due to the Python GIL (Global Interpreter Lock), it is necessary to run multiple Python processes to take full advantage of multi-CPU machines. Typically it is best to run one process per CPU.

Tornado includes a built-in multi-process mode to start several processes at once. This requires a slight alteration to the standard main function:

```
def main():
    app = make_app()
    server = tornado.httpserver.HTTPServer(app)
    server.bind(8888)
    server.start(0)  # forks one process per cpu
    IOLoop.current().start()
```

This is the easiest way to start multiple processes and have them all share the same port, although it has some limitations. First, each child process will have its own IOLoop, so it is important that nothing touch the global IOLoop instance (even indirectly) before the fork. Second, it is difficult to do zero-downtime updates in this model. Finally, since all the processes share the same port it is more difficult to monitor them individually.

For more sophisticated deployments, it is recommended to start the processes independently, and have each one listen on a different port. The "process groups" feature of supervisord is one good way to arrange this. When each process

uses a different port, an external load balancer such as HAProxy or nginx is usually needed to present a single address to outside visitors.

Running behind a load balancer

When running behind a load balancer like nginx, it is recommended to pass `xheaders=True` to the *HTTPServer* constructor. This will tell Tornado to use headers like `X-Real-IP` to get the user's IP address instead of attributing all traffic to the balancer's IP address.

This is a barebones nginx config file that is structurally similar to the one we use at FriendFeed. It assumes nginx and the Tornado servers are running on the same machine, and the four Tornado servers are running on ports 8000 - 8003:

```
user nginx;
worker_processes 1;

error_log /var/log/nginx/error.log;
pid /var/run/nginx.pid;

events {
    worker_connections 1024;
    use epoll;
}

http {
    # Enumerate all the Tornado servers here
    upstream frontends {
        server 127.0.0.1:8000;
        server 127.0.0.1:8001;
        server 127.0.0.1:8002;
        server 127.0.0.1:8003;
    }

    include /etc/nginx/mime.types;
    default_type application/octet-stream;

    access_log /var/log/nginx/access.log;

    keepalive_timeout 65;
    proxy_read_timeout 200;
    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    gzip on;
    gzip_min_length 1000;
    gzip_proxied any;
    gzip_types text/plain text/html text/css text/xml
               application/x-javascript application/xml
               application/atom+xml text/javascript;

    # Only retry if there was a communication error, not a timeout
    # on the Tornado server (to avoid propagating "queries of death"
    # to all frontends)
    proxy_next_upstream error;

    server {
        listen 80;

        # Allow file uploads
```

```
        client_max_body_size 50M;

        location ^~ /static/ {
            root /var/www;
            if ($query_string) {
                expires max;
            }
        }
        location = /favicon.ico {
            rewrite (.*) /static/favicon.ico;
        }
        location = /robots.txt {
            rewrite (.*) /static/robots.txt;
        }

        location / {
            proxy_pass_header Server;
            proxy_set_header Host $http_host;
            proxy_redirect off;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Scheme $scheme;
            proxy_pass http://frontends;
        }
    }
}
```

Static files and aggressive file caching

You can serve static files from Tornado by specifying the `static_path` setting in your application:

```
settings = {
    "static_path": os.path.join(os.path.dirname(__file__), "static"),
    "cookie_secret": "__TODO:_GENERATE_YOUR_OWN_RANDOM_VALUE_HERE__",
    "login_url": "/login",
    "xsrf_cookies": True,
}
application = tornado.web.Application([
    (r"/", MainHandler),
    (r"/login", LoginHandler),
    (r"/(apple-touch-icon\.png)", tornado.web.StaticFileHandler,
     dict(path=settings['static_path'])),
], **settings)
```

This setting will automatically make all requests that start with `/static/` serve from that static directory, e.g., `http://localhost:8888/static/foo.png` will serve the file `foo.png` from the specified static directory. We also automatically serve `/robots.txt` and `/favicon.ico` from the static directory (even though they don't start with the `/static/` prefix).

In the above settings, we have explicitly configured Tornado to serve `apple-touch-icon.png` from the root with the *`StaticFileHandler`*, though it is physically in the static file directory. (The capturing group in that regular expression is necessary to tell *`StaticFileHandler`* the requested filename; recall that capturing groups are passed to handlers as method arguments.) You could do the same thing to serve e.g. `sitemap.xml` from the site root. Of course, you can also avoid faking a root `apple-touch-icon.png` by using the appropriate `<link />` tag in your HTML.

To improve performance, it is generally a good idea for browsers to cache static resources aggressively so browsers won't send unnecessary `If-Modified-Since` or `Etag` requests that might block the rendering of the page. Tor-

nado supports this out of the box with *static content versioning*.

To use this feature, use the *static_url* method in your templates rather than typing the URL of the static file directly in your HTML:

```
<html>
   <head>
      <title>FriendFeed - {{ _("Home") }}</title>
   </head>
   <body>
     <div><img src="{{ static_url("images/logo.png") }}"/></div>
   </body>
 </html>
```

The `static_url()` function will translate that relative path to a URI that looks like `/static/images/logo.png?v=aae54`. The `v` argument is a hash of the content in `logo.png`, and its presence makes the Tornado server send cache headers to the user's browser that will make the browser cache the content indefinitely.

Since the `v` argument is based on the content of the file, if you update a file and restart your server, it will start sending a new `v` value, so the user's browser will automatically fetch the new file. If the file's contents don't change, the browser will continue to use a locally cached copy without ever checking for updates on the server, significantly improving rendering performance.

In production, you probably want to serve static files from a more optimized static file server like nginx. You can configure most any web server to recognize the version tags used by `static_url()` and set caching headers accordingly. Here is the relevant portion of the nginx configuration we use at FriendFeed:

```
location /static/ {
    root /var/friendfeed/static;
    if ($query_string) {
        expires max;
    }
 }
```

Debug mode and automatic reloading

If you pass `debug=True` to the `Application` constructor, the app will be run in debug/development mode. In this mode, several features intended for convenience while developing will be enabled (each of which is also available as an individual flag; if both are specified the individual flag takes precedence):

- `autoreload=True`: The app will watch for changes to its source files and reload itself when anything changes. This reduces the need to manually restart the server during development. However, certain failures (such as syntax errors at import time) can still take the server down in a way that debug mode cannot currently recover from.
- `compiled_template_cache=False`: Templates will not be cached.
- `static_hash_cache=False`: Static file hashes (used by the `static_url` function) will not be cached
- `serve_traceback=True`: When an exception in a *RequestHandler* is not caught, an error page including a stack trace will be generated.

Autoreload mode is not compatible with the multi-process mode of *HTTPServer*. You must not give *HTTPServer.start* an argument other than 1 (or call *tornado.process.fork_processes*) if you are using autoreload mode.

The automatic reloading feature of debug mode is available as a standalone module in *tornado.autoreload*. The two can be used in combination to provide extra robustness against syntax errors: set `autoreload=True` within the

app to detect changes while it is running, and start it with `python -m tornado.autoreload myserver.py` to catch any syntax errors or other errors at startup.

Reloading loses any Python interpreter command-line arguments (e.g. `-u`) because it re-executes Python using `sys.executable` and `sys.argv`. Additionally, modifying these variables will cause reloading to behave incorrectly.

On some platforms (including Windows and Mac OSX prior to 10.6), the process cannot be updated "in-place", so when a code change is detected the old server exits and a new one starts. This has been known to confuse some IDEs.

WSGI and Google App Engine

Tornado is normally intended to be run on its own, without a WSGI container. However, in some environments (such as Google App Engine), only WSGI is allowed and applications cannot run their own servers. In this case Tornado supports a limited mode of operation that does not support asynchronous operation but allows a subset of Tornado's functionality in a WSGI-only environment. The features that are not allowed in WSGI mode include coroutines, the `@asynchronous` decorator, *`AsyncHTTPClient`*, the `auth` module, and WebSockets.

You can convert a Tornado *`Application`* to a WSGI application with *`tornado.wsgi.WSGIAdapter`*. In this example, configure your WSGI container to find the `application` object:

```
import tornado.web
import tornado.wsgi

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

tornado_app = tornado.web.Application([
    (r"/", MainHandler),
])
application = tornado.wsgi.WSGIAdapter(tornado_app)
```

See the appengine example application for a full-featured AppEngine app built on Tornado.

4.2 Web framework

4.2.1 `tornado.web` — `RequestHandler` and `Application` classes

`tornado.web` provides a simple web framework with asynchronous features that allow it to scale to large numbers of open connections, making it ideal for long polling.

Here is a simple "Hello, world" example app:

```
import tornado.ioloop
import tornado.web

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

if __name__ == "__main__":
    application = tornado.web.Application([
        (r"/", MainHandler),
    ])
```

```
    application.listen(8888)
    tornado.ioloop.IOLoop.current().start()
```

See the User's guide for additional information.

Thread-safety notes

In general, methods on *RequestHandler* and elsewhere in Tornado are not thread-safe. In particular, methods such as *write()*, *finish()*, and *flush()* must only be called from the main thread. If you use multiple threads it is important to use *IOLoop.add_callback* to transfer control back to the main thread before finishing the request.

Request handlers

class tornado.web.**RequestHandler**(*application*, *request*, ***kwargs*)
Base class for HTTP request handlers.

Subclasses must define at least one of the methods defined in the "Entry points" section below.

Entry points

RequestHandler.**initialize**()
Hook for subclass initialization. Called for each request.

A dictionary passed as the third argument of a url spec will be supplied as keyword arguments to initialize().

Example:

```
    class ProfileHandler(RequestHandler):
        def initialize(self, database):
            self.database = database

        def get(self, username):
            ...

    app = Application([
        (r'/user/(.*)', ProfileHandler, dict(database=database)),
        ])
```

RequestHandler.**prepare**()
Called at the beginning of a request before *get*/*post*/etc.

Override this method to perform common initialization regardless of the request method.

Asynchronous support: Decorate this method with *gen.coroutine* or *return_future* to make it asynchronous (the *asynchronous* decorator cannot be used on *prepare*). If this method returns a *Future* execution will not proceed until the *Future* is done.

New in version 3.1: Asynchronous support.

RequestHandler.**on_finish**()
Called after the end of a request.

Override this method to perform cleanup, logging, etc. This method is a counterpart to *prepare*. on_finish may not produce any output, as it is called after the response has been sent to the client.

Implement any of the following methods (collectively known as the HTTP verb methods) to handle the corresponding HTTP method. These methods can be made asynchronous with one of the following decorators: *gen.coroutine*, *return_future*, or *asynchronous*.

The arguments to these methods come from the *URLSpec*: Any capturing groups in the regular expression become arguments to the HTTP verb methods (keyword arguments if the group is named, positional arguments if its unnamed).

To support a method not on this list, override the class variable `SUPPORTED_METHODS`:

```
class WebDAVHandler(RequestHandler):
    SUPPORTED_METHODS = RequestHandler.SUPPORTED_METHODS + ('PROPFIND',)

    def propfind(self):
        pass
```

RequestHandler.**get**(**args*, ***kwargs*)

RequestHandler.**head**(**args*, ***kwargs*)

RequestHandler.**post**(**args*, ***kwargs*)

RequestHandler.**delete**(**args*, ***kwargs*)

RequestHandler.**patch**(**args*, ***kwargs*)

RequestHandler.**put**(**args*, ***kwargs*)

RequestHandler.**options**(**args*, ***kwargs*)

Input

RequestHandler.**get_argument**(*name*, *default=<object object>*, *strip=True*)
Returns the value of the argument with the given name.

If default is not provided, the argument is considered to be required, and we raise a *MissingArgumentError* if it is missing.

If the argument appears in the url more than once, we return the last value.

The returned value is always unicode.

RequestHandler.**get_arguments**(*name*, *strip=True*)
Returns a list of the arguments with the given name.

If the argument is not present, returns an empty list.

The returned values are always unicode.

RequestHandler.**get_query_argument**(*name*, *default=<object object>*, *strip=True*)
Returns the value of the argument with the given name from the request query string.

If default is not provided, the argument is considered to be required, and we raise a *MissingArgumentError* if it is missing.

If the argument appears in the url more than once, we return the last value.

The returned value is always unicode.

New in version 3.2.

RequestHandler.**get_query_arguments**(*name*, *strip=True*)
Returns a list of the query arguments with the given name.

If the argument is not present, returns an empty list.

The returned values are always unicode.

New in version 3.2.

`RequestHandler.`**`get_body_argument`**(*name*, *default=<object object>*, *strip=True*)
Returns the value of the argument with the given name from the request body.

If default is not provided, the argument is considered to be required, and we raise a *MissingArgumentError* if it is missing.

If the argument appears in the url more than once, we return the last value.

The returned value is always unicode.

New in version 3.2.

`RequestHandler.`**`get_body_arguments`**(*name*, *strip=True*)
Returns a list of the body arguments with the given name.

If the argument is not present, returns an empty list.

The returned values are always unicode.

New in version 3.2.

`RequestHandler.`**`decode_argument`**(*value*, *name=None*)
Decodes an argument from the request.

The argument has been percent-decoded and is now a byte string. By default, this method decodes the argument as utf-8 and returns a unicode string, but this may be overridden in subclasses.

This method is used as a filter for both *get_argument()* and for values extracted from the url and passed to *get()*/*post()*/etc.

The name of the argument is provided if known, but may be None (e.g. for unnamed groups in the url regex).

`RequestHandler.`**`request`**
The *tornado.httputil.HTTPServerRequest* object containing additional request parameters including e.g. headers and body data.

`RequestHandler.`**`path_args`**

`RequestHandler.`**`path_kwargs`**
The `path_args` and `path_kwargs` attributes contain the positional and keyword arguments that are passed to the *HTTP verb methods*. These attributes are set before those methods are called, so the values are available during *prepare*.

Output

`RequestHandler.`**`set_status`**(*status_code*, *reason=None*)
Sets the status code for our response.

Parameters

- **`status_code`** (*int*) – Response status code. If `reason` is `None`, it must be present in `httplib.responses`.
- **`reason`** (*string*) – Human-readable reason phrase describing the status code. If `None`, it will be filled in from `httplib.responses`.

`RequestHandler.`**`set_header`**(*name*, *value*)
Sets the given response header name and value.

If a datetime is given, we automatically format it according to the HTTP specification. If the value is not a string, we convert it to a string. All header values are then encoded as UTF-8.

RequestHandler.**add_header**(*name*, *value*)
Adds the given response header and value.

Unlike *set_header*, *add_header* may be called multiple times to return multiple values for the same header.

RequestHandler.**clear_header**(*name*)
Clears an outgoing header, undoing a previous *set_header* call.

Note that this method does not apply to multi-valued headers set by *add_header*.

RequestHandler.**set_default_headers**()
Override this to set HTTP headers at the beginning of the request.

For example, this is the place to set a custom `Server` header. Note that setting such headers in the normal flow of request processing may not do what you want, since headers may be reset during error handling.

RequestHandler.**write**(*chunk*)
Writes the given chunk to the output buffer.

To write the output to the network, use the flush() method below.

If the given chunk is a dictionary, we write it as JSON and set the Content-Type of the response to be `application/json`. (if you want to send JSON as a different `Content-Type`, call set_header *after* calling write()).

Note that lists are not converted to JSON because of a potential cross-site security vulnerability. All JSON output should be wrapped in a dictionary. More details at http://haacked.com/archive/2009/06/25/json-hijacking.aspx/ and https://github.com/facebook/tornado/issues/1009

RequestHandler.**flush**(*include_footers=False*, *callback=None*)
Flushes the current output buffer to the network.

The `callback` argument, if given, can be used for flow control: it will be run when all flushed data has been written to the socket. Note that only one flush callback can be outstanding at a time; if another flush occurs before the previous flush's callback has been run, the previous callback will be discarded.

Changed in version 4.0: Now returns a *Future* if no callback is given.

RequestHandler.**finish**(*chunk=None*)
Finishes this response, ending the HTTP request.

RequestHandler.**render**(*template_name*, ***kwargs*)
Renders the template with the given arguments as the response.

RequestHandler.**render_string**(*template_name*, ***kwargs*)
Generate the given template with the given arguments.

We return the generated byte string (in utf8). To generate and write a template as a response, use render() above.

RequestHandler.**get_template_namespace**()
Returns a dictionary to be used as the default template namespace.

May be overridden by subclasses to add or modify values.

The results of this method will be combined with additional defaults in the *tornado.template* module and keyword arguments to *render* or *render_string*.

RequestHandler.**redirect**(*url*, *permanent=False*, *status=None*)
Sends a redirect to the given (optionally relative) URL.

If the `status` argument is specified, that value is used as the HTTP status code; otherwise either 301 (permanent) or 302 (temporary) is chosen based on the `permanent` argument. The default is 302 (temporary).

`RequestHandler.`**`send_error`**(*status_code=500*, ***kwargs*)
Sends the given HTTP error code to the browser.

If *`flush()`* has already been called, it is not possible to send an error, so this method will simply terminate the response. If output has been written but not yet flushed, it will be discarded and replaced with the error page.

Override *`write_error()`* to customize the error page that is returned. Additional keyword arguments are passed through to *`write_error`*.

`RequestHandler.`**`write_error`**(*status_code*, ***kwargs*)
Override to implement custom error pages.

`write_error` may call *`write`*, *`render`*, *`set_header`*, etc to produce output as usual.

If this error was caused by an uncaught exception (including HTTPError), an `exc_info` triple will be available as `kwargs["exc_info"]`. Note that this exception may not be the "current" exception for purposes of methods like `sys.exc_info()` or `traceback.format_exc`.

`RequestHandler.`**`clear`**()
Resets all headers and content for this response.

`RequestHandler.`**`data_received`**(*chunk*)
Implement this method to handle streamed request data.

Requires the *`stream_request_body`* decorator.

Cookies

`RequestHandler.`**`cookies`**
An alias for *`self.request.cookies`*.

`RequestHandler.`**`get_cookie`**(*name*, *default=None*)
Gets the value of the cookie with the given name, else default.

`RequestHandler.`**`set_cookie`**(*name*, *value*, *domain=None*, *expires=None*, *path='/'*, *expires_days=None*, ***kwargs*)
Sets the given cookie name/value with the given options.

Additional keyword arguments are set on the Cookie.Morsel directly. See http://docs.python.org/library/cookie.html#morsel-objects for available attributes.

`RequestHandler.`**`clear_cookie`**(*name*, *path='/'*, *domain=None*)
Deletes the cookie with the given name.

Due to limitations of the cookie protocol, you must pass the same path and domain to clear a cookie as were used when that cookie was set (but there is no way to find out on the server side which values were used for a given cookie).

`RequestHandler.`**`clear_all_cookies`**(*path='/'*, *domain=None*)
Deletes all the cookies the user sent with this request.

See *`clear_cookie`* for more information on the path and domain parameters.

Changed in version 3.2: Added the `path` and `domain` parameters.

`RequestHandler.`**`get_secure_cookie`**(*name*, *value=None*, *max_age_days=31*, *min_version=None*)
Returns the given signed cookie if it validates, or None.

The decoded cookie value is returned as a byte string (unlike *`get_cookie`*).

Changed in version 3.2.1: Added the `min_version` argument. Introduced cookie version 2; both versions 1 and 2 are accepted by default.

`RequestHandler.`**`get_secure_cookie_key_version`**(*name*, *value=None*)
Returns the signing key version of the secure cookie.

The version is returned as int.

`RequestHandler.`**`set_secure_cookie`**(*name*, *value*, *expires_days=30*, *version=None*, ***kwargs*)
Signs and timestamps a cookie so it cannot be forged.

You must specify the `cookie_secret` setting in your Application to use this method. It should be a long, random sequence of bytes to be used as the HMAC secret for the signature.

To read a cookie set with this method, use *get_secure_cookie()*.

Note that the `expires_days` parameter sets the lifetime of the cookie in the browser, but is independent of the `max_age_days` parameter to *get_secure_cookie*.

Secure cookies may contain arbitrary byte values, not just unicode strings (unlike regular cookies)

Changed in version 3.2.1: Added the `version` argument. Introduced cookie version 2 and made it the default.

`RequestHandler.`**`create_signed_value`**(*name*, *value*, *version=None*)
Signs and timestamps a string so it cannot be forged.

Normally used via set_secure_cookie, but provided as a separate method for non-cookie uses. To decode a value not stored as a cookie use the optional value argument to get_secure_cookie.

Changed in version 3.2.1: Added the `version` argument. Introduced cookie version 2 and made it the default.

`tornado.web.`**`MIN_SUPPORTED_SIGNED_VALUE_VERSION = 1`**
The oldest signed value version supported by this version of Tornado.

Signed values older than this version cannot be decoded.

New in version 3.2.1.

`tornado.web.`**`MAX_SUPPORTED_SIGNED_VALUE_VERSION = 2`**
The newest signed value version supported by this version of Tornado.

Signed values newer than this version cannot be decoded.

New in version 3.2.1.

`tornado.web.`**`DEFAULT_SIGNED_VALUE_VERSION = 2`**
The signed value version produced by *RequestHandler.create_signed_value*.

May be overridden by passing a `version` keyword argument.

New in version 3.2.1.

`tornado.web.`**`DEFAULT_SIGNED_VALUE_MIN_VERSION = 1`**
The oldest signed value accepted by *RequestHandler.get_secure_cookie*.

May be overridden by passing a `min_version` keyword argument.

New in version 3.2.1.

Other

`RequestHandler.`**`application`**
The *Application* object serving this request

RequestHandler.**check_etag_header**()
Checks the Etag header against requests's If-None-Match.

Returns True if the request's Etag matches and a 304 should be returned. For example:

```
self.set_etag_header()
if self.check_etag_header():
    self.set_status(304)
    return
```

This method is called automatically when the request is finished, but may be called earlier for applications that override *compute_etag* and want to do an early check for If-None-Match before completing the request. The Etag header should be set (perhaps with *set_etag_header*) before calling this method.

RequestHandler.**check_xsrf_cookie**()
Verifies that the _xsrf cookie matches the _xsrf argument.

To prevent cross-site request forgery, we set an _xsrf cookie and include the same value as a non-cookie field with all POST requests. If the two do not match, we reject the form submission as a potential forgery.

The _xsrf value may be set as either a form field named _xsrf or in a custom HTTP header named X-XSRFToken or X-CSRFToken (the latter is accepted for compatibility with Django).

See http://en.wikipedia.org/wiki/Cross-site_request_forgery

Prior to release 1.1.1, this check was ignored if the HTTP header X-Requested-With: XMLHTTPRequest was present. This exception has been shown to be insecure and has been removed. For more information please see http://www.djangoproject.com/weblog/2011/feb/08/security/ http://weblog.rubyonrails.org/2011/2/8/csrf-protection-bypass-in-ruby-on-rails

Changed in version 3.2.2: Added support for cookie version 2. Both versions 1 and 2 are supported.

RequestHandler.**compute_etag**()
Computes the etag header to be used for this request.

By default uses a hash of the content written so far.

May be overridden to provide custom etag implementations, or may return None to disable tornado's default etag support.

RequestHandler.**create_template_loader**(*template_path*)
Returns a new template loader for the given path.

May be overridden by subclasses. By default returns a directory-based loader on the given path, using the autoescape and template_whitespace application settings. If a template_loader application setting is supplied, uses that instead.

RequestHandler.**current_user**
The authenticated user for this request.

This is set in one of two ways:

- A subclass may override *get_current_user()*, which will be called automatically the first time self.current_user is accessed. *get_current_user()* will only be called once per request, and is cached for future access:

```
def get_current_user(self):
    user_cookie = self.get_secure_cookie("user")
    if user_cookie:
        return json.loads(user_cookie)
    return None
```

- It may be set as a normal variable, typically from an overridden *prepare()*:

```
        @gen.coroutine
        def prepare(self):
            user_id_cookie = self.get_secure_cookie("user_id")
            if user_id_cookie:
                self.current_user = yield load_user(user_id_cookie)
```

Note that *prepare()* may be a coroutine while *get_current_user()* may not, so the latter form is necessary if loading the user requires asynchronous operations.

The user object may any type of the application's choosing.

RequestHandler.**get_browser_locale**(*default='en_US'*)
Determines the user's locale from `Accept-Language` header.

See http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.4

RequestHandler.**get_current_user**()
Override to determine the current user from, e.g., a cookie.

This method may not be a coroutine.

RequestHandler.**get_login_url**()
Override to customize the login URL based on the request.

By default, we use the `login_url` application setting.

RequestHandler.**get_status**()
Returns the status code for our response.

RequestHandler.**get_template_path**()
Override to customize template path for each handler.

By default, we use the `template_path` application setting. Return None to load templates relative to the calling file.

RequestHandler.**get_user_locale**()
Override to determine the locale from the authenticated user.

If None is returned, we fall back to *get_browser_locale()*.

This method should return a *tornado.locale.Locale* object, most likely obtained via a call like `tornado.locale.get("en")`

RequestHandler.**locale**
The locale for the current session.

Determined by either *get_user_locale*, which you can override to set the locale based on, e.g., a user preference stored in a database, or *get_browser_locale*, which uses the `Accept-Language` header.

RequestHandler.**log_exception**(*typ*, *value*, *tb*)
Override to customize logging of uncaught exceptions.

By default logs instances of *HTTPError* as warnings without stack traces (on the `tornado.general` logger), and all other exceptions as errors with stack traces (on the `tornado.application` logger).

New in version 3.1.

RequestHandler.**on_connection_close**()
Called in async handlers if the client closed the connection.

Override this to clean up resources associated with long-lived connections. Note that this method is called only if the connection was closed during asynchronous processing; if you need to do cleanup after every request override *on_finish* instead.

Proxies may keep a connection open for a time (perhaps indefinitely) after the client has gone away, so this method may not be called promptly after the end user closes their connection.

RequestHandler.**require_setting**(*name*, *feature='this feature'*)
Raises an exception if the given app setting is not defined.

RequestHandler.**reverse_url**(*name*, **args*)
Alias for *Application.reverse_url*.

RequestHandler.**set_etag_header**()
Sets the response's Etag header using `self.compute_etag()`.

Note: no header will be set if `compute_etag()` returns `None`.

This method is called automatically when the request is finished.

RequestHandler.**settings**
An alias for *self.application.settings*.

RequestHandler.**static_url**(*path*, *include_host=None*, ***kwargs*)
Returns a static URL for the given relative static file path.

This method requires you set the `static_path` setting in your application (which specifies the root directory of your static files).

This method returns a versioned url (by default appending `?v=<signature>`), which allows the static files to be cached indefinitely. This can be disabled by passing `include_version=False` (in the default implementation; other static file implementations are not required to support this, but they may support other options).

By default this method returns URLs relative to the current host, but if `include_host` is true the URL returned will be absolute. If this handler has an `include_host` attribute, that value will be used as the default for all *static_url* calls that do not pass `include_host` as a keyword argument.

RequestHandler.**xsrf_form_html**()
An HTML `<input/>` element to be included with all POST forms.

It defines the `_xsrf` input value, which we check on all POST requests to prevent cross-site request forgery. If you have set the `xsrf_cookies` application setting, you must include this HTML within all of your HTML forms.

In a template, this method should be called with `{% module xsrf_form_html() %}`

See *check_xsrf_cookie()* above for more information.

RequestHandler.**xsrf_token**
The XSRF-prevention token for the current user/session.

To prevent cross-site request forgery, we set an '_xsrf' cookie and include the same '_xsrf' value as an argument with all POST requests. If the two do not match, we reject the form submission as a potential forgery.

See http://en.wikipedia.org/wiki/Cross-site_request_forgery

Changed in version 3.2.2: The xsrf token will now be have a random mask applied in every request, which makes it safe to include the token in pages that are compressed. See http://breachattack.com for more information on the issue fixed by this change. Old (version 1) cookies will be converted to version 2 when this method is called unless the `xsrf_cookie_version` *Application* setting is set to 1.

Changed in version 4.3: The `xsrf_cookie_kwargs` *Application* setting may be used to supply additional cookie options (which will be passed directly to *set_cookie*). For example, `xsrf_cookie_kwargs=dict(httponly=True, secure=True)` will set the `secure` and `httponly` flags on the `_xsrf` cookie.

Application configuration

class tornado.web.**Application**(*handlers=None*, *default_host=''*, *transforms=None*, ***settings*)
A collection of request handlers that make up a web application.

Instances of this class are callable and can be passed directly to HTTPServer to serve the application:

```
application = web.Application([
    (r"/", MainPageHandler),
])
http_server = httpserver.HTTPServer(application)
http_server.listen(8080)
ioloop.IOLoop.current().start()
```

The constructor for this class takes in a list of *URLSpec* objects or (regexp, request_class) tuples. When we receive requests, we iterate over the list in order and instantiate an instance of the first request class whose regexp matches the request path. The request class can be specified as either a class object or a (fully-qualified) name.

Each tuple can contain additional elements, which correspond to the arguments to the *URLSpec* constructor. (Prior to Tornado 3.2, only tuples of two or three elements were allowed).

A dictionary may be passed as the third element of the tuple, which will be used as keyword arguments to the handler's constructor and *initialize* method. This pattern is used for the *StaticFileHandler* in this example (note that a *StaticFileHandler* can be installed automatically with the static_path setting described below):

```
application = web.Application([
    (r"/static/(.*)", web.StaticFileHandler, {"path": "/var/www"}),
])
```

We support virtual hosts with the *add_handlers* method, which takes in a host regular expression as the first argument:

```
application.add_handlers(r"www\.myhost\.com", [
    (r"/article/([0-9]+)", ArticleHandler),
])
```

You can serve static files by sending the `static_path` setting as a keyword argument. We will serve those files from the `/static/` URI (this is configurable with the `static_url_prefix` setting), and we will serve `/favicon.ico` and `/robots.txt` from the same directory. A custom subclass of *StaticFileHandler* can be specified with the `static_handler_class` setting.

settings
Additional keyword arguments passed to the constructor are saved in the *settings* dictionary, and are often referred to in documentation as "application settings". Settings are used to customize various aspects of Tornado (although in some cases richer customization is possible by overriding methods in a subclass of *RequestHandler*). Some applications also like to use the *settings* dictionary as a way to make application-specific settings available to handlers without using global variables. Settings used in Tornado are described below.

General settings:

- `autoreload`: If `True`, the server process will restart when any source files change, as described in *Debug mode and automatic reloading*. This option is new in Tornado 3.2; previously this functionality was controlled by the `debug` setting.
- `debug`: Shorthand for several debug mode settings, described in *Debug mode and automatic reloading*. Setting `debug=True` is equivalent to `autoreload=True`, `compiled_template_cache=False`, `static_hash_cache=False`, `serve_traceback=True`.

- `default_handler_class` and `default_handler_args`: This handler will be used if no other match is found; use this to implement custom 404 pages (new in Tornado 3.2).
- `compress_response`: If `True`, responses in textual formats will be compressed automatically. New in Tornado 4.0.
- `gzip`: Deprecated alias for `compress_response` since Tornado 4.0.
- `log_function`: This function will be called at the end of every request to log the result (with one argument, the *`RequestHandler`* object). The default implementation writes to the `logging` module's root logger. May also be customized by overriding *`Application.log_request`*.
- `serve_traceback`: If true, the default error page will include the traceback of the error. This option is new in Tornado 3.2; previously this functionality was controlled by the `debug` setting.
- `ui_modules` and `ui_methods`: May be set to a mapping of *`UIModule`* or UI methods to be made available to templates. May be set to a module, dictionary, or a list of modules and/or dicts. See *UI modules* for more details.

Authentication and security settings:

- `cookie_secret`: Used by *`RequestHandler.get_secure_cookie`* and *`set_secure_cookie`* to sign cookies.
- `key_version`: Used by requestHandler *`set_secure_cookie`* to sign cookies with a specific key when `cookie_secret` is a key dictionary.
- `login_url`: The *`authenticated`* decorator will redirect to this url if the user is not logged in. Can be further customized by overriding *`RequestHandler.get_login_url`*
- `xsrf_cookies`: If true, *Cross-site request forgery protection* will be enabled.
- `xsrf_cookie_version`: Controls the version of new XSRF cookies produced by this server. Should generally be left at the default (which will always be the highest supported version), but may be set to a lower value temporarily during version transitions. New in Tornado 3.2.2, which introduced XSRF cookie version 2.
- `xsrf_cookie_kwargs`: May be set to a dictionary of additional arguments to be passed to *`RequestHandler.set_cookie`* for the XSRF cookie.
- `twitter_consumer_key`, `twitter_consumer_secret`, `friendfeed_consumer_key`, `friendfeed_consumer_secret`, `google_consumer_key`, `google_consumer_secret`, `facebook_api_key`, `facebook_secret`: Used in the *`tornado.auth`* module to authenticate to various APIs.

Template settings:

- `autoescape`: Controls automatic escaping for templates. May be set to `None` to disable escaping, or to the *name* of a function that all output should be passed through. Defaults to `"xhtml_escape"`. Can be changed on a per-template basis with the `{% autoescape %}` directive.
- `compiled_template_cache`: Default is `True`; if `False` templates will be recompiled on every request. This option is new in Tornado 3.2; previously this functionality was controlled by the `debug` setting.
- `template_path`: Directory containing template files. Can be further customized by overriding *`RequestHandler.get_template_path`*
- `template_loader`: Assign to an instance of *`tornado.template.BaseLoader`* to customize template loading. If this setting is used the `template_path` and `autoescape` settings are ignored. Can be further customized by overriding *`RequestHandler.create_template_loader`*.

- `template_whitespace`: Controls handling of whitespace in templates; see *tornado.template.filter_whitespace* for allowed values. New in Tornado 4.3.

Static file settings:

- `static_hash_cache`: Default is `True`; if `False` static urls will be recomputed on every request. This option is new in Tornado 3.2; previously this functionality was controlled by the `debug` setting.
- `static_path`: Directory from which static files will be served.
- `static_url_prefix`: Url prefix for static files, defaults to `"/static/"`.
- `static_handler_class`, `static_handler_args`: May be set to use a different handler for static files instead of the default *tornado.web.StaticFileHandler*. `static_handler_args`, if set, should be a dictionary of keyword arguments to be passed to the handler's `initialize` method.

listen(*port*, *address=''*, ***kwargs*)

Starts an HTTP server for this application on the given port.

This is a convenience alias for creating an *HTTPServer* object and calling its listen method. Keyword arguments not supported by *HTTPServer.listen* are passed to the *HTTPServer* constructor. For advanced uses (e.g. multi-process mode), do not use this method; create an *HTTPServer* and call its *TCPServer.bind*/*TCPServer.start* methods directly.

Note that after calling this method you still need to call `IOLoop.current().start()` to start the server.

Returns the *HTTPServer* object.

Changed in version 4.3: Now returns the *HTTPServer* object.

add_handlers(*host_pattern*, *host_handlers*)

Appends the given handlers to our handler list.

Host patterns are processed sequentially in the order they were added. All matching patterns will be considered.

reverse_url(*name*, **args*)

Returns a URL path for handler named `name`

The handler must be added to the application as a named *URLSpec*.

Args will be substituted for capturing groups in the *URLSpec* regex. They will be converted to strings if necessary, encoded as utf8, and url-escaped.

log_request(*handler*)

Writes a completed HTTP request to the logs.

By default writes to the python root logger. To change this behavior either subclass Application and override this method, or pass a function in the application settings dictionary as `log_function`.

class `tornado.web.`**URLSpec**(*pattern*, *handler*, *kwargs=None*, *name=None*)

Specifies mappings between URLs and handlers.

Parameters:

- `pattern`: Regular expression to be matched. Any capturing groups in the regex will be passed in to the handler's get/post/etc methods as arguments (by keyword if named, by position if unnamed. Named and unnamed capturing groups may may not be mixed in the same rule).
- `handler`: *RequestHandler* subclass to be invoked.
- `kwargs` (optional): A dictionary of additional arguments to be passed to the handler's constructor.

- `name` (optional): A name for this handler. Used by *`Application.reverse_url`*.

The `URLSpec` class is also available under the name `tornado.web.url`.

Decorators

`tornado.web.`**`asynchronous`**(*method*)

Wrap request handler methods with this if they are asynchronous.

This decorator is for callback-style asynchronous methods; for coroutines, use the `@gen.coroutine` decorator without `@asynchronous`. (It is legal for legacy reasons to use the two decorators together provided `@asynchronous` is first, but `@asynchronous` will be ignored in this case)

This decorator should only be applied to the *HTTP verb methods*; its behavior is undefined for any other method. This decorator does not *make* a method asynchronous; it tells the framework that the method *is* asynchronous. For this decorator to be useful the method must (at least sometimes) do something asynchronous.

If this decorator is given, the response is not finished when the method returns. It is up to the request handler to call *`self.finish()`* to finish the HTTP request. Without this decorator, the request is automatically finished when the `get()` or `post()` method returns. Example:

```
class MyRequestHandler(RequestHandler):
    @asynchronous
    def get(self):
       http = httpclient.AsyncHTTPClient()
       http.fetch("http://friendfeed.com/", self._on_download)

    def _on_download(self, response):
       self.write("Downloaded!")
       self.finish()
```

Changed in version 3.1: The ability to use `@gen.coroutine` without `@asynchronous`.

Changed in version 4.3: Returning anything but `None` or a yieldable object from a method decorated with `@asynchronous` is an error. Such return values were previously ignored silently.

`tornado.web.`**`authenticated`**(*method*)

Decorate methods with this to require that the user be logged in.

If the user is not logged in, they will be redirected to the configured *`login url`*.

If you configure a login url with a query parameter, Tornado will assume you know what you're doing and use it as-is. If not, it will add a `next` parameter so the login page knows where to send you once you're logged in.

`tornado.web.`**`addslash`**(*method*)

Use this decorator to add a missing trailing slash to the request path.

For example, a request to `/foo` would redirect to `/foo/` with this decorator. Your request handler mapping should use a regular expression like `r'/foo/?'` in conjunction with using the decorator.

`tornado.web.`**`removeslash`**(*method*)

Use this decorator to remove trailing slashes from the request path.

For example, a request to `/foo/` would redirect to `/foo` with this decorator. Your request handler mapping should use a regular expression like `r'/foo/*'` in conjunction with using the decorator.

`tornado.web.`**`stream_request_body`**(*cls*)

Apply to *`RequestHandler`* subclasses to enable streaming body support.

This decorator implies the following changes:

- *HTTPServerRequest.body* is undefined, and body arguments will not be included in *RequestHandler.get_argument*.
- *RequestHandler.prepare* is called when the request headers have been read instead of after the entire body has been read.
- The subclass must define a method `data_received(self, data):`, which will be called zero or more times as data is available. Note that if the request has an empty body, `data_received` may not be called.
- `prepare` and `data_received` may return Futures (such as via `@gen.coroutine`, in which case the next method will not be called until those futures have completed.
- The regular HTTP method (`post`, `put`, etc) will be called after the entire body has been read.

There is a subtle interaction between `data_received` and asynchronous `prepare`: The first call to `data_received` may occur at any point after the call to `prepare` has returned *or yielded*.

Everything else

exception `tornado.web.`**`HTTPError`**(*status_code=500*, *log_message=None*, **args*, ***kwargs*)

An exception that will turn into an HTTP error response.

Raising an *HTTPError* is a convenient alternative to calling *RequestHandler.send_error* since it automatically ends the current function.

To customize the response sent with an *HTTPError*, override *RequestHandler.write_error*.

Parameters

- **`status_code`** (*int*) – HTTP status code. Must be listed in `httplib.responses` unless the `reason` keyword argument is given.
- **`log_message`** (*string*) – Message to be written to the log for this error (will not be shown to the user unless the *Application* is in debug mode). May contain `%s`-style placeholders, which will be filled in with remaining positional parameters.
- **`reason`** (*string*) – Keyword-only argument. The HTTP "reason" phrase to pass in the status line along with `status_code`. Normally determined automatically from `status_code`, but can be used to use a non-standard numeric code.

exception `tornado.web.`**`Finish`**

An exception that ends the request without producing an error response.

When *Finish* is raised in a *RequestHandler*, the request will end (calling *RequestHandler.finish* if it hasn't already been called), but the error-handling methods (including *RequestHandler.write_error*) will not be called.

If *Finish()* was created with no arguments, the pending response will be sent as-is. If *Finish()* was given an argument, that argument will be passed to *RequestHandler.finish()*.

This can be a more convenient way to implement custom error pages than overriding `write_error` (especially in library code):

```
if self.current_user is None:
    self.set_status(401)
    self.set_header('WWW-Authenticate', 'Basic realm="something"')
    raise Finish()
```

Changed in version 4.3: Arguments passed to `Finish()` will be passed on to *RequestHandler.finish*.

exception tornado.web.**MissingArgumentError**(*arg_name*)
Exception raised by *RequestHandler.get_argument*.

This is a subclass of *HTTPError*, so if it is uncaught a 400 response code will be used instead of 500 (and a stack trace will not be logged).

New in version 3.1.

class tornado.web.**UIModule**(*handler*)
A re-usable, modular UI unit on a page.

UI modules often execute additional queries, and they can include additional CSS and JavaScript that will be included in the output page, which is automatically inserted on page render.

Subclasses of UIModule must override the *render* method.

render(**args*, ***kwargs*)
Override in subclasses to return this module's output.

embedded_javascript()
Override to return a JavaScript string to be embedded in the page.

javascript_files()
Override to return a list of JavaScript files needed by this module.

If the return values are relative paths, they will be passed to *RequestHandler.static_url*; otherwise they will be used as-is.

embedded_css()
Override to return a CSS string that will be embedded in the page.

css_files()
Override to returns a list of CSS files required by this module.

If the return values are relative paths, they will be passed to *RequestHandler.static_url*; otherwise they will be used as-is.

html_head()
Override to return an HTML string that will be put in the <head/> element.

html_body()
Override to return an HTML string that will be put at the end of the <body/> element.

render_string(*path*, ***kwargs*)
Renders a template and returns it as a string.

class tornado.web.**ErrorHandler**(*application*, *request*, ***kwargs*)
Generates an error response with `status_code` for all requests.

class tornado.web.**FallbackHandler**(*application*, *request*, ***kwargs*)
A *RequestHandler* that wraps another HTTP server callback.

The fallback is a callable object that accepts an *HTTPServerRequest*, such as an *Application* or *tornado.wsgi.WSGIContainer*. This is most useful to use both Tornado `RequestHandlers` and WSGI in the same server. Typical usage:

```
wsgi_app = tornado.wsgi.WSGIContainer(
    django.core.handlers.wsgi.WSGIHandler())
application = tornado.web.Application([
    (r"/foo", FooHandler),
    (r".*", FallbackHandler, dict(fallback=wsgi_app),
])
```

class tornado.web.**RedirectHandler**(*application*, *request*, ***kwargs*)
Redirects the client to the given URL for all GET requests.

You should provide the keyword argument `url` to the handler, e.g.:

```
application = web.Application([
    (r"/oldpath", web.RedirectHandler, {"url": "/newpath"}),
])
```

class tornado.web.**StaticFileHandler**(*application*, *request*, ***kwargs*)
A simple handler that can serve static content from a directory.

A *StaticFileHandler* is configured automatically if you pass the `static_path` keyword argument to *Application*. This handler can be customized with the `static_url_prefix`, `static_handler_class`, and `static_handler_args` settings.

To map an additional path to this handler for a static data directory you would add a line to your application like:

```
application = web.Application([
    (r"/content/(.*)", web.StaticFileHandler, {"path": "/var/www"}),
])
```

The handler constructor requires a `path` argument, which specifies the local root directory of the content to be served.

Note that a capture group in the regex is required to parse the value for the `path` argument to the get() method (different than the constructor argument above); see *URLSpec* for details.

To serve a file like `index.html` automatically when a directory is requested, set `static_handler_args=dict(default_filename="index.html")` in your application settings, or add `default_filename` as an initializer argument for your `StaticFileHandler`.

To maximize the effectiveness of browser caching, this class supports versioned urls (by default using the argument `?v=`). If a version is given, we instruct the browser to cache this file indefinitely. *make_static_url* (also available as *RequestHandler.static_url*) can be used to construct a versioned url.

This handler is intended primarily for use in development and light-duty file serving; for heavy traffic it will be more efficient to use a dedicated static file server (such as nginx or Apache). We support the HTTP `Accept-Ranges` mechanism to return partial content (because some browsers require this functionality to be present to seek in HTML5 audio or video).

Subclassing notes

This class is designed to be extensible by subclassing, but because of the way static urls are generated with class methods rather than instance methods, the inheritance patterns are somewhat unusual. Be sure to use the `@classmethod` decorator when overriding a class method. Instance methods may use the attributes `self.path` `self.absolute_path`, and `self.modified`.

Subclasses should only override methods discussed in this section; overriding other methods is error-prone. Overriding `StaticFileHandler.get` is particularly problematic due to the tight coupling with `compute_etag` and other methods.

To change the way static urls are generated (e.g. to match the behavior of another server or CDN), override *make_static_url*, *parse_url_path*, *get_cache_time*, and/or *get_version*.

To replace all interaction with the filesystem (e.g. to serve static content from a database), override *get_content*, *get_content_size*, *get_modified_time*, *get_absolute_path*, and *validate_absolute_path*.

Changed in version 3.1: Many of the methods for subclasses were added in Tornado 3.1.

compute_etag()
Sets the `Etag` header based on static url version.

This allows efficient `If-None-Match` checks against cached versions, and sends the correct `Etag` for a partial response (i.e. the same `Etag` as the full file).

New in version 3.1.

set_headers()
Sets the content and caching headers on the response.

New in version 3.1.

should_return_304()
Returns True if the headers indicate that we should return 304.

New in version 3.1.

classmethod get_absolute_path(*root*, *path*)
Returns the absolute location of `path` relative to `root`.

`root` is the path configured for this *StaticFileHandler* (in most cases the `static_path` *Application* setting).

This class method may be overridden in subclasses. By default it returns a filesystem path, but other strings may be used as long as they are unique and understood by the subclass's overridden *get_content*.

New in version 3.1.

validate_absolute_path(*root*, *absolute_path*)
Validate and return the absolute path.

`root` is the configured path for the *StaticFileHandler*, and `path` is the result of *get_absolute_path*

This is an instance method called during request processing, so it may raise *HTTPError* or use methods like *RequestHandler.redirect* (return None after redirecting to halt further processing). This is where 404 errors for missing files are generated.

This method may modify the path before returning it, but note that any such modifications will not be understood by *make_static_url*.

In instance methods, this method's result is available as `self.absolute_path`.

New in version 3.1.

classmethod get_content(*abspath*, *start=None*, *end=None*)
Retrieve the content of the requested resource which is located at the given absolute path.

This class method may be overridden by subclasses. Note that its signature is different from other overridable class methods (no `settings` argument); this is deliberate to ensure that `abspath` is able to stand on its own as a cache key.

This method should either return a byte string or an iterator of byte strings. The latter is preferred for large files as it helps reduce memory fragmentation.

New in version 3.1.

classmethod get_content_version(*abspath*)
Returns a version string for the resource at the given path.

This class method may be overridden by subclasses. The default implementation is a hash of the file's contents.

New in version 3.1.

get_content_size()
Retrieve the total size of the resource at the given path.

This method may be overridden by subclasses.

New in version 3.1.

Changed in version 4.0: This method is now always called, instead of only when partial results are requested.

get_modified_time()
Returns the time that `self.absolute_path` was last modified.

May be overridden in subclasses. Should return a `datetime` object or None.

New in version 3.1.

get_content_type()
Returns the `Content-Type` header to be used for this request.

New in version 3.1.

set_extra_headers(*path*)
For subclass to add extra headers to the response

get_cache_time(*path*, *modified*, *mime_type*)
Override to customize cache control behavior.

Return a positive number of seconds to make the result cacheable for that amount of time or 0 to mark resource as cacheable for an unspecified amount of time (subject to browser heuristics).

By default returns cache expiry of 10 years for resources requested with `v` argument.

classmethod make_static_url(*settings*, *path*, *include_version=True*)
Constructs a versioned url for the given path.

This method may be overridden in subclasses (but note that it is a class method rather than an instance method). Subclasses are only required to implement the signature `make_static_url(cls, settings, path)`; other keyword arguments may be passed through *static_url* but are not standard.

`settings` is the *Application.settings* dictionary. `path` is the static path being requested. The url returned should be relative to the current host.

`include_version` determines whether the generated URL should include the query string containing the version hash of the file corresponding to the given `path`.

parse_url_path(*url_path*)
Converts a static URL path into a filesystem path.

`url_path` is the path component of the URL with `static_url_prefix` removed. The return value should be filesystem path relative to `static_path`.

This is the inverse of *make_static_url*.

classmethod get_version(*settings*, *path*)
Generate the version string to be used in static URLs.

`settings` is the *Application.settings* dictionary and `path` is the relative location of the requested asset on the filesystem. The returned value should be a string, or `None` if no version could be determined.

Changed in version 3.1: This method was previously recommended for subclasses to override; *get_content_version* is now preferred as it allows the base class to handle caching of the result.

4.2.2 `tornado.template` — Flexible output generation

A simple template system that compiles templates to Python code.

Basic usage looks like:

```
t = template.Template("<html>{{ myvalue }}</html>")
print t.generate(myvalue="XXX")
```

Loader is a class that loads templates from a root directory and caches the compiled templates:

```
loader = template.Loader("/home/btaylor")
print loader.load("test.html").generate(myvalue="XXX")
```

We compile all templates to raw Python. Error-reporting is currently... uh, interesting. Syntax for the templates:

```
### base.html
<html>
  <head>
    <title>{% block title %}Default title{% end %}</title>
  </head>
  <body>
    <ul>
      {% for student in students %}
        {% block student %}
          <li>{{ escape(student.name) }}</li>
        {% end %}
      {% end %}
    </ul>
  </body>
</html>

### bold.html
{% extends "base.html" %}

{% block title %}A bolder title{% end %}

{% block student %}
  <li><span style="bold">{{ escape(student.name) }}</span></li>
{% end %}
```

Unlike most other template systems, we do not put any restrictions on the expressions you can include in your statements. `if` and `for` blocks get translated exactly into Python, so you can do complex expressions like:

```
{% for student in [p for p in people if p.student and p.age > 23] %}
  <li>{{ escape(student.name) }}</li>
{% end %}
```

Translating directly to Python means you can apply functions to expressions easily, like the `escape()` function in the examples above. You can pass functions in to your template just like any other variable (In a *RequestHandler*, override *RequestHandler.get_template_namespace*):

```
### Python code
def add(x, y):
   return x + y
template.execute(add=add)

### The template
{{ add(1, 2) }}
```

We provide the functions *escape()*, *url_escape()*, *json_encode()*, and *squeeze()* to all templates by default.

Typical applications do not create *Template* or *Loader* instances by hand, but instead use the *render* and *render_string* methods of *tornado.web.RequestHandler*, which load templates automatically based on the `template_path` *Application* setting.

Variable names beginning with `_tt_` are reserved by the template system and should not be used by application code.

Syntax Reference

Template expressions are surrounded by double curly braces: `{{ ... }}`. The contents may be any python expression, which will be escaped according to the current autoescape setting and inserted into the output. Other template directives use `{% %}`.

To comment out a section so that it is omitted from the output, surround it with `{# ... #}`.

These tags may be escaped as `{{!`, `{%!`, and `{#!` if you need to include a literal `{{`, `{%`, or `{#` in the output.

`{% apply *function* %}...{% end %}` Applies a function to the output of all template code between `apply` and `end`:

```
{% apply linkify %}{{name}} said: {{message}}{% end %}
```

Note that as an implementation detail apply blocks are implemented as nested functions and thus may interact strangely with variables set via `{% set %}`, or the use of `{% break %}` or `{% continue %}` within loops.

`{% autoescape *function* %}` Sets the autoescape mode for the current file. This does not affect other files, even those referenced by `{% include %}`. Note that autoescaping can also be configured globally, at the *Application* or *Loader*.:

```
{% autoescape xhtml_escape %}
{% autoescape None %}
```

`{% block *name* %}...{% end %}` Indicates a named, replaceable block for use with `{% extends %}`. Blocks in the parent template will be replaced with the contents of the same-named block in a child template.:

```
<!-- base.html -->
<title>{% block title %}Default title{% end %}</title>

<!-- mypage.html -->
{% extends "base.html" %}
{% block title %}My page title{% end %}
```

`{% comment ... %}` A comment which will be removed from the template output. Note that there is no `{% end %}` tag; the comment goes from the word `comment` to the closing `%}` tag.

`{% extends *filename* %}` Inherit from another template. Templates that use `extends` should contain one or more `block` tags to replace content from the parent template. Anything in the child template not contained in a `block` tag will be ignored. For an example, see the `{% block %}` tag.

`{% for *var* in *expr* %}...{% end %}` Same as the python `for` statement. `{% break %}` and `{% continue %}` may be used inside the loop.

`{% from *x* import *y* %}` Same as the python `import` statement.

`{% if *condition* %}...{% elif *condition* %}...{% else %}...{% end %}` Conditional statement - outputs the first section whose condition is true. (The `elif` and `else` sections are optional)

{% import *module* %} Same as the python `import` statement.

{% include *filename* %} Includes another template file. The included file can see all the local variables as if it were copied directly to the point of the `include` directive (the `{% autoescape %}` directive is an exception). Alternately, `{% module Template(filename, **kwargs) %}` may be used to include another template with an isolated namespace.

{% module *expr* %} Renders a *UIModule*. The output of the `UIModule` is not escaped:

```
{% module Template("foo.html", arg=42) %}
```

`UIModules` are a feature of the *tornado.web.RequestHandler* class (and specifically its `render` method) and will not work when the template system is used on its own in other contexts.

{% raw *expr* %} Outputs the result of the given expression without autoescaping.

{% set *x* = *y* %} Sets a local variable.

{% try %}...{% except %}...{% else %}...{% finally %}...{% end %} Same as the python `try` statement.

{% while *condition* %}... {% end %} Same as the python `while` statement. `{% break %}` and `{% continue %}` may be used inside the loop.

{% whitespace *mode* %} Sets the whitespace mode for the remainder of the current file (or until the next `{% whitespace %}` directive). See *filter_whitespace* for available options. New in Tornado 4.3.

Class reference

class tornado.template.**Template**(*template_string*, *name="<string>"*, *loader=None*, *compress_whitespace=None*, *autoescape="xhtml_escape"*, *whitespace=None*)

A compiled template.

We compile into Python from the given template_string. You can generate the template from variables with generate().

Construct a Template.

Parameters

- **template_string** (*str*) – the contents of the template file.
- **name** (*str*) – the filename from which the template was loaded (used for error message).
- **loader** (tornado.template.BaseLoader) – the *BaseLoader* responsible for this template, used to resolve `{% include %}` and `{% extend %}` directives.
- **compress_whitespace** (*bool*) – Deprecated since Tornado 4.3. Equivalent to `whitespace="single"` if true and `whitespace="all"` if false.
- **autoescape** (*str*) – The name of a function in the template namespace, or `None` to disable escaping by default.
- **whitespace** (*str*) – A string specifying treatment of whitespace; see *filter_whitespace* for options.

Changed in version 4.3: Added `whitespace` parameter; deprecated `compress_whitespace`.

generate(***kwargs*)

Generate this template with the given arguments.

class tornado.template.**BaseLoader**(*autoescape='xhtml_escape'*, *namespace=None*, *whitespace=None*)

Base class for template loaders.

You must use a template loader to use template constructs like {% extends %} and {% include %}. The loader caches all templates after they are loaded the first time.

Construct a template loader.

Parameters

- **autoescape** (*str*) – The name of a function in the template namespace, such as "xhtml_escape", or None to disable autoescaping by default.
- **namespace** (*dict*) – A dictionary to be added to the default template namespace, or None.
- **whitespace** (*str*) – A string specifying default behavior for whitespace in templates; see *filter_whitespace* for options. Default is "single" for files ending in ".html" and ".js" and "all" for other files.

Changed in version 4.3: Added whitespace parameter.

reset()
Resets the cache of compiled templates.

resolve_path(*name*, *parent_path=None*)
Converts a possibly-relative path to absolute (used internally).

load(*name*, *parent_path=None*)
Loads a template.

class tornado.template.**Loader**(*root_directory*, ***kwargs*)
A template loader that loads from a single root directory.

class tornado.template.**DictLoader**(*dict*, ***kwargs*)
A template loader that loads from a dictionary.

exception tornado.template.**ParseError**(*message*, *filename=None*, *lineno=0*)
Raised for template syntax errors.

ParseError instances have filename and lineno attributes indicating the position of the error.

Changed in version 4.3: Added filename and lineno attributes.

tornado.template.**filter_whitespace**(*mode*, *text*)
Transform whitespace in text according to mode.

Available modes are:

- all: Return all whitespace unmodified.
- single: Collapse consecutive whitespace with a single whitespace character, preserving newlines.
- oneline: Collapse all runs of whitespace into a single space character, removing all newlines in the process.

New in version 4.3.

4.2.3 tornado.escape — Escaping and string manipulation

Escaping/unescaping methods for HTML, JSON, URLs, and others.

Also includes a few other miscellaneous string manipulation functions that have crept in over time.

Escaping functions

tornado.escape.**xhtml_escape**(*value*)
Escapes a string so it is valid within HTML or XML.

Escapes the characters <, >, ", ', and &. When used in attribute values the escaped strings must be enclosed in quotes.

Changed in version 3.2: Added the single quote to the list of escaped characters.

tornado.escape.**xhtml_unescape**(*value*)
Un-escapes an XML-escaped string.

tornado.escape.**url_escape**(*value*, *plus=True*)
Returns a URL-encoded version of the given value.

If plus is true (the default), spaces will be represented as "+" instead of "%20". This is appropriate for query strings but not for the path component of a URL. Note that this default is the reverse of Python's urllib module.

New in version 3.1: The plus argument

tornado.escape.**url_unescape**(*value*, *encoding='utf-8'*, *plus=True*)
Decodes the given value from a URL.

The argument may be either a byte or unicode string.

If encoding is None, the result will be a byte string. Otherwise, the result is a unicode string in the specified encoding.

If plus is true (the default), plus signs will be interpreted as spaces (literal plus signs must be represented as "%2B"). This is appropriate for query strings and form-encoded values but not for the path component of a URL. Note that this default is the reverse of Python's urllib module.

New in version 3.1: The plus argument

tornado.escape.**json_encode**(*value*)
JSON-encodes the given Python object.

tornado.escape.**json_decode**(*value*)
Returns Python objects for the given JSON string.

Byte/unicode conversions

These functions are used extensively within Tornado itself, but should not be directly needed by most applications. Note that much of the complexity of these functions comes from the fact that Tornado supports both Python 2 and Python 3.

tornado.escape.**utf8**(*value*)
Converts a string argument to a byte string.

If the argument is already a byte string or None, it is returned unchanged. Otherwise it must be a unicode string and is encoded as utf8.

tornado.escape.**to_unicode**(*value*)
Converts a string argument to a unicode string.

If the argument is already a unicode string or None, it is returned unchanged. Otherwise it must be a byte string and is decoded as utf8.

tornado.escape.**native_str**()
Converts a byte or unicode string into type str. Equivalent to *utf8* on Python 2 and *to_unicode* on Python 3.

tornado.escape.**to_basestring**(*value*)
Converts a string argument to a subclass of basestring.

In python2, byte and unicode strings are mostly interchangeable, so functions that deal with a user-supplied argument in combination with ascii string constants can use either and should return the type the user supplied. In python3, the two types are not interchangeable, so this method is needed to convert byte strings to unicode.

tornado.escape.**recursive_unicode**(*obj*)
Walks a simple data structure, converting byte strings to unicode.

Supports lists, tuples, and dictionaries.

Miscellaneous functions

tornado.escape.**linkify**(*text, shorten=False, extra_params=''', require_protocol=False, permitted_protocols=['http', 'https']*)
Converts plain text into HTML with links.

For example: `linkify("Hello http://tornadoweb.org!")` would return `Hello <a href="http://tornadoweb.org">http://tornadoweb.org</a>!`

Parameters:

- `shorten`: Long urls will be shortened for display.
- **`extra_params`: Extra text to include in the link tag, or a callable** taking the link as an argument and returning the extra text e.g. `linkify(text, extra_params='rel="nofollow" class="external"')`, or:

```
def extra_params_cb(url):
    if url.startswith("http://example.com"):
        return 'class="internal"'
    else:
        return 'class="external" rel="nofollow"'
linkify(text, extra_params=extra_params_cb)
```

- **`require_protocol`: Only linkify urls which include a protocol. If** this is False, urls such as www.facebook.com will also be linkified.
- **`permitted_protocols`: List (or set) of protocols which should be** linkified, e.g. `linkify(text, permitted_protocols=["http", "ftp", "mailto"])`. It is very unsafe to include protocols such as `javascript`.

tornado.escape.**squeeze**(*value*)
Replace all sequences of whitespace chars with a single space.

4.2.4 `tornado.locale` — Internationalization support

Translation methods for generating localized strings.

To load a locale and generate a translated string:

```
user_locale = tornado.locale.get("es_LA")
print user_locale.translate("Sign out")
```

`tornado.locale.get()` returns the closest matching locale, not necessarily the specific locale you requested. You can support pluralization with additional arguments to *`translate()`*, e.g.:

```
people = [...]
message = user_locale.translate(
    "%(list)s is online", "%(list)s are online", len(people))
print message % {"list": user_locale.list(people)}
```

The first string is chosen if `len(people) == 1`, otherwise the second string is chosen.

Applications should call one of *load_translations* (which uses a simple CSV format) or *load_gettext_translations* (which uses the `.mo` format supported by `gettext` and related tools). If neither method is called, the *Locale.translate* method will simply return the original string.

`tornado.locale.`**`get`**(**locale_codes*)
Returns the closest match for the given locale codes.

We iterate over all given locale codes in order. If we have a tight or a loose match for the code (e.g., "en" for "en_US"), we return the locale. Otherwise we move to the next code in the list.

By default we return `en_US` if no translations are found for any of the specified locales. You can change the default locale with *set_default_locale()*.

`tornado.locale.`**`set_default_locale`**(*code*)
Sets the default locale.

The default locale is assumed to be the language used for all strings in the system. The translations loaded from disk are mappings from the default locale to the destination locale. Consequently, you don't need to create a translation file for the default locale.

`tornado.locale.`**`load_translations`**(*directory*, *encoding=None*)
Loads translations from CSV files in a directory.

Translations are strings with optional Python-style named placeholders (e.g., `My name is %(name)s`) and their associated translations.

The directory should have translation files of the form `LOCALE.csv`, e.g. `es_GT.csv`. The CSV files should have two or three columns: string, translation, and an optional plural indicator. Plural indicators should be one of "plural" or "singular". A given string can have both singular and plural forms. For example `%(name)s liked this` may have a different verb conjugation depending on whether %(name)s is one name or a list of names. There should be two rows in the CSV file for that string, one with plural indicator "singular", and one "plural". For strings with no verbs that would change on translation, simply use "unknown" or the empty string (or don't include the column at all).

The file is read using the `csv` module in the default "excel" dialect. In this format there should not be spaces after the commas.

If no `encoding` parameter is given, the encoding will be detected automatically (among UTF-8 and UTF-16) if the file contains a byte-order marker (BOM), defaulting to UTF-8 if no BOM is present.

Example translation `es_LA.csv`:

```
"I love you","Te amo"
"%(name)s liked this","A %(name)s les gustó esto","plural"
"%(name)s liked this","A %(name)s le gustó esto","singular"
```

Changed in version 4.3: Added `encoding` parameter. Added support for BOM-based encoding detection, UTF-16, and UTF-8-with-BOM.

`tornado.locale.`**`load_gettext_translations`**(*directory*, *domain*)
Loads translations from `gettext`'s locale tree

Locale tree is similar to system's `/usr/share/locale`, like:

```
{directory}/{lang}/LC_MESSAGES/{domain}.mo
```

Three steps are required to have you app translated:

1.Generate POT translation file:

```
xgettext --language=Python --keyword=_:1,2 -d mydomain file1.py file2.html etc
```

2.Merge against existing POT file:

```
msgmerge old.po mydomain.po > new.po
```

3.Compile:

```
msgfmt mydomain.po -o {directory}/pt_BR/LC_MESSAGES/mydomain.mo
```

tornado.locale.**get_supported_locales**()
Returns a list of all the supported locale codes.

class tornado.locale.**Locale**(*code*, *translations*)
Object representing a locale.

After calling one of *load_translations* or *load_gettext_translations*, call *get* or *get_closest* to get a Locale object.

classmethod get_closest(**locale_codes*)
Returns the closest match for the given locale code.

classmethod get(*code*)
Returns the Locale for the given locale code.

If it is not supported, we raise an exception.

translate(*message*, *plural_message=None*, *count=None*)
Returns the translation for the given message for this locale.

If `plural_message` is given, you must also provide `count`. We return `plural_message` when `count != 1`, and we return the singular form for the given message when `count == 1`.

format_date(*date*, *gmt_offset=0*, *relative=True*, *shorter=False*, *full_format=False*)
Formats the given date (which should be GMT).

By default, we return a relative time (e.g., "2 minutes ago"). You can return an absolute date string with `relative=False`.

You can force a full format date ("July 10, 1980") with `full_format=True`.

This method is primarily intended for dates in the past. For dates in the future, we fall back to full format.

format_day(*date*, *gmt_offset=0*, *dow=True*)
Formats the given date as a day of week.

Example: "Monday, January 22". You can remove the day of week with `dow=False`.

list(*parts*)
Returns a comma-separated list for the given list of parts.

The format is, e.g., "A, B and C", "A and B" or just "A" for lists of size 1.

friendly_number(*value*)
Returns a comma-separated number for the given integer.

class tornado.locale.**CSVLocale**(*code*, *translations*)
Locale implementation using tornado's CSV translation format.

class tornado.locale.**GettextLocale**(*code*, *translations*)
Locale implementation using the gettext module.

pgettext(*context*, *message*, *plural_message=None*, *count=None*)
Allows to set context for translation, accepts plural forms.

Usage example:

```
pgettext("law", "right")
pgettext("good", "right")
```

Plural message example:

```
pgettext("organization", "club", "clubs", len(clubs))
pgettext("stick", "club", "clubs", len(clubs))
```

To generate POT file with context, add following options to step 1 of *load_gettext_translations* sequence:

```
xgettext [basic options] --keyword=pgettext:1c,2 --keyword=pgettext:1c,2,3
```

New in version 4.2.

4.2.5 tornado.websocket — Bidirectional communication to the browser

Implementation of the WebSocket protocol.

WebSockets allow for bidirectional communication between the browser and server.

WebSockets are supported in the current versions of all major browsers, although older versions that do not support WebSockets are still in use (refer to http://caniuse.com/websockets for details).

This module implements the final version of the WebSocket protocol as defined in RFC 6455. Certain browser versions (notably Safari 5.x) implemented an earlier draft of the protocol (known as "draft 76") and are not compatible with this module.

Changed in version 4.0: Removed support for the draft 76 protocol version.

class tornado.websocket.**WebSocketHandler**(*application*, *request*, ***kwargs*)
Subclass this class to create a basic WebSocket handler.

Override *on_message* to handle incoming messages, and use *write_message* to send messages to the client. You can also override *open* and *on_close* to handle opened and closed connections.

See http://dev.w3.org/html5/websockets/ for details on the JavaScript interface. The protocol is specified at http://tools.ietf.org/html/rfc6455.

Here is an example WebSocket handler that echos back all received messages back to the client:

```
class EchoWebSocket(tornado.websocket.WebSocketHandler):
    def open(self):
        print("WebSocket opened")

    def on_message(self, message):
        self.write_message(u"You said: " + message)

    def on_close(self):
        print("WebSocket closed")
```

WebSockets are not standard HTTP connections. The "handshake" is HTTP, but after the handshake, the protocol is message-based. Consequently, most of the Tornado HTTP facilities are not available in handlers of this

type. The only communication methods available to you are *write_message()*, *ping()*, and *close()*. Likewise, your request handler class should implement *open()* method rather than `get()` or `post()`.

If you map the handler above to `/websocket` in your application, you can invoke it in JavaScript with:

```
var ws = new WebSocket("ws://localhost:8888/websocket");
ws.onopen = function() {
   ws.send("Hello, world");
};
ws.onmessage = function (evt) {
   alert(evt.data);
};
```

This script pops up an alert box that says "You said: Hello, world".

Web browsers allow any site to open a websocket connection to any other, instead of using the same-origin policy that governs other network access from javascript. This can be surprising and is a potential security hole, so since Tornado 4.0 *WebSocketHandler* requires applications that wish to receive cross-origin websockets to opt in by overriding the *check_origin* method (see that method's docs for details). Failure to do so is the most likely cause of 403 errors when making a websocket connection.

When using a secure websocket connection (`wss://`) with a self-signed certificate, the connection from a browser may fail because it wants to show the "accept this certificate" dialog but has nowhere to show it. You must first visit a regular HTML page using the same certificate to accept it before the websocket connection will succeed.

Event handlers

`WebSocketHandler.`**`open`**(**args*, ***kwargs*)
Invoked when a new WebSocket is opened.

The arguments to *open* are extracted from the *tornado.web.URLSpec* regular expression, just like the arguments to *tornado.web.RequestHandler.get*.

`WebSocketHandler.`**`on_message`**(*message*)
Handle incoming messages on the WebSocket

This method must be overridden.

`WebSocketHandler.`**`on_close`**()
Invoked when the WebSocket is closed.

If the connection was closed cleanly and a status code or reason phrase was supplied, these values will be available as the attributes `self.close_code` and `self.close_reason`.

Changed in version 4.0: Added `close_code` and `close_reason` attributes.

`WebSocketHandler.`**`select_subprotocol`**(*subprotocols*)
Invoked when a new WebSocket requests specific subprotocols.

`subprotocols` is a list of strings identifying the subprotocols proposed by the client. This method may be overridden to return one of those strings to select it, or `None` to not select a subprotocol. Failure to select a subprotocol does not automatically abort the connection, although clients may close the connection if none of their proposed subprotocols was selected.

Output

`WebSocketHandler.`**`write_message`**(*message*, *binary=False*)
Sends the given message to the client of this Web Socket.

The message may be either a string or a dict (which will be encoded as json). If the `binary` argument is false, the message will be sent as utf8; in binary mode any byte string is allowed.

If the connection is already closed, raises *`WebSocketClosedError`*.

Changed in version 3.2: *`WebSocketClosedError`* was added (previously a closed connection would raise an `AttributeError`)

Changed in version 4.3: Returns a *`Future`* which can be used for flow control.

`WebSocketHandler.`**`close`**(*code=None*, *reason=None*)
Closes this Web Socket.

Once the close handshake is successful the socket will be closed.

`code` may be a numeric status code, taken from the values defined in RFC 6455 section 7.4.1. `reason` may be a textual message about why the connection is closing. These values are made available to the client, but are not otherwise interpreted by the websocket protocol.

Changed in version 4.0: Added the `code` and `reason` arguments.

Configuration

`WebSocketHandler.`**`check_origin`**(*origin*)
Override to enable support for allowing alternate origins.

The `origin` argument is the value of the `Origin` HTTP header, the url responsible for initiating this request. This method is not called for clients that do not send this header; such requests are always allowed (because all browsers that implement WebSockets support this header, and non-browser clients do not have the same cross-site security concerns).

Should return True to accept the request or False to reject it. By default, rejects all requests with an origin on a host other than this one.

This is a security protection against cross site scripting attacks on browsers, since WebSockets are allowed to bypass the usual same-origin policies and don't use CORS headers.

To accept all cross-origin traffic (which was the default prior to Tornado 4.0), simply override this method to always return true:

```
def check_origin(self, origin):
    return True
```

To allow connections from any subdomain of your site, you might do something like:

```
def check_origin(self, origin):
    parsed_origin = urllib.parse.urlparse(origin)
    return parsed_origin.netloc.endswith(".mydomain.com")
```

New in version 4.0.

`WebSocketHandler.`**`get_compression_options`**()
Override to return compression options for the connection.

If this method returns None (the default), compression will be disabled. If it returns a dict (even an empty one), it will be enabled. The contents of the dict may be used to control the memory and CPU usage of the compression, but no such options are currently implemented.

New in version 4.1.

`WebSocketHandler.`**`set_nodelay`**(*value*)
Set the no-delay flag for this stream.

By default, small messages may be delayed and/or combined to minimize the number of packets sent. This can sometimes cause 200-500ms delays due to the interaction between Nagle's algorithm and TCP delayed ACKs. To reduce this delay (at the expense of possibly increasing bandwidth usage), call `self.set_nodelay(True)` once the websocket connection is established.

See *BaseIOStream.set_nodelay* for additional details.

New in version 3.1.

Other

`WebSocketHandler.`**`ping`**(*data*)
Send ping frame to the remote end.

`WebSocketHandler.`**`on_pong`**(*data*)
Invoked when the response to a ping frame is received.

exception `tornado.websocket.`**`WebSocketClosedError`**
Raised by operations on a closed connection.

New in version 3.2.

Client-side support

`tornado.websocket.`**`websocket_connect`**(*url*, *io_loop=None*, *callback=None*, *connect_timeout=None*, *on_message_callback=None*, *compression_options=None*)
Client-side websocket support.

Takes a url and returns a Future whose result is a *WebSocketClientConnection*.

`compression_options` is interpreted in the same way as the return value of *WebSocketHandler.get_compression_options*.

The connection supports two styles of operation. In the coroutine style, the application typically calls *read_message* in a loop:

```
conn = yield websocket_connect(url)
while True:
    msg = yield conn.read_message()
    if msg is None: break
    # Do something with msg
```

In the callback style, pass an `on_message_callback` to `websocket_connect`. In both styles, a message of `None` indicates that the connection has been closed.

Changed in version 3.2: Also accepts `HTTPRequest` objects in place of urls.

Changed in version 4.1: Added `compression_options` and `on_message_callback`. The `io_loop` argument is deprecated.

class `tornado.websocket.`**`WebSocketClientConnection`**(*io_loop*, *request*, *on_message_callback=None*, *compression_options=None*)
WebSocket client connection.

This class should not be instantiated directly; use the *websocket_connect* function instead.

`close`(*code=None*, *reason=None*)
Closes the websocket connection.

`code` and `reason` are documented under *WebSocketHandler.close*.

New in version 3.2.

Changed in version 4.0: Added the `code` and `reason` arguments.

write_message(*message*, *binary=False*)
Sends a message to the WebSocket server.

read_message(*callback=None*)
Reads a message from the WebSocket server.

If on_message_callback was specified at WebSocket initialization, this function will never return messages

Returns a future whose result is the message, or None if the connection is closed. If a callback argument is given it will be called with the future when it is ready.

4.3 HTTP servers and clients

4.3.1 `tornado.httpserver` — Non-blocking HTTP server

A non-blocking, single-threaded HTTP server.

Typical applications have little direct interaction with the *HTTPServer* class except to start a server at the beginning of the process (and even that is often done indirectly via *tornado.web.Application.listen*).

Changed in version 4.0: The `HTTPRequest` class that used to live in this module has been moved to *tornado.httputil.HTTPServerRequest*. The old name remains as an alias.

HTTP Server

class tornado.httpserver.**HTTPServer**(**args*, ***kwargs*)
A non-blocking, single-threaded HTTP server.

A server is defined by a subclass of *HTTPServerConnectionDelegate*, or, for backwards compatibility, a callback that takes an *HTTPServerRequest* as an argument. The delegate is usually a *tornado.web.Application*.

HTTPServer supports keep-alive connections by default (automatically for HTTP/1.1, or for HTTP/1.0 when the client requests `Connection: keep-alive`).

If `xheaders` is `True`, we support the `X-Real-Ip`/`X-Forwarded-For` and `X-Scheme`/`X-Forwarded-Proto` headers, which override the remote IP and URI scheme/protocol for all requests. These headers are useful when running Tornado behind a reverse proxy or load balancer. The `protocol` argument can also be set to `https` if Tornado is run behind an SSL-decoding proxy that does not set one of the supported `xheaders`.

To make this server serve SSL traffic, send the `ssl_options` keyword argument with an `ssl.SSLContext` object. For compatibility with older versions of Python `ssl_options` may also be a dictionary of keyword arguments for the `ssl.wrap_socket` method.:

```
ssl_ctx = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ssl_ctx.load_cert_chain(os.path.join(data_dir, "mydomain.crt"),
                        os.path.join(data_dir, "mydomain.key"))
HTTPServer(applicaton, ssl_options=ssl_ctx)
```

HTTPServer initialization follows one of three patterns (the initialization methods are defined on *tornado.tcpserver.TCPServer*):

1. *listen*: simple single-process:

```
server = HTTPServer(app)
server.listen(8888)
IOLoop.current().start()
```

In many cases, *tornado.web.Application.listen* can be used to avoid the need to explicitly create the *HTTPServer*.

2. *bind*/*start*: simple multi-process:

```
server = HTTPServer(app)
server.bind(8888)
server.start(0)  # Forks multiple sub-processes
IOLoop.current().start()
```

When using this interface, an *IOLoop* must *not* be passed to the *HTTPServer* constructor. *start* will always start the server on the default singleton *IOLoop*.

3. *add_sockets*: advanced multi-process:

```
sockets = tornado.netutil.bind_sockets(8888)
tornado.process.fork_processes(0)
server = HTTPServer(app)
server.add_sockets(sockets)
IOLoop.current().start()
```

The *add_sockets* interface is more complicated, but it can be used with *tornado.process.fork_processes* to give you more flexibility in when the fork happens. *add_sockets* can also be used in single-process servers if you want to create your listening sockets in some way other than *tornado.netutil.bind_sockets*.

Changed in version 4.0: Added `decompress_request`, `chunk_size`, `max_header_size`, `idle_connection_timeout`, `body_timeout`, `max_body_size` arguments. Added support for *HTTPServerConnectionDelegate* instances as `request_callback`.

Changed in version 4.1: *HTTPServerConnectionDelegate.start_request* is now called with two arguments `(server_conn, request_conn)` (in accordance with the documentation) instead of one `(request_conn)`.

Changed in version 4.2: *HTTPServer* is now a subclass of *tornado.util.Configurable*.

4.3.2 `tornado.httpclient` — Asynchronous HTTP client

Blocking and non-blocking HTTP client interfaces.

This module defines a common interface shared by two implementations, `simple_httpclient` and `curl_httpclient`. Applications may either instantiate their chosen implementation class directly or use the *AsyncHTTPClient* class from this module, which selects an implementation that can be overridden with the *AsyncHTTPClient.configure* method.

The default implementation is `simple_httpclient`, and this is expected to be suitable for most users' needs. However, some applications may wish to switch to `curl_httpclient` for reasons such as the following:

- `curl_httpclient` has some features not found in `simple_httpclient`, including support for HTTP proxies and the ability to use a specified network interface.
- `curl_httpclient` is more likely to be compatible with sites that are not-quite-compliant with the HTTP spec, or sites that use little-exercised features of HTTP.
- `curl_httpclient` is faster.

- `curl_httpclient` was the default prior to Tornado 2.0.

Note that if you are using `curl_httpclient`, it is highly recommended that you use a recent version of `libcurl` and `pycurl`. Currently the minimum supported version of libcurl is 7.21.1, and the minimum version of pycurl is 7.18.2. It is highly recommended that your `libcurl` installation is built with asynchronous DNS resolver (threaded or c-ares), otherwise you may encounter various problems with request timeouts (for more information, see http://curl.haxx.se/libcurl/c/curl_easy_setopt.html#CURLOPTCONNECTTIMEOUTMS and comments in curl_httpclient.py).

To select `curl_httpclient`, call *`AsyncHTTPClient.configure`* at startup:

```
AsyncHTTPClient.configure("tornado.curl_httpclient.CurlAsyncHTTPClient")
```

HTTP client interfaces

class `tornado.httpclient.`**`HTTPClient`**(*async_client_class=None*, ***kwargs*)
A blocking HTTP client.

This interface is provided for convenience and testing; most applications that are running an IOLoop will want to use *`AsyncHTTPClient`* instead. Typical usage looks like this:

```
http_client = httpclient.HTTPClient()
try:
    response = http_client.fetch("http://www.google.com/")
    print response.body
except httpclient.HTTPError as e:
    # HTTPError is raised for non-200 responses; the response
    # can be found in e.response.
    print("Error: " + str(e))
except Exception as e:
    # Other errors are possible, such as IOError.
    print("Error: " + str(e))
http_client.close()
```

`close`()
Closes the HTTPClient, freeing any resources used.

`fetch`(*request*, ***kwargs*)
Executes a request, returning an *`HTTPResponse`*.

The request may be either a string URL or an *`HTTPRequest`* object. If it is a string, we construct an *`HTTPRequest`* using any additional kwargs: `HTTPRequest(request, **kwargs)`

If an error occurs during the fetch, we raise an *`HTTPError`* unless the `raise_error` keyword argument is set to False.

class `tornado.httpclient.`**`AsyncHTTPClient`**
An non-blocking HTTP client.

Example usage:

```
def handle_response(response):
    if response.error:
        print "Error:", response.error
    else:
        print response.body

http_client = AsyncHTTPClient()
http_client.fetch("http://www.google.com/", handle_response)
```

The constructor for this class is magic in several respects: It actually creates an instance of an implementation-specific subclass, and instances are reused as a kind of pseudo-singleton (one per *IOLoop*). The keyword argument `force_instance=True` can be used to suppress this singleton behavior. Unless `force_instance=True` is used, no arguments other than `io_loop` should be passed to the *AsyncHTTPClient* constructor. The implementation subclass as well as arguments to its constructor can be set with the static method *configure()*

All *AsyncHTTPClient* implementations support a `defaults` keyword argument, which can be used to set default values for *HTTPRequest* attributes. For example:

```
AsyncHTTPClient.configure(
    None, defaults=dict(user_agent="MyUserAgent"))
# or with force_instance:
client = AsyncHTTPClient(force_instance=True,
    defaults=dict(user_agent="MyUserAgent"))
```

Changed in version 4.1: The `io_loop` argument is deprecated.

close()
Destroys this HTTP client, freeing any file descriptors used.

This method is **not needed in normal use** due to the way that *AsyncHTTPClient* objects are transparently reused. `close()` is generally only necessary when either the *IOLoop* is also being closed, or the `force_instance=True` argument was used when creating the *AsyncHTTPClient*.

No other methods may be called on the *AsyncHTTPClient* after `close()`.

fetch(*request*, *callback=None*, *raise_error=True*, ***kwargs*)
Executes a request, asynchronously returning an *HTTPResponse*.

The request may be either a string URL or an *HTTPRequest* object. If it is a string, we construct an *HTTPRequest* using any additional kwargs: `HTTPRequest(request, **kwargs)`

This method returns a *Future* whose result is an *HTTPResponse*. By default, the `Future` will raise an *HTTPError* if the request returned a non-200 response code (other errors may also be raised if the server could not be contacted). Instead, if `raise_error` is set to False, the response will always be returned regardless of the response code.

If a `callback` is given, it will be invoked with the *HTTPResponse*. In the callback interface, *HTTPError* is not automatically raised. Instead, you must check the response's `error` attribute or call its *rethrow* method.

classmethod configure(*impl*, ***kwargs*)
Configures the *AsyncHTTPClient* subclass to use.

`AsyncHTTPClient()` actually creates an instance of a subclass. This method may be called with either a class object or the fully-qualified name of such a class (or `None` to use the default, `SimpleAsyncHTTPClient`)

If additional keyword arguments are given, they will be passed to the constructor of each subclass instance created. The keyword argument `max_clients` determines the maximum number of simultaneous *fetch()* operations that can execute in parallel on each *IOLoop*. Additional arguments may be supported depending on the implementation class in use.

Example:

```
AsyncHTTPClient.configure("tornado.curl_httpclient.CurlAsyncHTTPClient")
```

Request objects

class tornado.httpclient.**HTTPRequest**(*url*, *method='GET'*, *headers=None*, *body=None*, *auth_username=None*, *auth_password=None*, *auth_mode=None*, *connect_timeout=None*, *request_timeout=None*, *if_modified_since=None*, *follow_redirects=None*, *max_redirects=None*, *user_agent=None*, *use_gzip=None*, *network_interface=None*, *streaming_callback=None*, *header_callback=None*, *prepare_curl_callback=None*, *proxy_host=None*, *proxy_port=None*, *proxy_username=None*, *proxy_password=None*, *allow_nonstandard_methods=None*, *validate_cert=None*, *ca_certs=None*, *allow_ipv6=None*, *client_key=None*, *client_cert=None*, *body_producer=None*, *expect_100_continue=False*, *decompress_response=None*, *ssl_options=None*)

HTTP client request object.

All parameters except `url` are optional.

Parameters

- **url** (*string*) – URL to fetch
- **method** (*string*) – HTTP method, e.g. "GET" or "POST"
- **headers** (*HTTPHeaders* or `dict`) – Additional HTTP headers to pass on the request
- **body** – HTTP request body as a string (byte or unicode; if unicode the utf-8 encoding will be used)
- **body_producer** – Callable used for lazy/asynchronous request bodies. It is called with one argument, a `write` function, and should return a *Future*. It should call the write function with new data as it becomes available. The write function returns a *Future* which can be used for flow control. Only one of `body` and `body_producer` may be specified. `body_producer` is not supported on `curl_httpclient`. When using `body_producer` it is recommended to pass a `Content-Length` in the headers as otherwise chunked encoding will be used, and many servers do not support chunked encoding on requests. New in Tornado 4.0
- **auth_username** (*string*) – Username for HTTP authentication
- **auth_password** (*string*) – Password for HTTP authentication
- **auth_mode** (*string*) – Authentication mode; default is "basic". Allowed values are implementation-defined; `curl_httpclient` supports "basic" and "digest"; `simple_httpclient` only supports "basic"
- **connect_timeout** (*float*) – Timeout for initial connection in seconds
- **request_timeout** (*float*) – Timeout for entire request in seconds
- **if_modified_since** (`datetime` or `float`) – Timestamp for `If-Modified-Since` header
- **follow_redirects** (*bool*) – Should redirects be followed automatically or return the 3xx response?
- **max_redirects** (*int*) – Limit for `follow_redirects`
- **user_agent** (*string*) – String to send as `User-Agent` header

- **decompress_response** (*bool*) – Request a compressed response from the server and decompress it after downloading. Default is True. New in Tornado 4.0.
- **use_gzip** (*bool*) – Deprecated alias for `decompress_response` since Tornado 4.0.
- **network_interface** (*string*) – Network interface to use for request. `curl_httpclient` only; see note below.
- **streaming_callback** (*callable*) – If set, `streaming_callback` will be run with each chunk of data as it is received, and `HTTPResponse.body` and `HTTPResponse.buffer` will be empty in the final response.
- **header_callback** (*callable*) – If set, `header_callback` will be run with each header line as it is received (including the first line, e.g. `HTTP/1.0 200 OK\r\n`, and a final line containing only `\r\n`. All lines include the trailing newline characters). `HTTPResponse.headers` will be empty in the final response. This is most useful in conjunction with `streaming_callback`, because it's the only way to get access to header data while the request is in progress.
- **prepare_curl_callback** (*callable*) – If set, will be called with a `pycurl.Curl` object to allow the application to make additional `setopt` calls.
- **proxy_host** (*string*) – HTTP proxy hostname. To use proxies, `proxy_host` and `proxy_port` must be set; `proxy_username` and `proxy_pass` are optional. Proxies are currently only supported with `curl_httpclient`.
- **proxy_port** (*int*) – HTTP proxy port
- **proxy_username** (*string*) – HTTP proxy username
- **proxy_password** (*string*) – HTTP proxy password
- **allow_nonstandard_methods** (*bool*) – Allow unknown values for `method` argument?
- **validate_cert** (*bool*) – For HTTPS requests, validate the server's certificate?
- **ca_certs** (*string*) – filename of CA certificates in PEM format, or None to use defaults. See note below when used with `curl_httpclient`.
- **client_key** (*string*) – Filename for client SSL key, if any. See note below when used with `curl_httpclient`.
- **client_cert** (*string*) – Filename for client SSL certificate, if any. See note below when used with `curl_httpclient`.
- **ssl_options** (*ssl.SSLContext*) – `ssl.SSLContext` object for use in `simple_httpclient` (unsupported by `curl_httpclient`). Overrides `validate_cert`, `ca_certs`, `client_key`, and `client_cert`.
- **allow_ipv6** (*bool*) – Use IPv6 when available? Default is true.
- **expect_100_continue** (*bool*) – If true, send the `Expect: 100-continue` header and wait for a continue response before sending the request body. Only supported with simple_httpclient.

Note: When using `curl_httpclient` certain options may be inherited by subsequent fetches because `pycurl` does not allow them to be cleanly reset. This applies to the `ca_certs`, `client_key`, `client_cert`, and `network_interface` arguments. If you use these options, you should pass them on every request (you don't have to always use the same values, but it's not possible to mix requests that specify these options with ones that use the defaults).

New in version 3.1: The `auth_mode` argument.

New in version 4.0: The `body_producer` and `expect_100_continue` arguments.

New in version 4.2: The `ssl_options` argument.

Response objects

class `tornado.httpclient.`**`HTTPResponse`**(*request*, *code*, *headers=None*, *buffer=None*, *effective_url=None*, *error=None*, *request_time=None*, *time_info=None*, *reason=None*)

HTTP Response object.

Attributes:

- request: HTTPRequest object
- code: numeric HTTP status code, e.g. 200 or 404
- reason: human-readable reason phrase describing the status code
- headers: *`tornado.httputil.HTTPHeaders`* object
- effective_url: final location of the resource after following any redirects
- buffer: `cStringIO` object for response body
- body: response body as string (created on demand from `self.buffer`)
- error: Exception object, if any
- request_time: seconds from request start to finish
- time_info: dictionary of diagnostic timing information from the request. Available data are subject to change, but currently uses timings available from http://curl.haxx.se/libcurl/c/curl_easy_getinfo.html, plus `queue`, which is the delay (if any) introduced by waiting for a slot under *`AsyncHTTPClient`*'s `max_clients` setting.

`rethrow`()

If there was an error on the request, raise an *`HTTPError`*.

Exceptions

exception `tornado.httpclient.`**`HTTPError`**(*code*, *message=None*, *response=None*)

Exception thrown for an unsuccessful HTTP request.

Attributes:

- `code` - HTTP error integer error code, e.g. 404. Error code 599 is used when no HTTP response was received, e.g. for a timeout.
- `response` - *`HTTPResponse`* object, if any.

Note that if `follow_redirects` is False, redirects become HTTPErrors, and you can look at `error.response.headers['Location']` to see the destination of the redirect.

Command-line interface

This module provides a simple command-line interface to fetch a url using Tornado's HTTP client. Example usage:

```
# Fetch the url and print its body
python -m tornado.httpclient http://www.google.com

# Just print the headers
python -m tornado.httpclient --print_headers --print_body=false http://www.google.com
```

Implementations

class tornado.simple_httpclient.**SimpleAsyncHTTPClient**
Non-blocking HTTP client with no external dependencies.

This class implements an HTTP 1.1 client on top of Tornado's IOStreams. Some features found in the curl-based AsyncHTTPClient are not yet supported. In particular, proxies are not supported, connections are not reused, and callers cannot select the network interface to be used.

initialize(*io_loop*, *max_clients=10*, *hostname_mapping=None*, *max_buffer_size=104857600*, *resolver=None*, *defaults=None*, *max_header_size=None*, *max_body_size=None*)
Creates a AsyncHTTPClient.

Only a single AsyncHTTPClient instance exists per IOLoop in order to provide limitations on the number of pending connections. `force_instance=True` may be used to suppress this behavior.

Note that because of this implicit reuse, unless `force_instance` is used, only the first call to the constructor actually uses its arguments. It is recommended to use the `configure` method instead of the constructor to ensure that arguments take effect.

`max_clients` is the number of concurrent requests that can be in progress; when this limit is reached additional requests will be queued. Note that time spent waiting in this queue still counts against the `request_timeout`.

`hostname_mapping` is a dictionary mapping hostnames to IP addresses. It can be used to make local DNS changes when modifying system-wide settings like `/etc/hosts` is not possible or desirable (e.g. in unittests).

`max_buffer_size` (default 100MB) is the number of bytes that can be read into memory at once. `max_body_size` (defaults to `max_buffer_size`) is the largest response body that the client will accept. Without a `streaming_callback`, the smaller of these two limits applies; with a `streaming_callback` only `max_body_size` does.

Changed in version 4.2: Added the `max_body_size` argument.

class tornado.curl_httpclient.**CurlAsyncHTTPClient**(*io_loop*, *max_clients=10*, *defaults=None*)
`libcurl`-based HTTP client.

4.3.3 tornado.httputil — Manipulate HTTP headers and URLs

HTTP utility code shared by clients and servers.

This module also defines the *HTTPServerRequest* class which is exposed via *tornado.web.RequestHandler.request*.

class tornado.httputil.**HTTPHeaders**(**args*, ***kwargs*)
A dictionary that maintains `Http-Header-Case` for all keys.

Supports multiple values per key via a pair of new methods, *add()* and *get_list()*. The regular dictionary interface returns a single value per key, with multiple values joined by a comma.

```
>>> h = HTTPHeaders({"content-type": "text/html"})
>>> list(h.keys())
['Content-Type']
>>> h["Content-Type"]
'text/html'
```

```
>>> h.add("Set-Cookie", "A=B")
>>> h.add("Set-Cookie", "C=D")
>>> h["set-cookie"]
'A=B,C=D'
>>> h.get_list("set-cookie")
['A=B', 'C=D']
```

```
>>> for (k,v) in sorted(h.get_all()):
...    print('%s: %s' % (k,v))
...
Content-Type: text/html
Set-Cookie: A=B
Set-Cookie: C=D
```

add(*name*, *value*)
Adds a new value for the given key.

get_list(*name*)
Returns all values for the given header as a list.

get_all()
Returns an iterable of all (name, value) pairs.

If a header has multiple values, multiple pairs will be returned with the same name.

parse_line(*line*)
Updates the dictionary with a single header line.

```
>>> h = HTTPHeaders()
>>> h.parse_line("Content-Type: text/html")
>>> h.get('content-type')
'text/html'
```

classmethod parse(*headers*)
Returns a dictionary from HTTP header text.

```
>>> h = HTTPHeaders.parse("Content-Type: text/html\r\nContent-Length: 42\r\n")
>>> sorted(h.items())
[('Content-Length', '42'), ('Content-Type', 'text/html')]
```

class tornado.httputil.**HTTPServerRequest**(*method=None*, *uri=None*, *version='HTTP/1.0'*, *headers=None*, *body=None*, *host=None*, *files=None*, *connection=None*, *start_line=None*)
A single HTTP request.

All attributes are type `str` unless otherwise noted.

method
HTTP request method, e.g. "GET" or "POST"

uri
The requested uri.

path
The path portion of *uri*

query
The query portion of *uri*

version
HTTP version specified in request, e.g. "HTTP/1.1"

headers
HTTPHeaders dictionary-like object for request headers. Acts like a case-insensitive dictionary with additional methods for repeated headers.

body
Request body, if present, as a byte string.

remote_ip
Client's IP address as a string. If `HTTPServer.xheaders` is set, will pass along the real IP address provided by a load balancer in the `X-Real-Ip` or `X-Forwarded-For` header.

Changed in version 3.1: The list format of `X-Forwarded-For` is now supported.

protocol
The protocol used, either "http" or "https". If `HTTPServer.xheaders` is set, will pass along the protocol used by a load balancer if reported via an `X-Scheme` header.

host
The requested hostname, usually taken from the `Host` header.

arguments
GET/POST arguments are available in the arguments property, which maps arguments names to lists of values (to support multiple values for individual names). Names are of type `str`, while arguments are byte strings. Note that this is different from *RequestHandler.get_argument*, which returns argument values as unicode strings.

query_arguments
Same format as `arguments`, but contains only arguments extracted from the query string.

New in version 3.2.

body_arguments
Same format as `arguments`, but contains only arguments extracted from the request body.

New in version 3.2.

files
File uploads are available in the files property, which maps file names to lists of *HTTPFile*.

connection
An HTTP request is attached to a single HTTP connection, which can be accessed through the "connection" attribute. Since connections are typically kept open in HTTP/1.1, multiple requests can be handled sequentially on a single connection.

Changed in version 4.0: Moved from `tornado.httpserver.HTTPRequest`.

supports_http_1_1()
Returns True if this request supports HTTP/1.1 semantics.

Deprecated since version 4.0: Applications are less likely to need this information with the introduction of *HTTPConnection*. If you still need it, access the `version` attribute directly.

cookies
A dictionary of Cookie.Morsel objects.

write(*chunk*, *callback=None*)
Writes the given chunk to the response stream.

Deprecated since version 4.0: Use `request.connection` and the *HTTPConnection* methods to write the response.

finish()
Finishes this HTTP request on the open connection.

Deprecated since version 4.0: Use `request.connection` and the *HTTPConnection* methods to write the response.

full_url()
Reconstructs the full URL for this request.

request_time()
Returns the amount of time it took for this request to execute.

get_ssl_certificate(*binary_form=False*)
Returns the client's SSL certificate, if any.

To use client certificates, the HTTPServer's `ssl.SSLContext.verify_mode` field must be set, e.g.:

```
ssl_ctx = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ssl_ctx.load_cert_chain("foo.crt", "foo.key")
ssl_ctx.load_verify_locations("cacerts.pem")
ssl_ctx.verify_mode = ssl.CERT_REQUIRED
server = HTTPServer(app, ssl_options=ssl_ctx)
```

By default, the return value is a dictionary (or None, if no client certificate is present). If `binary_form` is true, a DER-encoded form of the certificate is returned instead. See SSLSocket.getpeercert() in the standard library for more details. http://docs.python.org/library/ssl.html#sslsocket-objects

exception `tornado.httputil.`**HTTPInputError**
Exception class for malformed HTTP requests or responses from remote sources.

New in version 4.0.

exception `tornado.httputil.`**HTTPOutputError**
Exception class for errors in HTTP output.

New in version 4.0.

class `tornado.httputil.`**HTTPServerConnectionDelegate**
Implement this interface to handle requests from *HTTPServer*.

New in version 4.0.

start_request(*server_conn*, *request_conn*)
This method is called by the server when a new request has started.

Parameters

- **server_conn** – is an opaque object representing the long-lived (e.g. tcp-level) connection.
- **request_conn** – is a *HTTPConnection* object for a single request/response exchange.

This method should return a *HTTPMessageDelegate*.

on_close(*server_conn*)
This method is called when a connection has been closed.

Parameters **server_conn** – is a server connection that has previously been passed to `start_request`.

class tornado.httputil.**HTTPMessageDelegate**
Implement this interface to handle an HTTP request or response.

New in version 4.0.

headers_received(*start_line*, *headers*)
Called when the HTTP headers have been received and parsed.

Parameters

- **start_line** – a *RequestStartLine* or *ResponseStartLine* depending on whether this is a client or server message.
- **headers** – a *HTTPHeaders* instance.

Some *HTTPConnection* methods can only be called during headers_received.

May return a *Future*; if it does the body will not be read until it is done.

data_received(*chunk*)
Called when a chunk of data has been received.

May return a *Future* for flow control.

finish()
Called after the last chunk of data has been received.

on_connection_close()
Called if the connection is closed without finishing the request.

If headers_received is called, either finish or on_connection_close will be called, but not both.

class tornado.httputil.**HTTPConnection**
Applications use this interface to write their responses.

New in version 4.0.

write_headers(*start_line*, *headers*, *chunk=None*, *callback=None*)
Write an HTTP header block.

Parameters

- **start_line** – a *RequestStartLine* or *ResponseStartLine*.
- **headers** – a *HTTPHeaders* instance.
- **chunk** – the first (optional) chunk of data. This is an optimization so that small responses can be written in the same call as their headers.
- **callback** – a callback to be run when the write is complete.

The version field of start_line is ignored.

Returns a *Future* if no callback is given.

write(*chunk*, *callback=None*)
Writes a chunk of body data.

The callback will be run when the write is complete. If no callback is given, returns a Future.

finish()
Indicates that the last body data has been written.

tornado.httputil.**url_concat**(*url*, *args*)
Concatenate url and arguments regardless of whether url has existing query parameters.

`args` may be either a dictionary or a list of key-value pairs (the latter allows for multiple values with the same key.

```
>>> url_concat("http://example.com/foo", dict(c="d"))
'http://example.com/foo?c=d'
>>> url_concat("http://example.com/foo?a=b", dict(c="d"))
'http://example.com/foo?a=b&c=d'
>>> url_concat("http://example.com/foo?a=b", [("c", "d"), ("c", "d2")])
'http://example.com/foo?a=b&c=d&c=d2'
```

class `tornado.httputil.`**`HTTPFile`**
Represents a file uploaded via a form.

For backwards compatibility, its instance attributes are also accessible as dictionary keys.

- `filename`
- `body`
- `content_type`

`tornado.httputil.`**`parse_body_arguments`**(*content_type*, *body*, *arguments*, *files*, *headers=None*)
Parses a form request body.

Supports `application/x-www-form-urlencoded` and `multipart/form-data`. The `content_type` parameter should be a string and `body` should be a byte string. The `arguments` and `files` parameters are dictionaries that will be updated with the parsed contents.

`tornado.httputil.`**`parse_multipart_form_data`**(*boundary*, *data*, *arguments*, *files*)
Parses a `multipart/form-data` body.

The `boundary` and `data` parameters are both byte strings. The dictionaries given in the arguments and files parameters will be updated with the contents of the body.

`tornado.httputil.`**`format_timestamp`**(*ts*)
Formats a timestamp in the format used by HTTP.

The argument may be a numeric timestamp as returned by `time.time`, a time tuple as returned by `time.gmtime`, or a `datetime.datetime` object.

```
>>> format_timestamp(1359312200)
'Sun, 27 Jan 2013 18:43:20 GMT'
```

class `tornado.httputil.`**`RequestStartLine`**
RequestStartLine(method, path, version)

`method`
Alias for field number 0

`path`
Alias for field number 1

`version`
Alias for field number 2

`tornado.httputil.`**`parse_request_start_line`**(*line*)
Returns a (method, path, version) tuple for an HTTP 1.x request line.

The response is a `collections.namedtuple`.

```
>>> parse_request_start_line("GET /foo HTTP/1.1")
RequestStartLine(method='GET', path='/foo', version='HTTP/1.1')
```

class tornado.httputil.**ResponseStartLine**
ResponseStartLine(version, code, reason)

code
Alias for field number 1

reason
Alias for field number 2

version
Alias for field number 0

tornado.httputil.**parse_response_start_line**(*line*)
Returns a (version, code, reason) tuple for an HTTP 1.x response line.

The response is a collections.namedtuple.

```
>>> parse_response_start_line("HTTP/1.1 200 OK")
ResponseStartLine(version='HTTP/1.1', code=200, reason='OK')
```

tornado.httputil.**split_host_and_port**(*netloc*)
Returns `(host, port)` tuple from `netloc`.

Returned `port` will be `None` if not present.

New in version 4.1.

tornado.httputil.**parse_cookie**(*cookie*)
Parse a `Cookie` HTTP header into a dict of name/value pairs.

This function attempts to mimic browser cookie parsing behavior; it specifically does not follow any of the cookie-related RFCs (because browsers don't either).

The algorithm used is identical to that used by Django version 1.9.10.

New in version 4.4.2.

4.3.4 tornado.http1connection – HTTP/1.x client/server implementation

Client and server implementations of HTTP/1.x.

New in version 4.0.

class tornado.http1connection.**HTTP1ConnectionParameters**(*no_keep_alive=False*, *chunk_size=None*, *max_header_size=None*, *header_timeout=None*, *max_body_size=None*, *body_timeout=None*, *decompress=False*)
Parameters for *HTTP1Connection* and *HTTP1ServerConnection*.

Parameters

- **no_keep_alive** (*bool*) – If true, always close the connection after one request.
- **chunk_size** (*int*) – how much data to read into memory at once
- **max_header_size** (*int*) – maximum amount of data for HTTP headers
- **header_timeout** (*float*) – how long to wait for all headers (seconds)
- **max_body_size** (*int*) – maximum amount of data for body

- **body_timeout** (*float*) – how long to wait while reading body (seconds)
- **decompress** (*bool*) – if true, decode incoming `Content-Encoding: gzip`

class `tornado.http1connection.`**HTTP1Connection**(*stream*, *is_client*, *params=None*, *context=None*)

Implements the HTTP/1.x protocol.

This class can be on its own for clients, or via *HTTP1ServerConnection* for servers.

Parameters

- **stream** – an *IOStream*
- **is_client** (*bool*) – client or server
- **params** – a *HTTP1ConnectionParameters* instance or `None`
- **context** – an opaque application-defined object that can be accessed as `connection.context`.

read_response(*delegate*)

Read a single HTTP response.

Typical client-mode usage is to write a request using *write_headers*, *write*, and *finish*, and then call `read_response`.

Parameters **delegate** – a *HTTPMessageDelegate*

Returns a *Future* that resolves to None after the full response has been read.

set_close_callback(*callback*)

Sets a callback that will be run when the connection is closed.

Deprecated since version 4.0: Use *HTTPMessageDelegate.on_connection_close* instead.

detach()

Take control of the underlying stream.

Returns the underlying *IOStream* object and stops all further HTTP processing. May only be called during *HTTPMessageDelegate.headers_received*. Intended for implementing protocols like websockets that tunnel over an HTTP handshake.

set_body_timeout(*timeout*)

Sets the body timeout for a single request.

Overrides the value from *HTTP1ConnectionParameters*.

set_max_body_size(*max_body_size*)

Sets the body size limit for a single request.

Overrides the value from *HTTP1ConnectionParameters*.

write_headers(*start_line*, *headers*, *chunk=None*, *callback=None*)

Implements *HTTPConnection.write_headers*.

write(*chunk*, *callback=None*)

Implements *HTTPConnection.write*.

For backwards compatibility is is allowed but deprecated to skip *write_headers* and instead call *write()* with a pre-encoded header block.

finish()

Implements *HTTPConnection.finish*.

class tornado.http1connection.**HTTP1ServerConnection**(*stream*, *params=None*, *context=None*)

An HTTP/1.x server.

Parameters

- **stream** – an *IOStream*
- **params** – a *HTTP1ConnectionParameters* or None
- **context** – an opaque application-defined object that is accessible as `connection.context`

close()

Closes the connection.

Returns a *Future* that resolves after the serving loop has exited.

start_serving(*delegate*)

Starts serving requests on this connection.

Parameters **delegate** – a *HTTPServerConnectionDelegate*

4.4 Asynchronous networking

4.4.1 tornado.ioloop — Main event loop

An I/O event loop for non-blocking sockets.

Typical applications will use a single *IOLoop* object, in the *IOLoop.instance* singleton. The *IOLoop.start* method should usually be called at the end of the `main()` function. Atypical applications may use more than one *IOLoop*, such as one *IOLoop* per thread, or per `unittest` case.

In addition to I/O events, the *IOLoop* can also schedule time-based events. *IOLoop.add_timeout* is a non-blocking alternative to `time.sleep`.

IOLoop objects

class tornado.ioloop.**IOLoop**

A level-triggered I/O loop.

We use `epoll` (Linux) or `kqueue` (BSD and Mac OS X) if they are available, or else we fall back on select(). If you are implementing a system that needs to handle thousands of simultaneous connections, you should use a system that supports either `epoll` or `kqueue`.

Example usage for a simple TCP server:

```
import errno
import functools
import tornado.ioloop
import socket

def connection_ready(sock, fd, events):
    while True:
        try:
            connection, address = sock.accept()
        except socket.error as e:
            if e.args[0] not in (errno.EWOULDBLOCK, errno.EAGAIN):
                raise
```

```
                return
            connection.setblocking(0)
            handle_connection(connection, address)

    if __name__ == '__main__':
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
        sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        sock.setblocking(0)
        sock.bind(("", port))
        sock.listen(128)

        io_loop = tornado.ioloop.IOLoop.current()
        callback = functools.partial(connection_ready, sock)
        io_loop.add_handler(sock.fileno(), callback, io_loop.READ)
        io_loop.start()
```

By default, a newly-constructed *IOLoop* becomes the thread's current *IOLoop*, unless there already is a current *IOLoop*. This behavior can be controlled with the `make_current` argument to the *IOLoop* constructor: if `make_current=True`, the new *IOLoop* will always try to become current and it raises an error if there is already a current instance. If `make_current=False`, the new *IOLoop* will not try to become current.

Changed in version 4.2: Added the `make_current` keyword argument to the *IOLoop* constructor.

Running an IOLoop

static `IOLoop.`**`current`**(*instance=True*)
Returns the current thread's *IOLoop*.

If an *IOLoop* is currently running or has been marked as current by *make_current*, returns that instance. If there is no current *IOLoop*, returns *IOLoop.instance()* (i.e. the main thread's *IOLoop*, creating one if necessary) if `instance` is true.

In general you should use *IOLoop.current* as the default when constructing an asynchronous object, and use *IOLoop.instance* when you mean to communicate to the main thread from a different one.

Changed in version 4.1: Added `instance` argument to control the fallback to *IOLoop.instance()*.

`IOLoop.`**`make_current`**()
Makes this the *IOLoop* for the current thread.

An *IOLoop* automatically becomes current for its thread when it is started, but it is sometimes useful to call *make_current* explicitly before starting the *IOLoop*, so that code run at startup time can find the right instance.

Changed in version 4.1: An *IOLoop* created while there is no current *IOLoop* will automatically become current.

static `IOLoop.`**`instance`**()
Returns a global *IOLoop* instance.

Most applications have a single, global *IOLoop* running on the main thread. Use this method to get this instance from another thread. In most other cases, it is better to use *current()* to get the current thread's *IOLoop*.

static `IOLoop.`**`initialized`**()
Returns true if the singleton instance has been created.

`IOLoop.`**`install`**()
Installs this *IOLoop* object as the singleton instance.

This is normally not necessary as *instance()* will create an *IOLoop* on demand, but you may want to call *install* to use a custom subclass of *IOLoop*.

When using an *IOLoop* subclass, *install* must be called prior to creating any objects that implicitly create their own *IOLoop* (e.g., *tornado.httpclient.AsyncHTTPClient*).

static IOLoop.**clear_instance**()
Clear the global *IOLoop* instance.

New in version 4.0.

IOLoop.**start**()
Starts the I/O loop.

The loop will run until one of the callbacks calls *stop()*, which will make the loop stop after the current event iteration completes.

IOLoop.**stop**()
Stop the I/O loop.

If the event loop is not currently running, the next call to *start()* will return immediately.

To use asynchronous methods from otherwise-synchronous code (such as unit tests), you can start and stop the event loop like this:

```
ioloop = IOLoop()
async_method(ioloop=ioloop, callback=ioloop.stop)
ioloop.start()
```

`ioloop.start()` will return after `async_method` has run its callback, whether that callback was invoked before or after `ioloop.start`.

Note that even after *stop* has been called, the *IOLoop* is not completely stopped until *IOLoop.start* has also returned. Some work that was scheduled before the call to *stop* may still be run before the *IOLoop* shuts down.

IOLoop.**run_sync**(*func*, *timeout=None*)
Starts the *IOLoop*, runs the given function, and stops the loop.

The function must return either a yieldable object or `None`. If the function returns a yieldable object, the *IOLoop* will run until the yieldable is resolved (and *run_sync()* will return the yieldable's result). If it raises an exception, the *IOLoop* will stop and the exception will be re-raised to the caller.

The keyword-only argument `timeout` may be used to set a maximum duration for the function. If the timeout expires, a `TimeoutError` is raised.

This method is useful in conjunction with *tornado.gen.coroutine* to allow asynchronous calls in a `main()` function:

```
@gen.coroutine
def main():
    # do stuff...

if __name__ == '__main__':
    IOLoop.current().run_sync(main)
```

Changed in version 4.3: Returning a non-`None`, non-yieldable value is now an error.

IOLoop.**close**(*all_fds=False*)
Closes the *IOLoop*, freeing any resources used.

If `all_fds` is true, all file descriptors registered on the IOLoop will be closed (not just the ones created by the *IOLoop* itself).

Many applications will only use a single *IOLoop* that runs for the entire lifetime of the process. In that case closing the *IOLoop* is not necessary since everything will be cleaned up when the process exits. *IOLoop.close* is provided mainly for scenarios such as unit tests, which create and destroy a large number of IOLoops.

An *IOLoop* must be completely stopped before it can be closed. This means that *IOLoop.stop()* must be called *and* *IOLoop.start()* must be allowed to return before attempting to call *IOLoop.close()*. Therefore the call to *close* will usually appear just after the call to *start* rather than near the call to *stop*.

Changed in version 3.1: If the *IOLoop* implementation supports non-integer objects for "file descriptors", those objects will have their close method when all_fds is true.

I/O events

IOLoop.**add_handler**(*fd*, *handler*, *events*)
Registers the given handler to receive the given events for fd.

The fd argument may either be an integer file descriptor or a file-like object with a fileno() method (and optionally a close() method, which may be called when the *IOLoop* is shut down).

The events argument is a bitwise or of the constants IOLoop.READ, IOLoop.WRITE, and IOLoop.ERROR.

When an event occurs, handler(fd, events) will be run.

Changed in version 4.0: Added the ability to pass file-like objects in addition to raw file descriptors.

IOLoop.**update_handler**(*fd*, *events*)
Changes the events we listen for fd.

Changed in version 4.0: Added the ability to pass file-like objects in addition to raw file descriptors.

IOLoop.**remove_handler**(*fd*)
Stop listening for events on fd.

Changed in version 4.0: Added the ability to pass file-like objects in addition to raw file descriptors.

Callbacks and timeouts

IOLoop.**add_callback**(*callback*, **args*, ***kwargs*)
Calls the given callback on the next I/O loop iteration.

It is safe to call this method from any thread at any time, except from a signal handler. Note that this is the **only** method in *IOLoop* that makes this thread-safety guarantee; all other interaction with the *IOLoop* must be done from that *IOLoop*'s thread. *add_callback()* may be used to transfer control from other threads to the *IOLoop*'s thread.

To add a callback from a signal handler, see *add_callback_from_signal*.

IOLoop.**add_callback_from_signal**(*callback*, **args*, ***kwargs*)
Calls the given callback on the next I/O loop iteration.

Safe for use from a Python signal handler; should not be used otherwise.

Callbacks added with this method will be run without any *stack_context*, to avoid picking up the context of the function that was interrupted by the signal.

IOLoop.**add_future**(*future*, *callback*)
Schedules a callback on the IOLoop when the given *Future* is finished.

The callback is invoked with one argument, the *Future*.

IOLoop.**add_timeout**(*deadline*, *callback*, **args*, ***kwargs*)
Runs the `callback` at the time `deadline` from the I/O loop.

Returns an opaque handle that may be passed to *remove_timeout* to cancel.

`deadline` may be a number denoting a time (on the same scale as *IOLoop.time*, normally `time.time`), or a `datetime.timedelta` object for a deadline relative to the current time. Since Tornado 4.0, *call_later* is a more convenient alternative for the relative case since it does not require a timedelta object.

Note that it is not safe to call *add_timeout* from other threads. Instead, you must use *add_callback* to transfer control to the *IOLoop*'s thread, and then call *add_timeout* from there.

Subclasses of IOLoop must implement either *add_timeout* or *call_at*; the default implementations of each will call the other. *call_at* is usually easier to implement, but subclasses that wish to maintain compatibility with Tornado versions prior to 4.0 must use *add_timeout* instead.

Changed in version 4.0: Now passes through `*args` and `**kwargs` to the callback.

IOLoop.**call_at**(*when*, *callback*, **args*, ***kwargs*)
Runs the `callback` at the absolute time designated by `when`.

`when` must be a number using the same reference point as *IOLoop.time*.

Returns an opaque handle that may be passed to *remove_timeout* to cancel. Note that unlike the `asyncio` method of the same name, the returned object does not have a `cancel()` method.

See *add_timeout* for comments on thread-safety and subclassing.

New in version 4.0.

IOLoop.**call_later**(*delay*, *callback*, **args*, ***kwargs*)
Runs the `callback` after `delay` seconds have passed.

Returns an opaque handle that may be passed to *remove_timeout* to cancel. Note that unlike the `asyncio` method of the same name, the returned object does not have a `cancel()` method.

See *add_timeout* for comments on thread-safety and subclassing.

New in version 4.0.

IOLoop.**remove_timeout**(*timeout*)
Cancels a pending timeout.

The argument is a handle as returned by *add_timeout*. It is safe to call *remove_timeout* even if the callback has already been run.

IOLoop.**spawn_callback**(*callback*, **args*, ***kwargs*)
Calls the given callback on the next IOLoop iteration.

Unlike all other callback-related methods on IOLoop, `spawn_callback` does not associate the callback with its caller's `stack_context`, so it is suitable for fire-and-forget callbacks that should not interfere with the caller.

New in version 4.0.

IOLoop.**time**()
Returns the current time according to the *IOLoop*'s clock.

The return value is a floating-point number relative to an unspecified time in the past.

By default, the *IOLoop*'s time function is `time.time`. However, it may be configured to use e.g. `time.monotonic` instead. Calls to *add_timeout* that pass a number instead of a `datetime.timedelta` should use this function to compute the appropriate time, so they can work no matter what time function is chosen.

class tornado.ioloop.**PeriodicCallback**(*callback*, *callback_time*, *io_loop=None*)
Schedules the given callback to be called periodically.

The callback is called every `callback_time` milliseconds. Note that the timeout is given in milliseconds, while most other time-related functions in Tornado use seconds.

If the callback runs for longer than `callback_time` milliseconds, subsequent invocations will be skipped to get back on schedule.

start must be called after the *PeriodicCallback* is created.

Changed in version 4.1: The `io_loop` argument is deprecated.

start()
Starts the timer.

stop()
Stops the timer.

is_running()
Return True if this *PeriodicCallback* has been started.

New in version 4.1.

Debugging and error handling

IOLoop.**handle_callback_exception**(*callback*)
This method is called whenever a callback run by the *IOLoop* throws an exception.

By default simply logs the exception as an error. Subclasses may override this method to customize reporting of exceptions.

The exception itself is not passed explicitly, but is available in `sys.exc_info`.

IOLoop.**set_blocking_signal_threshold**(*seconds*, *action*)
Sends a signal if the *IOLoop* is blocked for more than `s` seconds.

Pass `seconds=None` to disable. Requires Python 2.6 on a unixy platform.

The action parameter is a Python signal handler. Read the documentation for the `signal` module for more information. If `action` is None, the process will be killed if it is blocked for too long.

IOLoop.**set_blocking_log_threshold**(*seconds*)
Logs a stack trace if the *IOLoop* is blocked for more than `s` seconds.

Equivalent to `set_blocking_signal_threshold(seconds, self.log_stack)`

IOLoop.**log_stack**(*signal*, *frame*)
Signal handler to log the stack trace of the current thread.

For use with *set_blocking_signal_threshold*.

Methods for subclasses

IOLoop.**initialize**(*make_current=None*)

IOLoop.**close_fd**(*fd*)
Utility method to close an `fd`.

If `fd` is a file-like object, we close it directly; otherwise we use `os.close`.

This method is provided for use by *IOLoop* subclasses (in implementations of `IOLoop.close(all_fds=True)` and should not generally be used by application code.

New in version 4.0.

`IOLoop.`**`split_fd`**(*fd*)
Returns an (fd, obj) pair from an `fd` parameter.

We accept both raw file descriptors and file-like objects as input to *add_handler* and related methods. When a file-like object is passed, we must retain the object itself so we can close it correctly when the *IOLoop* shuts down, but the poller interfaces favor file descriptors (they will accept file-like objects and call `fileno()` for you, but they always return the descriptor itself).

This method is provided for use by *IOLoop* subclasses and should not generally be used by application code.

New in version 4.0.

4.4.2 `tornado.iostream` — Convenient wrappers for non-blocking sockets

Utility classes to write to and read from non-blocking files and sockets.

Contents:

- *BaseIOStream*: Generic interface for reading and writing.
- *IOStream*: Implementation of BaseIOStream using non-blocking sockets.
- *SSLIOStream*: SSL-aware version of IOStream.
- *PipeIOStream*: Pipe-based IOStream implementation.

Base class

class `tornado.iostream.`**`BaseIOStream`**(*io_loop=None*, *max_buffer_size=None*, *read_chunk_size=None*, *max_write_buffer_size=None*)
A utility class to write to and read from a non-blocking file or socket.

We support a non-blocking `write()` and a family of `read_*()` methods. All of the methods take an optional `callback` argument and return a *Future* only if no callback is given. When the operation completes, the callback will be run or the *Future* will resolve with the data read (or `None` for `write()`). All outstanding `Futures` will resolve with a *StreamClosedError* when the stream is closed; users of the callback interface will be notified via *BaseIOStream.set_close_callback* instead.

When a stream is closed due to an error, the IOStream's `error` attribute contains the exception object.

Subclasses must implement *fileno*, *close_fd*, *write_to_fd*, *read_from_fd*, and optionally *get_fd_error*.

BaseIOStream constructor.

Parameters

- **`io_loop`** – The *IOLoop* to use; defaults to *IOLoop.current*. Deprecated since Tornado 4.1.
- **`max_buffer_size`** – Maximum amount of incoming data to buffer; defaults to 100MB.
- **`read_chunk_size`** – Amount of data to read at one time from the underlying transport; defaults to 64KB.
- **`max_write_buffer_size`** – Amount of outgoing data to buffer; defaults to unlimited.

Changed in version 4.0: Add the `max_write_buffer_size` parameter. Changed default `read_chunk_size` to 64KB.

Main interface

`BaseIOStream.`**`write`**(*data*, *callback=None*)
Asynchronously write the given data to this stream.

If `callback` is given, we call it when all of the buffered write data has been successfully written to the stream. If there was previously buffered write data and an old write callback, that callback is simply overwritten with this new callback.

If no `callback` is given, this method returns a *`Future`* that resolves (with a result of `None`) when the write has been completed. If *`write`* is called again before that *`Future`* has resolved, the previous future will be orphaned and will never resolve.

Changed in version 4.0: Now returns a *`Future`* if no callback is given.

`BaseIOStream.`**`read_bytes`**(*num_bytes*, *callback=None*, *streaming_callback=None*, *partial=False*)
Asynchronously read a number of bytes.

If a `streaming_callback` is given, it will be called with chunks of data as they become available, and the final result will be empty. Otherwise, the result is all the data that was read. If a callback is given, it will be run with the data as an argument; if not, this method returns a *`Future`*.

If `partial` is true, the callback is run as soon as we have any bytes to return (but never more than `num_bytes`)

Changed in version 4.0: Added the `partial` argument. The callback argument is now optional and a *`Future`* will be returned if it is omitted.

`BaseIOStream.`**`read_until`**(*delimiter*, *callback=None*, *max_bytes=None*)
Asynchronously read until we have found the given delimiter.

The result includes all the data read including the delimiter. If a callback is given, it will be run with the data as an argument; if not, this method returns a *`Future`*.

If `max_bytes` is not None, the connection will be closed if more than `max_bytes` bytes have been read and the delimiter is not found.

Changed in version 4.0: Added the `max_bytes` argument. The `callback` argument is now optional and a *`Future`* will be returned if it is omitted.

`BaseIOStream.`**`read_until_regex`**(*regex*, *callback=None*, *max_bytes=None*)
Asynchronously read until we have matched the given regex.

The result includes the data that matches the regex and anything that came before it. If a callback is given, it will be run with the data as an argument; if not, this method returns a *`Future`*.

If `max_bytes` is not None, the connection will be closed if more than `max_bytes` bytes have been read and the regex is not satisfied.

Changed in version 4.0: Added the `max_bytes` argument. The `callback` argument is now optional and a *`Future`* will be returned if it is omitted.

`BaseIOStream.`**`read_until_close`**(*callback=None*, *streaming_callback=None*)
Asynchronously reads all data from the socket until it is closed.

If a `streaming_callback` is given, it will be called with chunks of data as they become available, and the final result will be empty. Otherwise, the result is all the data that was read. If a callback is given, it will be run with the data as an argument; if not, this method returns a *`Future`*.

Note that if a `streaming_callback` is used, data will be read from the socket as quickly as it becomes available; there is no way to apply backpressure or cancel the reads. If flow control or cancellation are desired, use a loop with *read_bytes(partial=True)* instead.

Changed in version 4.0: The callback argument is now optional and a *Future* will be returned if it is omitted.

`BaseIOStream.`**`close`**(*exc_info=False*)
Close this stream.

If `exc_info` is true, set the `error` attribute to the current exception from `sys.exc_info` (or if `exc_info` is a tuple, use that instead of `sys.exc_info`).

`BaseIOStream.`**`set_close_callback`**(*callback*)
Call the given callback when the stream is closed.

This is not necessary for applications that use the *Future* interface; all outstanding `Futures` will resolve with a *StreamClosedError* when the stream is closed.

`BaseIOStream.`**`closed`**()
Returns true if the stream has been closed.

`BaseIOStream.`**`reading`**()
Returns true if we are currently reading from the stream.

`BaseIOStream.`**`writing`**()
Returns true if we are currently writing to the stream.

`BaseIOStream.`**`set_nodelay`**(*value*)
Sets the no-delay flag for this stream.

By default, data written to TCP streams may be held for a time to make the most efficient use of bandwidth (according to Nagle's algorithm). The no-delay flag requests that data be written as soon as possible, even if doing so would consume additional bandwidth.

This flag is currently defined only for TCP-based `IOStreams`.

New in version 3.1.

Methods for subclasses

`BaseIOStream.`**`fileno`**()
Returns the file descriptor for this stream.

`BaseIOStream.`**`close_fd`**()
Closes the file underlying this stream.

`close_fd` is called by *BaseIOStream* and should not be called elsewhere; other users should call *close* instead.

`BaseIOStream.`**`write_to_fd`**(*data*)
Attempts to write `data` to the underlying file.

Returns the number of bytes written.

`BaseIOStream.`**`read_from_fd`**()
Attempts to read from the underlying file.

Returns `None` if there was nothing to read (the socket returned `EWOULDBLOCK` or equivalent), otherwise returns the data. When possible, should return no more than `self.read_chunk_size` bytes at a time.

`BaseIOStream.`**`get_fd_error`**()
Returns information about any error on the underlying file.

This method is called after the *IOLoop* has signaled an error on the file descriptor, and should return an Exception (such as `socket.error` with additional information, or None if no such information is available.

Implementations

class `tornado.iostream.`**IOStream**(*socket*, **args*, ***kwargs*)

Socket-based *IOStream* implementation.

This class supports the read and write methods from *BaseIOStream* plus a *connect* method.

The `socket` parameter may either be connected or unconnected. For server operations the socket is the result of calling `socket.accept`. For client operations the socket is created with `socket.socket`, and may either be connected before passing it to the *IOStream* or connected with *IOStream.connect*.

A very simple (and broken) HTTP client using this class:

```
import tornado.ioloop
import tornado.iostream
import socket

def send_request():
    stream.write(b"GET / HTTP/1.0\r\nHost: friendfeed.com\r\n\r\n")
    stream.read_until(b"\r\n\r\n", on_headers)

def on_headers(data):
    headers = {}
    for line in data.split(b"\r\n"):
        parts = line.split(b":")
        if len(parts) == 2:
            headers[parts[0].strip()] = parts[1].strip()
    stream.read_bytes(int(headers[b"Content-Length"]), on_body)

def on_body(data):
    print(data)
    stream.close()
    tornado.ioloop.IOLoop.current().stop()

if __name__ == '__main__':
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
    stream = tornado.iostream.IOStream(s)
    stream.connect(("friendfeed.com", 80), send_request)
    tornado.ioloop.IOLoop.current().start()
```

connect(*address*, *callback=None*, *server_hostname=None*)

Connects the socket to a remote address without blocking.

May only be called if the socket passed to the constructor was not previously connected. The address parameter is in the same format as for `socket.connect` for the type of socket passed to the IOStream constructor, e.g. an `(ip, port)` tuple. Hostnames are accepted here, but will be resolved synchronously and block the IOLoop. If you have a hostname instead of an IP address, the *TCPClient* class is recommended instead of calling this method directly. *TCPClient* will do asynchronous DNS resolution and handle both IPv4 and IPv6.

If `callback` is specified, it will be called with no arguments when the connection is completed; if not this method returns a *Future* (whose result after a successful connection will be the stream itself).

In SSL mode, the `server_hostname` parameter will be used for certificate validation (unless disabled in the `ssl_options`) and SNI (if supported; requires Python 2.7.9+).

Note that it is safe to call *IOStream.write* while the connection is pending, in which case the data will be written as soon as the connection is ready. Calling *IOStream* read methods before the socket is connected works on some platforms but is non-portable.

Changed in version 4.0: If no callback is given, returns a *Future*.

Changed in version 4.2: SSL certificates are validated by default; pass `ssl_options=dict(cert_reqs=ssl.CERT_NONE)` or a suitably-configured `ssl.SSLContext` to the *SSLIOStream* constructor to disable.

start_tls(*server_side*, *ssl_options=None*, *server_hostname=None*)
Convert this *IOStream* to an *SSLIOStream*.

This enables protocols that begin in clear-text mode and switch to SSL after some initial negotiation (such as the `STARTTLS` extension to SMTP and IMAP).

This method cannot be used if there are outstanding reads or writes on the stream, or if there is any data in the IOStream's buffer (data in the operating system's socket buffer is allowed). This means it must generally be used immediately after reading or writing the last clear-text data. It can also be used immediately after connecting, before any reads or writes.

The `ssl_options` argument may be either an `ssl.SSLContext` object or a dictionary of keyword arguments for the `ssl.wrap_socket` function. The `server_hostname` argument will be used for certificate validation unless disabled in the `ssl_options`.

This method returns a *Future* whose result is the new *SSLIOStream*. After this method has been called, any other operation on the original stream is undefined.

If a close callback is defined on this stream, it will be transferred to the new stream.

New in version 4.0.

Changed in version 4.2: SSL certificates are validated by default; pass `ssl_options=dict(cert_reqs=ssl.CERT_NONE)` or a suitably-configured `ssl.SSLContext` to disable.

class `tornado.iostream.`**SSLIOStream**(**args*, ***kwargs*)
A utility class to write to and read from a non-blocking SSL socket.

If the socket passed to the constructor is already connected, it should be wrapped with:

```
ssl.wrap_socket(sock, do_handshake_on_connect=False, **kwargs)
```

before constructing the *SSLIOStream*. Unconnected sockets will be wrapped when *IOStream.connect* is finished.

The `ssl_options` keyword argument may either be an `ssl.SSLContext` object or a dictionary of keywords arguments for `ssl.wrap_socket`

wait_for_handshake(*callback=None*)
Wait for the initial SSL handshake to complete.

If a `callback` is given, it will be called with no arguments once the handshake is complete; otherwise this method returns a *Future* which will resolve to the stream itself after the handshake is complete.

Once the handshake is complete, information such as the peer's certificate and NPN/ALPN selections may be accessed on `self.socket`.

This method is intended for use on server-side streams or after using *IOStream.start_tls*; it should not be used with *IOStream.connect* (which already waits for the handshake to complete). It may only be called once per stream.

New in version 4.2.

class tornado.iostream.**PipeIOStream**(*fd*, **args*, ***kwargs*)
Pipe-based *IOStream* implementation.

The constructor takes an integer file descriptor (such as one returned by os.pipe) rather than an open file object. Pipes are generally one-way, so a *PipeIOStream* can be used for reading or writing but not both.

Exceptions

exception tornado.iostream.**StreamBufferFullError**
Exception raised by *IOStream* methods when the buffer is full.

exception tornado.iostream.**StreamClosedError**(*real_error=None*)
Exception raised by *IOStream* methods when the stream is closed.

Note that the close callback is scheduled to run *after* other callbacks on the stream (to allow for buffered data to be processed), so you may see this error before you see the close callback.

The real_error attribute contains the underlying error that caused the stream to close (if any).

Changed in version 4.3: Added the real_error attribute.

exception tornado.iostream.**UnsatisfiableReadError**
Exception raised when a read cannot be satisfied.

Raised by read_until and read_until_regex with a max_bytes argument.

4.4.3 tornado.netutil — Miscellaneous network utilities

Miscellaneous network utility code.

tornado.netutil.**bind_sockets**(*port*, *address=None*, *family=<AddressFamily.AF_UNSPEC: 0>*, *backlog=128*, *flags=None*, *reuse_port=False*)
Creates listening sockets bound to the given port and address.

Returns a list of socket objects (multiple sockets are returned if the given address maps to multiple IP addresses, which is most common for mixed IPv4 and IPv6 use).

Address may be either an IP address or hostname. If it's a hostname, the server will listen on all IP addresses associated with the name. Address may be an empty string or None to listen on all available interfaces. Family may be set to either socket.AF_INET or socket.AF_INET6 to restrict to IPv4 or IPv6 addresses, otherwise both will be used if available.

The backlog argument has the same meaning as for socket.listen().

flags is a bitmask of AI_* flags to getaddrinfo, like socket.AI_PASSIVE | socket.AI_NUMERICHOST.

reuse_port option sets SO_REUSEPORT option for every socket in the list. If your platform doesn't support this option ValueError will be raised.

tornado.netutil.**bind_unix_socket**(*file*, *mode=384*, *backlog=128*)
Creates a listening unix socket.

If a socket with the given name already exists, it will be deleted. If any other file with that name exists, an exception will be raised.

Returns a socket object (not a list of socket objects like *bind_sockets*)

tornado.netutil.**add_accept_handler**(*sock*, *callback*, *io_loop=None*)
Adds an *IOLoop* event handler to accept new connections on sock.

When a connection is accepted, `callback(connection, address)` will be run (`connection` is a socket object, and `address` is the address of the other end of the connection). Note that this signature is different from the `callback(fd, events)` signature used for *IOLoop* handlers.

Changed in version 4.1: The `io_loop` argument is deprecated.

`tornado.netutil.`**`is_valid_ip`**(*ip*)
Returns true if the given string is a well-formed IP address.

Supports IPv4 and IPv6.

class `tornado.netutil.`**`Resolver`**
Configurable asynchronous DNS resolver interface.

By default, a blocking implementation is used (which simply calls `socket.getaddrinfo`). An alternative implementation can be chosen with the *Resolver.configure* class method:

```
Resolver.configure('tornado.netutil.ThreadedResolver')
```

The implementations of this interface included with Tornado are

- *tornado.netutil.BlockingResolver*
- *tornado.netutil.ThreadedResolver*
- *tornado.netutil.OverrideResolver*
- *tornado.platform.twisted.TwistedResolver*
- *tornado.platform.caresresolver.CaresResolver*

`resolve`(*host*, *port*, *family=<AddressFamily.AF_UNSPEC: 0>*, *callback=None*)
Resolves an address.

The `host` argument is a string which may be a hostname or a literal IP address.

Returns a *Future* whose result is a list of (family, address) pairs, where address is a tuple suitable to pass to `socket.connect` (i.e. a `(host, port)` pair for IPv4; additional fields may be present for IPv6). If a `callback` is passed, it will be run with the result as an argument when it is complete.

Raises **`IOError`** – if the address cannot be resolved.

Changed in version 4.4: Standardized all implementations to raise `IOError`.

`close`()
Closes the *Resolver*, freeing any resources used.

New in version 3.1.

class `tornado.netutil.`**`ExecutorResolver`**
Resolver implementation using a `concurrent.futures.Executor`.

Use this instead of *ThreadedResolver* when you require additional control over the executor being used.

The executor will be shut down when the resolver is closed unless `close_resolver=False`; use this if you want to reuse the same executor elsewhere.

Changed in version 4.1: The `io_loop` argument is deprecated.

class `tornado.netutil.`**`BlockingResolver`**
Default *Resolver* implementation, using `socket.getaddrinfo`.

The *IOLoop* will be blocked during the resolution, although the callback will not be run until the next *IOLoop* iteration.

class tornado.netutil.**ThreadedResolver**
Multithreaded non-blocking *Resolver* implementation.

Requires the concurrent.futures package to be installed (available in the standard library since Python 3.2, installable with pip install futures in older versions).

The thread pool size can be configured with:

```
Resolver.configure('tornado.netutil.ThreadedResolver',
                   num_threads=10)
```

Changed in version 3.1: All ThreadedResolvers share a single thread pool, whose size is set by the first one to be created.

class tornado.netutil.**OverrideResolver**
Wraps a resolver with a mapping of overrides.

This can be used to make local DNS changes (e.g. for testing) without modifying system-wide settings.

The mapping can contain either host strings or host-port pairs.

tornado.netutil.**ssl_options_to_context**(*ssl_options*)
Try to convert an ssl_options dictionary to an SSLContext object.

The ssl_options dictionary contains keywords to be passed to ssl.wrap_socket. In Python 2.7.9+, ssl.SSLContext objects can be used instead. This function converts the dict form to its SSLContext equivalent, and may be used when a component which accepts both forms needs to upgrade to the SSLContext version to use features like SNI or NPN.

tornado.netutil.**ssl_wrap_socket**(*socket*, *ssl_options*, *server_hostname=None*, ***kwargs*)
Returns an ssl.SSLSocket wrapping the given socket.

ssl_options may be either an ssl.SSLContext object or a dictionary (as accepted by *ssl_options_to_context*). Additional keyword arguments are passed to wrap_socket (either the SSLContext method or the ssl module function as appropriate).

4.4.4 tornado.tcpclient — IOStream connection factory

A non-blocking TCP connection factory.

class tornado.tcpclient.**TCPClient**(*resolver=None*, *io_loop=None*)
A non-blocking TCP connection factory.

Changed in version 4.1: The io_loop argument is deprecated.

connect(*host*, *port*, *af=<AddressFamily.AF_UNSPEC: 0>*, *ssl_options=None*, *max_buffer_size=None*)
Connect to the given host and port.

Asynchronously returns an *IOStream* (or *SSLIOStream* if ssl_options is not None).

4.4.5 tornado.tcpserver — Basic IOStream-based TCP server

A non-blocking, single-threaded TCP server.

class tornado.tcpserver.**TCPServer**(*io_loop=None*, *ssl_options=None*, *max_buffer_size=None*, *read_chunk_size=None*)
A non-blocking, single-threaded TCP server.

To use *TCPServer*, define a subclass which overrides the *handle_stream* method.

To make this server serve SSL traffic, send the `ssl_options` keyword argument with an `ssl.SSLContext` object. For compatibility with older versions of Python `ssl_options` may also be a dictionary of keyword arguments for the `ssl.wrap_socket` method.:

```
ssl_ctx = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ssl_ctx.load_cert_chain(os.path.join(data_dir, "mydomain.crt"),
                        os.path.join(data_dir, "mydomain.key"))
TCPServer(ssl_options=ssl_ctx)
```

TCPServer initialization follows one of three patterns:

1. *listen*: simple single-process:

```
server = TCPServer()
server.listen(8888)
IOLoop.current().start()
```

2. *bind*/*start*: simple multi-process:

```
server = TCPServer()
server.bind(8888)
server.start(0)  # Forks multiple sub-processes
IOLoop.current().start()
```

When using this interface, an *IOLoop* must *not* be passed to the *TCPServer* constructor. *start* will always start the server on the default singleton *IOLoop*.

3. *add_sockets*: advanced multi-process:

```
sockets = bind_sockets(8888)
tornado.process.fork_processes(0)
server = TCPServer()
server.add_sockets(sockets)
IOLoop.current().start()
```

The *add_sockets* interface is more complicated, but it can be used with *tornado.process.fork_processes* to give you more flexibility in when the fork happens. *add_sockets* can also be used in single-process servers if you want to create your listening sockets in some way other than *bind_sockets*.

New in version 3.1: The `max_buffer_size` argument.

listen(*port*, *address=''*)
Starts accepting connections on the given port.

This method may be called more than once to listen on multiple ports. *listen* takes effect immediately; it is not necessary to call *TCPServer.start* afterwards. It is, however, necessary to start the *IOLoop*.

add_sockets(*sockets*)
Makes this server start accepting connections on the given sockets.

The `sockets` parameter is a list of socket objects such as those returned by *bind_sockets*. *add_sockets* is typically used in combination with that method and *tornado.process.fork_processes* to provide greater control over the initialization of a multi-process server.

add_socket(*socket*)
Singular version of *add_sockets*. Takes a single socket object.

bind(*port*, *address=None*, *family=<AddressFamily.AF_UNSPEC: 0>*, *backlog=128*, *reuse_port=False*)
Binds this server to the given port on the given address.

To start the server, call *start*. If you want to run this server in a single process, you can call *listen* as a shortcut to the sequence of *bind* and *start* calls.

Address may be either an IP address or hostname. If it's a hostname, the server will listen on all IP addresses associated with the name. Address may be an empty string or None to listen on all available interfaces. Family may be set to either `socket.AF_INET` or `socket.AF_INET6` to restrict to IPv4 or IPv6 addresses, otherwise both will be used if available.

The `backlog` argument has the same meaning as for `socket.listen`. The `reuse_port` argument has the same meaning as for *bind_sockets*.

This method may be called multiple times prior to *start* to listen on multiple ports or interfaces.

Changed in version 4.4: Added the `reuse_port` argument.

start(*num_processes=1*)
Starts this server in the *IOLoop*.

By default, we run the server in this process and do not fork any additional child process.

If num_processes is `None` or <= 0, we detect the number of cores available on this machine and fork that number of child processes. If num_processes is given and > 1, we fork that specific number of sub-processes.

Since we use processes and not threads, there is no shared memory between any server code.

Note that multiple processes are not compatible with the autoreload module (or the `autoreload=True` option to *tornado.web.Application* which defaults to True when `debug=True`). When using multiple processes, no IOLoops can be created or referenced until after the call to `TCPServer.start(n)`.

stop()
Stops listening for new connections.

Requests currently in progress may still continue after the server is stopped.

handle_stream(*stream*, *address*)
Override to handle a new *IOStream* from an incoming connection.

This method may be a coroutine; if so any exceptions it raises asynchronously will be logged. Accepting of incoming connections will not be blocked by this coroutine.

If this *TCPServer* is configured for SSL, `handle_stream` may be called before the SSL handshake has completed. Use *SSLIOStream.wait_for_handshake* if you need to verify the client's certificate or use NPN/ALPN.

Changed in version 4.2: Added the option for this method to be a coroutine.

4.5 Coroutines and concurrency

4.5.1 `tornado.gen` — Simplify asynchronous code

`tornado.gen` is a generator-based interface to make it easier to work in an asynchronous environment. Code using the `gen` module is technically asynchronous, but it is written as a single generator instead of a collection of separate functions.

For example, the following asynchronous handler:

```
class AsyncHandler(RequestHandler):
    @asynchronous
    def get(self):
        http_client = AsyncHTTPClient()
        http_client.fetch("http://example.com",
                          callback=self.on_fetch)

    def on_fetch(self, response):
        do_something_with_response(response)
        self.render("template.html")
```

could be written with `gen` as:

```
class GenAsyncHandler(RequestHandler):
    @gen.coroutine
    def get(self):
        http_client = AsyncHTTPClient()
        response = yield http_client.fetch("http://example.com")
        do_something_with_response(response)
        self.render("template.html")
```

Most asynchronous functions in Tornado return a *`Future`*; yielding this object returns its *`result`*.

You can also yield a list or dict of `Futures`, which will be started at the same time and run in parallel; a list or dict of results will be returned when they are all finished:

```
@gen.coroutine
def get(self):
    http_client = AsyncHTTPClient()
    response1, response2 = yield [http_client.fetch(url1),
                                  http_client.fetch(url2)]
    response_dict = yield dict(response3=http_client.fetch(url3),
                               response4=http_client.fetch(url4))
    response3 = response_dict['response3']
    response4 = response_dict['response4']
```

If the `singledispatch` library is available (standard in Python 3.4, available via the singledispatch package on older versions), additional types of objects may be yielded. Tornado includes support for `asyncio.Future` and Twisted's `Deferred` class when `tornado.platform.asyncio` and `tornado.platform.twisted` are imported. See the *`convert_yielded`* function to extend this mechanism.

Changed in version 3.2: Dict support added.

Changed in version 4.1: Support added for yielding `asyncio` Futures and Twisted Deferreds via `singledispatch`.

Decorators

`tornado.gen.`**`coroutine`**(*func*, *replace_callback=True*)

Decorator for asynchronous generators.

Any generator that yields objects from this module must be wrapped in either this decorator or *`engine`*.

Coroutines may "return" by raising the special exception *`Return(value)`*. In Python 3.3+, it is also possible for the function to simply use the `return value` statement (prior to Python 3.3 generators were not allowed to also return values). In all versions of Python a coroutine that simply wishes to exit early may use the `return` statement without a value.

Functions with this decorator return a *Future*. Additionally, they may be called with a `callback` keyword argument, which will be invoked with the future's result when it resolves. If the coroutine fails, the callback will not be run and an exception will be raised into the surrounding *StackContext*. The `callback` argument is not visible inside the decorated function; it is handled by the decorator itself.

From the caller's perspective, `@gen.coroutine` is similar to the combination of `@return_future` and `@gen.engine`.

> **Warning:** When exceptions occur inside a coroutine, the exception information will be stored in the *Future* object. You must examine the result of the *Future* object, or the exception may go unnoticed by your code. This means yielding the function if called from another coroutine, using something like *IOLoop.run_sync* for top-level calls, or passing the *Future* to *IOLoop.add_future*.

`tornado.gen.`**`engine`**(*func*)

Callback-oriented decorator for asynchronous generators.

This is an older interface; for new code that does not need to be compatible with versions of Tornado older than 3.0 the *coroutine* decorator is recommended instead.

This decorator is similar to *coroutine*, except it does not return a *Future* and the `callback` argument is not treated specially.

In most cases, functions decorated with *engine* should take a `callback` argument and invoke it with their result when they are finished. One notable exception is the *RequestHandler HTTP verb methods*, which use `self.finish()` in place of a callback argument.

Utility functions

exception `tornado.gen.`**`Return`**(*value=None*)

Special exception to return a value from a *coroutine*.

If this exception is raised, its value argument is used as the result of the coroutine:

```
@gen.coroutine
def fetch_json(url):
    response = yield AsyncHTTPClient().fetch(url)
    raise gen.Return(json_decode(response.body))
```

In Python 3.3, this exception is no longer necessary: the `return` statement can be used directly to return a value (previously `yield` and `return` with a value could not be combined in the same function).

By analogy with the return statement, the value argument is optional, but it is never necessary to `raise gen.Return()`. The `return` statement can be used with no arguments instead.

`tornado.gen.`**`with_timeout`**(*timeout*, *future*, *io_loop=None*, *quiet_exceptions=()*)

Wraps a *Future* (or other yieldable object) in a timeout.

Raises *TimeoutError* if the input future does not complete before `timeout`, which may be specified in any form allowed by *IOLoop.add_timeout* (i.e. a `datetime.timedelta` or an absolute time relative to *IOLoop.time*)

If the wrapped *Future* fails after it has timed out, the exception will be logged unless it is of a type contained in `quiet_exceptions` (which may be an exception type or a sequence of types).

Does not support *YieldPoint* subclasses.

New in version 4.0.

Changed in version 4.1: Added the `quiet_exceptions` argument and the logging of unhandled exceptions.

Changed in version 4.4: Added support for yieldable objects other than *Future*.

exception tornado.gen.**TimeoutError**
Exception raised by with_timeout.

tornado.gen.**sleep**(*duration*)
Return a *Future* that resolves after the given number of seconds.

When used with yield in a coroutine, this is a non-blocking analogue to time.sleep (which should not be used in coroutines because it is blocking):

```
yield gen.sleep(0.5)
```

Note that calling this function on its own does nothing; you must wait on the *Future* it returns (usually by yielding it).

New in version 4.1.

tornado.gen.**moment**
A special object which may be yielded to allow the IOLoop to run for one iteration.

This is not needed in normal use but it can be helpful in long-running coroutines that are likely to yield Futures that are ready instantly.

Usage: yield gen.moment

New in version 4.0.

class tornado.gen.**WaitIterator**(**args*, ***kwargs*)
Provides an iterator to yield the results of futures as they finish.

Yielding a set of futures like this:

```
results = yield [future1, future2]
```

pauses the coroutine until both future1 and future2 return, and then restarts the coroutine with the results of both futures. If either future is an exception, the expression will raise that exception and all the results will be lost.

If you need to get the result of each future as soon as possible, or if you need the result of some futures even if others produce errors, you can use WaitIterator:

```
wait_iterator = gen.WaitIterator(future1, future2)
while not wait_iterator.done():
    try:
        result = yield wait_iterator.next()
    except Exception as e:
        print("Error {} from {}".format(e, wait_iterator.current_future))
    else:
        print("Result {} received from {} at {}".format(
            result, wait_iterator.current_future,
            wait_iterator.current_index))
```

Because results are returned as soon as they are available the output from the iterator *will not be in the same order as the input arguments*. If you need to know which future produced the current result, you can use the attributes WaitIterator.current_future, or WaitIterator.current_index to get the index of the future from the input list. (if keyword arguments were used in the construction of the *WaitIterator*, current_index will use the corresponding keyword).

On Python 3.5, *WaitIterator* implements the async iterator protocol, so it can be used with the async for statement (note that in this version the entire iteration is aborted if any value raises an exception, while the previous example can continue past individual errors):

```
    async for result in gen.WaitIterator(future1, future2):
        print("Result {} received from {} at {}".format(
            result, wait_iterator.current_future,
            wait_iterator.current_index))
```

New in version 4.1.

Changed in version 4.3: Added `async for` support in Python 3.5.

done()
Returns True if this iterator has no more results.

next()
Returns a *Future* that will yield the next available result.

Note that this *Future* will not be the same object as any of the inputs.

tornado.gen.**multi**(*children*, *quiet_exceptions=()*)
Runs multiple asynchronous operations in parallel.

`children` may either be a list or a dict whose values are yieldable objects. `multi()` returns a new yieldable object that resolves to a parallel structure containing their results. If `children` is a list, the result is a list of results in the same order; if it is a dict, the result is a dict with the same keys.

That is, `results = yield multi(list_of_futures)` is equivalent to:

```
    results = []
    for future in list_of_futures:
        results.append(yield future)
```

If any children raise exceptions, `multi()` will raise the first one. All others will be logged, unless they are of types contained in the `quiet_exceptions` argument.

If any of the inputs are *YieldPoints*, the returned yieldable object is a *YieldPoint*. Otherwise, returns a *Future*. This means that the result of *multi* can be used in a native coroutine if and only if all of its children can be.

In a `yield`-based coroutine, it is not normally necessary to call this function directly, since the coroutine runner will do it automatically when a list or dict is yielded. However, it is necessary in `await`-based coroutines, or to pass the `quiet_exceptions` argument.

This function is available under the names `multi()` and `Multi()` for historical reasons.

Changed in version 4.2: If multiple yieldables fail, any exceptions after the first (which is raised) will be logged. Added the `quiet_exceptions` argument to suppress this logging for selected exception types.

Changed in version 4.3: Replaced the class `Multi` and the function `multi_future` with a unified function `multi`. Added support for yieldables other than *YieldPoint* and *Future*.

tornado.gen.**multi_future**(*children*, *quiet_exceptions=()*)
Wait for multiple asynchronous futures in parallel.

This function is similar to *multi*, but does not support *YieldPoints*.

New in version 4.0.

Changed in version 4.2: If multiple `Futures` fail, any exceptions after the first (which is raised) will be logged. Added the `quiet_exceptions` argument to suppress this logging for selected exception types.

Deprecated since version 4.3: Use *multi* instead.

tornado.gen.**Task**(*func*, **args*, ***kwargs*)
Adapts a callback-based asynchronous function for use in coroutines.

Takes a function (and optional additional arguments) and runs it with those arguments plus a `callback` keyword argument. The argument passed to the callback is returned as the result of the yield expression.

Changed in version 4.0: `gen.Task` is now a function that returns a *Future*, instead of a subclass of *YieldPoint*. It still behaves the same way when yielded.

class tornado.gen.**Arguments**
The result of a *Task* or *Wait* whose callback had more than one argument (or keyword arguments).

The *Arguments* object is a `collections.namedtuple` and can be used either as a tuple `(args, kwargs)` or an object with attributes `args` and `kwargs`.

tornado.gen.**convert_yielded**(*yielded*)
Convert a yielded object into a *Future*.

The default implementation accepts lists, dictionaries, and Futures.

If the `singledispatch` library is available, this function may be extended to support additional types. For example:

```
@convert_yielded.register(asyncio.Future)
def _(asyncio_future):
    return tornado.platform.asyncio.to_tornado_future(asyncio_future)
```

New in version 4.1.

tornado.gen.**maybe_future**(*x*)
Converts `x` into a *Future*.

If `x` is already a *Future*, it is simply returned; otherwise it is wrapped in a new *Future*. This is suitable for use as `result = yield gen.maybe_future(f())` when you don't know whether `f()` returns a *Future* or not.

Deprecated since version 4.3: This function only handles `Futures`, not other yieldable objects. Instead of *maybe_future*, check for the non-future result types you expect (often just `None`), and `yield` anything unknown.

Legacy interface

Before support for *Futures* was introduced in Tornado 3.0, coroutines used subclasses of *YieldPoint* in their `yield` expressions. These classes are still supported but should generally not be used except for compatibility with older interfaces. None of these classes are compatible with native (`await`-based) coroutines.

class tornado.gen.**YieldPoint**
Base class for objects that may be yielded from the generator.

Deprecated since version 4.0: Use *Futures* instead.

start(*runner*)
Called by the runner after the generator has yielded.

No other methods will be called on this object before `start`.

is_ready()
Called by the runner to determine whether to resume the generator.

Returns a boolean; may be called more than once.

get_result()
Returns the value to use as the result of the yield expression.

This method will only be called once, and only after *is_ready* has returned true.

class tornado.gen.**Callback**(*key*)
Returns a callable object that will allow a matching *Wait* to proceed.

The key may be any value suitable for use as a dictionary key, and is used to match Callbacks to their corresponding Waits. The key must be unique among outstanding callbacks within a single run of the generator function, but may be reused across different runs of the same function (so constants generally work fine).

The callback may be called with zero or one arguments; if an argument is given it will be returned by *Wait*.

Deprecated since version 4.0: Use *Futures* instead.

class tornado.gen.**Wait**(*key*)
Returns the argument passed to the result of a previous *Callback*.

Deprecated since version 4.0: Use *Futures* instead.

class tornado.gen.**WaitAll**(*keys*)
Returns the results of multiple previous *Callbacks*.

The argument is a sequence of *Callback* keys, and the result is a list of results in the same order.

WaitAll is equivalent to yielding a list of *Wait* objects.

Deprecated since version 4.0: Use *Futures* instead.

class tornado.gen.**MultiYieldPoint**(*children*, *quiet_exceptions=()*)
Runs multiple asynchronous operations in parallel.

This class is similar to *multi*, but it always creates a stack context even when no children require it. It is not compatible with native coroutines.

Changed in version 4.2: If multiple YieldPoints fail, any exceptions after the first (which is raised) will be logged. Added the quiet_exceptions argument to suppress this logging for selected exception types.

Changed in version 4.3: Renamed from Multi to MultiYieldPoint. The name Multi remains as an alias for the equivalent *multi* function.

Deprecated since version 4.3: Use *multi* instead.

4.5.2 tornado.concurrent — Work with threads and futures

Utilities for working with threads and Futures.

Futures are a pattern for concurrent programming introduced in Python 3.2 in the concurrent.futures package. This package defines a mostly-compatible *Future* class designed for use from coroutines, as well as some utility functions for interacting with the concurrent.futures package.

class tornado.concurrent.**Future**
Placeholder for an asynchronous result.

A Future encapsulates the result of an asynchronous operation. In synchronous applications Futures are used to wait for the result from a thread or process pool; in Tornado they are normally used with *IOLoop.add_future* or by yielding them in a *gen.coroutine*.

tornado.concurrent.Future is similar to concurrent.futures.Future, but not thread-safe (and therefore faster for use with single-threaded event loops).

In addition to exception and set_exception, methods exc_info and set_exc_info are supported to capture tracebacks in Python 2. The traceback is automatically available in Python 3, but in the Python 2 futures backport this information is discarded. This functionality was previously available in a separate class TracebackFuture, which is now a deprecated alias for this class.

Changed in version 4.0: *tornado.concurrent.Future* is always a thread-unsafe `Future` with support for the `exc_info` methods. Previously it would be an alias for the thread-safe `concurrent.futures.Future` if that package was available and fall back to the thread-unsafe implementation if it was not.

Changed in version 4.1: If a *Future* contains an error but that error is never observed (by calling `result()`, `exception()`, or `exc_info()`), a stack trace will be logged when the *Future* is garbage collected. This normally indicates an error in the application, but in cases where it results in undesired logging it may be necessary to suppress the logging by ensuring that the exception is observed: `f.add_done_callback(lambda f: f.exception())`.

Consumer methods

`Future.`**`result`**(*timeout=None*)
If the operation succeeded, return its result. If it failed, re-raise its exception.

This method takes a `timeout` argument for compatibility with `concurrent.futures.Future` but it is an error to call it before the *Future* is done, so the `timeout` is never used.

`Future.`**`exception`**(*timeout=None*)
If the operation raised an exception, return the `Exception` object. Otherwise returns None.

This method takes a `timeout` argument for compatibility with `concurrent.futures.Future` but it is an error to call it before the *Future* is done, so the `timeout` is never used.

`Future.`**`exc_info`**()
Returns a tuple in the same format as `sys.exc_info` or None.

New in version 4.0.

`Future.`**`add_done_callback`**(*fn*)
Attaches the given callback to the *Future*.

It will be invoked with the *Future* as its argument when the Future has finished running and its result is available. In Tornado consider using *IOLoop.add_future* instead of calling *add_done_callback* directly.

`Future.`**`done`**()
Returns True if the future has finished running.

`Future.`**`running`**()
Returns True if this operation is currently running.

`Future.`**`cancel`**()
Cancel the operation, if possible.

Tornado `Futures` do not support cancellation, so this method always returns False.

`Future.`**`cancelled`**()
Returns True if the operation has been cancelled.

Tornado `Futures` do not support cancellation, so this method always returns False.

Producer methods

`Future.`**`set_result`**(*result*)
Sets the result of a `Future`.

It is undefined to call any of the `set` methods more than once on the same object.

`Future.`**`set_exception`**(*exception*)
Sets the exception of a `Future.`

Future.**set_exc_info**(*exc_info*)
Sets the exception information of a Future.

Preserves tracebacks on Python 2.

New in version 4.0.

tornado.concurrent.**run_on_executor**(**args*, ***kwargs*)
Decorator to run a synchronous method asynchronously on an executor.

The decorated method may be called with a callback keyword argument and returns a future.

The *IOLoop* and executor to be used are determined by the io_loop and executor attributes of self. To use different attributes, pass keyword arguments to the decorator:

```
@run_on_executor(executor='_thread_pool')
def foo(self):
    pass
```

Changed in version 4.2: Added keyword arguments to use alternative attributes.

tornado.concurrent.**return_future**(*f*)
Decorator to make a function that returns via callback return a *Future*.

The wrapped function should take a callback keyword argument and invoke it with one argument when it has finished. To signal failure, the function can simply raise an exception (which will be captured by the *StackContext* and passed along to the Future).

From the caller's perspective, the callback argument is optional. If one is given, it will be invoked when the function is complete with *Future.result()* as an argument. If the function fails, the callback will not be run and an exception will be raised into the surrounding *StackContext*.

If no callback is given, the caller should use the Future to wait for the function to complete (perhaps by yielding it in a *gen.engine* function, or passing it to *IOLoop.add_future*).

Usage:

```
@return_future
def future_func(arg1, arg2, callback):
    # Do stuff (possibly asynchronous)
    callback(result)

@gen.engine
def caller(callback):
    yield future_func(arg1, arg2)
    callback()
```

Note that @return_future and @gen.engine can be applied to the same function, provided @return_future appears first. However, consider using @gen.coroutine instead of this combination.

tornado.concurrent.**chain_future**(*a*, *b*)
Chain two futures together so that when one completes, so does the other.

The result (success or failure) of a will be copied to b, unless b has already been completed or cancelled by the time a finishes.

4.5.3 tornado.locks – Synchronization primitives

New in version 4.2.

Coordinate coroutines with synchronization primitives analogous to those the standard library provides to threads.

(Note that these primitives are not actually thread-safe and cannot be used in place of those from the standard library–they are meant to coordinate Tornado coroutines in a single-threaded app, not to protect shared objects in a multi-threaded app.)

Condition

class `tornado.locks.`**`Condition`**

A condition allows one or more coroutines to wait until notified.

Like a standard `threading.Condition`, but does not need an underlying lock that is acquired and released.

With a *`Condition`*, coroutines can wait to be notified by other coroutines:

```
from tornado import gen
from tornado.ioloop import IOLoop
from tornado.locks import Condition

condition = Condition()

@gen.coroutine
def waiter():
    print("I'll wait right here")
    yield condition.wait()  # Yield a Future.
    print("I'm done waiting")

@gen.coroutine
def notifier():
    print("About to notify")
    condition.notify()
    print("Done notifying")

@gen.coroutine
def runner():
    # Yield two Futures; wait for waiter() and notifier() to finish.
    yield [waiter(), notifier()]

IOLoop.current().run_sync(runner)
```

```
I'll wait right here
About to notify
Done notifying
I'm done waiting
```

`wait` takes an optional `timeout` argument, which is either an absolute timestamp:

```
io_loop = IOLoop.current()

# Wait up to 1 second for a notification.
yield condition.wait(timeout=io_loop.time() + 1)
```

...or a `datetime.timedelta` for a timeout relative to the current time:

```
# Wait up to 1 second.
yield condition.wait(timeout=datetime.timedelta(seconds=1))
```

The method raises *`tornado.gen.TimeoutError`* if there's no notification before the deadline.

`wait`(*timeout=None*)

Wait for *`notify`*.

Returns a *Future* that resolves `True` if the condition is notified, or `False` after a timeout.

notify(*n=1*)
Wake `n` waiters.

notify_all()
Wake all waiters.

Event

class `tornado.locks.`**Event**
An event blocks coroutines until its internal flag is set to True.

Similar to `threading.Event`.

A coroutine can wait for an event to be set. Once it is set, calls to `yield event.wait()` will not block unless the event has been cleared:

```
from tornado import gen
from tornado.ioloop import IOLoop
from tornado.locks import Event

event = Event()

@gen.coroutine
def waiter():
    print("Waiting for event")
    yield event.wait()
    print("Not waiting this time")
    yield event.wait()
    print("Done")

@gen.coroutine
def setter():
    print("About to set the event")
    event.set()

@gen.coroutine
def runner():
    yield [waiter(), setter()]

IOLoop.current().run_sync(runner)
```

```
Waiting for event
About to set the event
Not waiting this time
Done
```

is_set()
Return `True` if the internal flag is true.

set()
Set the internal flag to `True`. All waiters are awakened.

Calling *wait* once the flag is set will not block.

clear()
Reset the internal flag to `False`.

Calls to *wait* will block until *set* is called.

wait(*timeout=None*)
Block until the internal flag is true.

Returns a Future, which raises *tornado.gen.TimeoutError* after a timeout.

Semaphore

class tornado.locks.**Semaphore**(*value=1*)
A lock that can be acquired a fixed number of times before blocking.

A Semaphore manages a counter representing the number of *release* calls minus the number of *acquire* calls, plus an initial value. The *acquire* method blocks if necessary until it can return without making the counter negative.

Semaphores limit access to a shared resource. To allow access for two workers at a time:

```
from tornado import gen
from tornado.ioloop import IOLoop
from tornado.locks import Semaphore

sem = Semaphore(2)

@gen.coroutine
def worker(worker_id):
    yield sem.acquire()
    try:
        print("Worker %d is working" % worker_id)
        yield use_some_resource()
    finally:
        print("Worker %d is done" % worker_id)
        sem.release()

@gen.coroutine
def runner():
    # Join all workers.
    yield [worker(i) for i in range(3)]

IOLoop.current().run_sync(runner)
```

```
Worker 0 is working
Worker 1 is working
Worker 0 is done
Worker 2 is working
Worker 1 is done
Worker 2 is done
```

Workers 0 and 1 are allowed to run concurrently, but worker 2 waits until the semaphore has been released once, by worker 0.

acquire is a context manager, so `worker` could be written as:

```
@gen.coroutine
def worker(worker_id):
    with (yield sem.acquire()):
        print("Worker %d is working" % worker_id)
        yield use_some_resource()

    # Now the semaphore has been released.
    print("Worker %d is done" % worker_id)
```

In Python 3.5, the semaphore itself can be used as an async context manager:

```
async def worker(worker_id):
    async with sem:
        print("Worker %d is working" % worker_id)
        await use_some_resource()

    # Now the semaphore has been released.
    print("Worker %d is done" % worker_id)
```

Changed in version 4.3: Added `async with` support in Python 3.5.

release()
Increment the counter and wake one waiter.

acquire(*timeout=None*)
Decrement the counter. Returns a Future.

Block if the counter is zero and wait for a *release*. The Future raises *TimeoutError* after the deadline.

BoundedSemaphore

class tornado.locks.**BoundedSemaphore**(*value=1*)
A semaphore that prevents release() being called too many times.

If *release* would increment the semaphore's value past the initial value, it raises ValueError. Semaphores are mostly used to guard resources with limited capacity, so a semaphore released too many times is a sign of a bug.

release()
Increment the counter and wake one waiter.

acquire(*timeout=None*)
Decrement the counter. Returns a Future.

Block if the counter is zero and wait for a *release*. The Future raises *TimeoutError* after the deadline.

Lock

class tornado.locks.**Lock**
A lock for coroutines.

A Lock begins unlocked, and *acquire* locks it immediately. While it is locked, a coroutine that yields *acquire* waits until another coroutine calls *release*.

Releasing an unlocked lock raises RuntimeError.

acquire supports the context manager protocol in all Python versions:

```
>>> from tornado import gen, locks
>>> lock = locks.Lock()
>>>
>>> @gen.coroutine
... def f():
...    with (yield lock.acquire()):
...        # Do something holding the lock.
...        pass
```

```
...
...     # Now the lock is released.
```

In Python 3.5, *Lock* also supports the async context manager protocol. Note that in this case there is no *acquire*, because `async with` includes both the `yield` and the `acquire` (just as it does with `threading.Lock`):

```
>>> async def f():
...     async with lock:
...         # Do something holding the lock.
...         pass
...
...     # Now the lock is released.
```

Changed in version 4.3: Added `async with` support in Python 3.5.

acquire(*timeout=None*)
Attempt to lock. Returns a Future.

Returns a Future, which raises *tornado.gen.TimeoutError* after a timeout.

release()
Unlock.

The first coroutine in line waiting for *acquire* gets the lock.

If not locked, raise a `RuntimeError`.

4.5.4 tornado.queues – Queues for coroutines

New in version 4.2.

Classes

Queue

class tornado.queues.**Queue**(*maxsize=0*)
Coordinate producer and consumer coroutines.

If maxsize is 0 (the default) the queue size is unbounded.

```
from tornado import gen
from tornado.ioloop import IOLoop
from tornado.queues import Queue

q = Queue(maxsize=2)

@gen.coroutine
def consumer():
    while True:
        item = yield q.get()
        try:
            print('Doing work on %s' % item)
            yield gen.sleep(0.01)
        finally:
            q.task_done()

@gen.coroutine
```

```
    def producer():
        for item in range(5):
            yield q.put(item)
            print('Put %s' % item)

    @gen.coroutine
    def main():
        # Start consumer without waiting (since it never finishes).
        IOLoop.current().spawn_callback(consumer)
        yield producer()     # Wait for producer to put all tasks.
        yield q.join()       # Wait for consumer to finish all tasks.
        print('Done')

    IOLoop.current().run_sync(main)
```

```
    Put 0
    Put 1
    Doing work on 0
    Put 2
    Doing work on 1
    Put 3
    Doing work on 2
    Put 4
    Doing work on 3
    Doing work on 4
    Done
```

In Python 3.5, *Queue* implements the async iterator protocol, so `consumer()` could be rewritten as:

```
    async def consumer():
        async for item in q:
            try:
                print('Doing work on %s' % item)
                yield gen.sleep(0.01)
            finally:
                q.task_done()
```

Changed in version 4.3: Added `async for` support in Python 3.5.

maxsize
Number of items allowed in the queue.

qsize()
Number of items in the queue.

put(*item*, *timeout=None*)
Put an item into the queue, perhaps waiting until there is room.

Returns a Future, which raises *tornado.gen.TimeoutError* after a timeout.

put_nowait(*item*)
Put an item into the queue without blocking.

If no free slot is immediately available, raise *QueueFull*.

get(*timeout=None*)
Remove and return an item from the queue.

Returns a Future which resolves once an item is available, or raises *tornado.gen.TimeoutError* after a timeout.

get_nowait()
Remove and return an item from the queue without blocking.

Return an item if one is immediately available, else raise *QueueEmpty*.

task_done()
Indicate that a formerly enqueued task is complete.

Used by queue consumers. For each *get* used to fetch a task, a subsequent call to *task_done* tells the queue that the processing on the task is complete.

If a *join* is blocking, it resumes when all items have been processed; that is, when every *put* is matched by a *task_done*.

Raises ValueError if called more times than *put*.

join(*timeout=None*)
Block until all items in the queue are processed.

Returns a Future, which raises *tornado.gen.TimeoutError* after a timeout.

PriorityQueue

class tornado.queues.**PriorityQueue**(*maxsize=0*)
A *Queue* that retrieves entries in priority order, lowest first.

Entries are typically tuples like (priority number, data).

```
from tornado.queues import PriorityQueue

q = PriorityQueue()
q.put((1, 'medium-priority item'))
q.put((0, 'high-priority item'))
q.put((10, 'low-priority item'))

print(q.get_nowait())
print(q.get_nowait())
print(q.get_nowait())
```

```
(0, 'high-priority item')
(1, 'medium-priority item')
(10, 'low-priority item')
```

LifoQueue

class tornado.queues.**LifoQueue**(*maxsize=0*)
A *Queue* that retrieves the most recently put items first.

```
from tornado.queues import LifoQueue

q = LifoQueue()
q.put(3)
q.put(2)
q.put(1)

print(q.get_nowait())
print(q.get_nowait())
print(q.get_nowait())
```

```
1
2
3
```

Exceptions

QueueEmpty

exception tornado.queues.**QueueEmpty**
Raised by *Queue.get_nowait* when the queue has no items.

QueueFull

exception tornado.queues.**QueueFull**
Raised by *Queue.put_nowait* when a queue is at its maximum size.

4.5.5 tornado.process — Utilities for multiple processes

Utilities for working with multiple processes, including both forking the server into multiple processes and managing subprocesses.

exception tornado.process.**CalledProcessError**
An alias for subprocess.CalledProcessError.

tornado.process.**cpu_count**()
Returns the number of processors on this machine.

tornado.process.**fork_processes**(*num_processes*, *max_restarts=100*)
Starts multiple worker processes.

If num_processes is None or <= 0, we detect the number of cores available on this machine and fork that number of child processes. If num_processes is given and > 0, we fork that specific number of sub-processes.

Since we use processes and not threads, there is no shared memory between any server code.

Note that multiple processes are not compatible with the autoreload module (or the autoreload=True option to *tornado.web.Application* which defaults to True when debug=True). When using multiple processes, no IOLoops can be created or referenced until after the call to fork_processes.

In each child process, fork_processes returns its *task id*, a number between 0 and num_processes. Processes that exit abnormally (due to a signal or non-zero exit status) are restarted with the same id (up to max_restarts times). In the parent process, fork_processes returns None if all child processes have exited normally, but will otherwise only exit by throwing an exception.

tornado.process.**task_id**()
Returns the current task id, if any.

Returns None if this process was not created by *fork_processes*.

class tornado.process.**Subprocess**(**args*, ***kwargs*)
Wraps subprocess.Popen with IOStream support.

The constructor is the same as subprocess.Popen with the following additions:

- stdin, stdout, and stderr may have the value tornado.process.Subprocess.STREAM, which will make the corresponding attribute of the resulting Subprocess a *PipeIOStream*.

•A new keyword argument `io_loop` may be used to pass in an IOLoop.

The `Subprocess.STREAM` option and the `set_exit_callback` and `wait_for_exit` methods do not work on Windows. There is therefore no reason to use this class instead of `subprocess.Popen` on that platform.

Changed in version 4.1: The `io_loop` argument is deprecated.

set_exit_callback(*callback*)
Runs `callback` when this process exits.

The callback takes one argument, the return code of the process.

This method uses a `SIGCHLD` handler, which is a global setting and may conflict if you have other libraries trying to handle the same signal. If you are using more than one `IOLoop` it may be necessary to call *Subprocess.initialize* first to designate one `IOLoop` to run the signal handlers.

In many cases a close callback on the stdout or stderr streams can be used as an alternative to an exit callback if the signal handler is causing a problem.

wait_for_exit(*raise_error=True*)
Returns a *Future* which resolves when the process exits.

Usage:

```
ret = yield proc.wait_for_exit()
```

This is a coroutine-friendly alternative to *set_exit_callback* (and a replacement for the blocking `subprocess.Popen.wait`).

By default, raises `subprocess.CalledProcessError` if the process has a non-zero exit status. Use `wait_for_exit(raise_error=False)` to suppress this behavior and return the exit status without raising.

New in version 4.2.

classmethod initialize(*io_loop=None*)
Initializes the `SIGCHLD` handler.

The signal handler is run on an *IOLoop* to avoid locking issues. Note that the *IOLoop* used for signal handling need not be the same one used by individual Subprocess objects (as long as the `IOLoops` are each running in separate threads).

Changed in version 4.1: The `io_loop` argument is deprecated.

classmethod uninitialize()
Removes the `SIGCHLD` handler.

4.6 Integration with other services

4.6.1 tornado.auth — Third-party login with OpenID and OAuth

This module contains implementations of various third-party authentication schemes.

All the classes in this file are class mixins designed to be used with the *tornado.web.RequestHandler* class. They are used in two ways:

- On a login handler, use methods such as `authenticate_redirect()`, `authorize_redirect()`, and `get_authenticated_user()` to establish the user's identity and store authentication tokens to your database and/or cookies.

- In non-login handlers, use methods such as `facebook_request()` or `twitter_request()` to use the authentication tokens to make requests to the respective services.

They all take slightly different arguments due to the fact all these services implement authentication and authorization slightly differently. See the individual service classes below for complete documentation.

Example usage for Google OAuth:

```
class GoogleOAuth2LoginHandler(tornado.web.RequestHandler,
                               tornado.auth.GoogleOAuth2Mixin):
    @tornado.gen.coroutine
    def get(self):
        if self.get_argument('code', False):
            user = yield self.get_authenticated_user(
                redirect_uri='http://your.site.com/auth/google',
                code=self.get_argument('code'))
            # Save the user with e.g. set_secure_cookie
        else:
            yield self.authorize_redirect(
                redirect_uri='http://your.site.com/auth/google',
                client_id=self.settings['google_oauth']['key'],
                scope=['profile', 'email'],
                response_type='code',
                extra_params={'approval_prompt': 'auto'})
```

Changed in version 4.0: All of the callback interfaces in this module are now guaranteed to run their callback with an argument of `None` on error. Previously some functions would do this while others would simply terminate the request on their own. This change also ensures that errors are more consistently reported through the `Future` interfaces.

Common protocols

These classes implement the OpenID and OAuth standards. They will generally need to be subclassed to use them with any particular site. The degree of customization required will vary, but in most cases overridding the class attributes (which are named beginning with underscores for historical reasons) should be sufficient.

class `tornado.auth.`**`OpenIdMixin`**

Abstract implementation of OpenID and Attribute Exchange.

Class attributes:

- `_OPENID_ENDPOINT`: the identity provider's URI.

`authenticate_redirect`(*callback_uri=None, ax_attrs=['name', 'email', 'language', 'username'], callback=None*)

Redirects to the authentication URL for this service.

After authentication, the service will redirect back to the given callback URI with additional parameters including `openid.mode`.

We request the given attributes for the authenticated user by default (name, email, language, and username). If you don't need all those attributes for your app, you can request fewer with the ax_attrs keyword argument.

Changed in version 3.1: Returns a *`Future`* and takes an optional callback. These are not strictly necessary as this method is synchronous, but they are supplied for consistency with *`OAuthMixin.authorize_redirect`*.

`get_authenticated_user`(*callback*, *http_client=None*)

Fetches the authenticated user data upon redirect.

This method should be called by the handler that receives the redirect from the *authenticate_redirect()* method (which is often the same as the one that calls it; in that case you would call *get_authenticated_user* if the `openid.mode` parameter is present and *authenticate_redirect* if it is not).

The result of this method will generally be used to set a cookie.

get_auth_http_client()
Returns the *AsyncHTTPClient* instance to be used for auth requests.

May be overridden by subclasses to use an HTTP client other than the default.

class tornado.auth.**OAuthMixin**
Abstract implementation of OAuth 1.0 and 1.0a.

See *TwitterMixin* below for an example implementation.

Class attributes:

- `_OAUTH_AUTHORIZE_URL`: The service's OAuth authorization url.
- `_OAUTH_ACCESS_TOKEN_URL`: The service's OAuth access token url.
- `_OAUTH_VERSION`: May be either "1.0" or "1.0a".
- `_OAUTH_NO_CALLBACKS`: Set this to True if the service requires advance registration of callbacks.

Subclasses must also override the *_oauth_get_user_future* and *_oauth_consumer_token* methods.

authorize_redirect(*callback_uri=None*, *extra_params=None*, *http_client=None*, *callback=None*)
Redirects the user to obtain OAuth authorization for this service.

The `callback_uri` may be omitted if you have previously registered a callback URI with the third-party service. For some services (including Friendfeed), you must use a previously-registered callback URI and cannot specify a callback via this method.

This method sets a cookie called `_oauth_request_token` which is subsequently used (and cleared) in *get_authenticated_user* for security purposes.

Note that this method is asynchronous, although it calls *RequestHandler.finish* for you so it may not be necessary to pass a callback or use the *Future* it returns. However, if this method is called from a function decorated with *gen.coroutine*, you must call it with `yield` to keep the response from being closed prematurely.

Changed in version 3.1: Now returns a *Future* and takes an optional callback, for compatibility with *gen.coroutine*.

get_authenticated_user(*callback*, *http_client=None*)
Gets the OAuth authorized user and access token.

This method should be called from the handler for your OAuth callback URL to complete the registration process. We run the callback with the authenticated user dictionary. This dictionary will contain an `access_key` which can be used to make authorized requests to this service on behalf of the user. The dictionary will also contain other fields such as `name`, depending on the service used.

_oauth_consumer_token()
Subclasses must override this to return their OAuth consumer keys.

The return value should be a `dict` with keys `key` and `secret`.

_oauth_get_user_future(*access_token*, *callback*)
Subclasses must override this to get basic information about the user.

Should return a *Future* whose result is a dictionary containing information about the user, which may have been retrieved by using `access_token` to make a request to the service.

The access token will be added to the returned dictionary to make the result of *get_authenticated_user*.

For backwards compatibility, the callback-based `_oauth_get_user` method is also supported.

get_auth_http_client()
Returns the *AsyncHTTPClient* instance to be used for auth requests.

May be overridden by subclasses to use an HTTP client other than the default.

class `tornado.auth.`**OAuth2Mixin**
Abstract implementation of OAuth 2.0.

See *FacebookGraphMixin* or *GoogleOAuth2Mixin* below for example implementations.

Class attributes:

- `_OAUTH_AUTHORIZE_URL`: The service's authorization url.
- `_OAUTH_ACCESS_TOKEN_URL`: The service's access token url.

authorize_redirect(*redirect_uri=None*, *client_id=None*, *client_secret=None*, *extra_params=None*, *callback=None*, *scope=None*, *response_type='code'*)
Redirects the user to obtain OAuth authorization for this service.

Some providers require that you register a redirect URL with your application instead of passing one via this method. You should call this method to log the user in, and then call `get_authenticated_user` in the handler for your redirect URL to complete the authorization process.

Changed in version 3.1: Returns a *Future* and takes an optional callback. These are not strictly necessary as this method is synchronous, but they are supplied for consistency with *OAuthMixin.authorize_redirect*.

oauth2_request(*url*, *callback*, *access_token=None*, *post_args=None*, ***args*)
Fetches the given URL auth an OAuth2 access token.

If the request is a POST, `post_args` should be provided. Query string arguments should be given as keyword arguments.

Example usage:

..testcode:

```
class MainHandler(tornado.web.RequestHandler,
                  tornado.auth.FacebookGraphMixin):
    @tornado.web.authenticated
    @tornado.gen.coroutine
    def get(self):
        new_entry = yield self.oauth2_request(
            "https://graph.facebook.com/me/feed",
            post_args={"message": "I am posting from my Tornado application!"},
            access_token=self.current_user["access_token"])

        if not new_entry:
            # Call failed; perhaps missing permission?
            yield self.authorize_redirect()
            return
        self.finish("Posted a message!")
```

New in version 4.3.

get_auth_http_client()
Returns the *AsyncHTTPClient* instance to be used for auth requests.

May be overridden by subclasses to use an HTTP client other than the default.

New in version 4.3.

Google

class tornado.auth.**GoogleOAuth2Mixin**
Google authentication using OAuth2.

In order to use, register your application with Google and copy the relevant parameters to your application settings.

- Go to the Google Dev Console at http://console.developers.google.com
- Select a project, or create a new one.
- In the sidebar on the left, select APIs & Auth.
- In the list of APIs, find the Google+ API service and set it to ON.
- In the sidebar on the left, select Credentials.
- In the OAuth section of the page, select Create New Client ID.
- Set the Redirect URI to point to your auth handler
- Copy the "Client secret" and "Client ID" to the application settings as {"google_oauth": {"key": CLIENT_ID, "secret": CLIENT_SECRET}}

New in version 3.2.

get_authenticated_user(*redirect_uri*, *code*, *callback*)
Handles the login for the Google user, returning an access token.

The result is a dictionary containing an `access_token` field ([among others](https://developers.google.com/identity/protocols/OAuth2WebServer#handlingtheresponse)). Unlike other `get_authenticated_user` methods in this package, this method does not return any additional information about the user. The returned access token can be used with *OAuth2Mixin.oauth2_request* to request additional information (perhaps from `https://www.googleapis.com/oauth2/v2/userinfo`)

Example usage:

```
class GoogleOAuth2LoginHandler(tornado.web.RequestHandler,
                               tornado.auth.GoogleOAuth2Mixin):
    @tornado.gen.coroutine
    def get(self):
        if self.get_argument('code', False):
            access = yield self.get_authenticated_user(
                redirect_uri='http://your.site.com/auth/google',
                code=self.get_argument('code'))
            user = yield self.oauth2_request(
                "https://www.googleapis.com/oauth2/v1/userinfo",
                access_token=access["access_token"])
            # Save the user and access token with
            # e.g. set_secure_cookie.
        else:
            yield self.authorize_redirect(
                redirect_uri='http://your.site.com/auth/google',
```

```
                        client_id=self.settings['google_oauth']['key'],
                        scope=['profile', 'email'],
                        response_type='code',
                        extra_params={'approval_prompt': 'auto'})
```

Facebook

class tornado.auth.**FacebookGraphMixin**

Facebook authentication using the new Graph API and OAuth2.

get_authenticated_user(*redirect_uri*, *client_id*, *client_secret*, *code*, *callback*, *extra_fields=None*)

Handles the login for the Facebook user, returning a user object.

Example usage:

```
class FacebookGraphLoginHandler(tornado.web.RequestHandler,
                                tornado.auth.FacebookGraphMixin):
  @tornado.gen.coroutine
  def get(self):
      if self.get_argument("code", False):
          user = yield self.get_authenticated_user(
              redirect_uri='/auth/facebookgraph/',
              client_id=self.settings["facebook_api_key"],
              client_secret=self.settings["facebook_secret"],
              code=self.get_argument("code"))
          # Save the user with e.g. set_secure_cookie
      else:
          yield self.authorize_redirect(
              redirect_uri='/auth/facebookgraph/',
              client_id=self.settings["facebook_api_key"],
              extra_params={"scope": "read_stream,offline_access"})
```

facebook_request(*path*, *callback*, *access_token=None*, *post_args=None*, ***args*)

Fetches the given relative API path, e.g., "/btaylor/picture"

If the request is a POST, `post_args` should be provided. Query string arguments should be given as keyword arguments.

An introduction to the Facebook Graph API can be found at http://developers.facebook.com/docs/api

Many methods require an OAuth access token which you can obtain through *authorize_redirect* and *get_authenticated_user*. The user returned through that process includes an `access_token` attribute that can be used to make authenticated requests via this method.

Example usage:

..testcode:

```
class MainHandler(tornado.web.RequestHandler,
                  tornado.auth.FacebookGraphMixin):
    @tornado.web.authenticated
    @tornado.gen.coroutine
    def get(self):
        new_entry = yield self.facebook_request(
            "/me/feed",
            post_args={"message": "I am posting from my Tornado application!"},
            access_token=self.current_user["access_token"])
```

```
                if not new_entry:
                    # Call failed; perhaps missing permission?
                    yield self.authorize_redirect()
                    return
                self.finish("Posted a message!")
```

The given path is relative to `self._FACEBOOK_BASE_URL`, by default "https://graph.facebook.com".

This method is a wrapper around *OAuth2Mixin.oauth2_request*; the only difference is that this method takes a relative path, while `oauth2_request` takes a complete url.

Changed in version 3.1: Added the ability to override `self._FACEBOOK_BASE_URL`.

Twitter

class `tornado.auth.`**TwitterMixin**

Twitter OAuth authentication.

To authenticate with Twitter, register your application with Twitter at http://twitter.com/apps. Then copy your Consumer Key and Consumer Secret to the application *settings* `twitter_consumer_key` and `twitter_consumer_secret`. Use this mixin on the handler for the URL you registered as your application's callback URL.

When your application is set up, you can use this mixin like this to authenticate the user with Twitter and get access to their stream:

```
class TwitterLoginHandler(tornado.web.RequestHandler,
                          tornado.auth.TwitterMixin):
    @tornado.gen.coroutine
    def get(self):
        if self.get_argument("oauth_token", None):
            user = yield self.get_authenticated_user()
            # Save the user using e.g. set_secure_cookie()
        else:
            yield self.authorize_redirect()
```

The user object returned by *get_authenticated_user* includes the attributes `username`, `name`, `access_token`, and all of the custom Twitter user attributes described at https://dev.twitter.com/docs/api/1.1/get/users/show

authenticate_redirect(*callback_uri=None*, *callback=None*)

Just like *authorize_redirect*, but auto-redirects if authorized.

This is generally the right interface to use if you are using Twitter for single-sign on.

Changed in version 3.1: Now returns a *Future* and takes an optional callback, for compatibility with *gen.coroutine*.

twitter_request(*path*, *callback=None*, *access_token=None*, *post_args=None*, ***args*)

Fetches the given API path, e.g., `statuses/user_timeline/btaylor`

The path should not include the format or API version number. (we automatically use JSON format and API version 1).

If the request is a POST, `post_args` should be provided. Query string arguments should be given as keyword arguments.

All the Twitter methods are documented at http://dev.twitter.com/

Many methods require an OAuth access token which you can obtain through *authorize_redirect* and *get_authenticated_user*. The user returned through that process includes an 'access_token' attribute that can be used to make authenticated requests via this method. Example usage:

```
class MainHandler(tornado.web.RequestHandler,
                  tornado.auth.TwitterMixin):
    @tornado.web.authenticated
    @tornado.gen.coroutine
    def get(self):
        new_entry = yield self.twitter_request(
            "/statuses/update",
            post_args={"status": "Testing Tornado Web Server"},
            access_token=self.current_user["access_token"])
        if not new_entry:
            # Call failed; perhaps missing permission?
            yield self.authorize_redirect()
            return
        self.finish("Posted a message!")
```

4.6.2 `tornado.wsgi` — Interoperability with other Python frameworks and servers

WSGI support for the Tornado web framework.

WSGI is the Python standard for web servers, and allows for interoperability between Tornado and other Python web frameworks and servers. This module provides WSGI support in two ways:

- *WSGIAdapter* converts a *tornado.web.Application* to the WSGI application interface. This is useful for running a Tornado app on another HTTP server, such as Google App Engine. See the *WSGIAdapter* class documentation for limitations that apply.
- *WSGIContainer* lets you run other WSGI applications and frameworks on the Tornado HTTP server. For example, with this class you can mix Django and Tornado handlers in a single server.

Running Tornado apps on WSGI servers

class tornado.wsgi.**WSGIAdapter**(*application*)

Converts a *tornado.web.Application* instance into a WSGI application.

Example usage:

```
import tornado.web
import tornado.wsgi
import wsgiref.simple_server

class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("Hello, world")

if __name__ == "__main__":
    application = tornado.web.Application([
        (r"/", MainHandler),
    ])
    wsgi_app = tornado.wsgi.WSGIAdapter(application)
    server = wsgiref.simple_server.make_server('', 8888, wsgi_app)
    server.serve_forever()
```

See the appengine demo for an example of using this module to run a Tornado app on Google App Engine.

In WSGI mode asynchronous methods are not supported. This means that it is not possible to use *AsyncHTTPClient*, or the *tornado.auth* or *tornado.websocket* modules.

New in version 4.0.

class tornado.wsgi.**WSGIApplication**(*handlers=None*, *default_host=''*, *transforms=None*, ***settings*)

A WSGI equivalent of *tornado.web.Application*.

Deprecated since version 4.0: Use a regular *Application* and wrap it in *WSGIAdapter* instead.

Running WSGI apps on Tornado servers

class tornado.wsgi.**WSGIContainer**(*wsgi_application*)

Makes a WSGI-compatible function runnable on Tornado's HTTP server.

> **Warning:** WSGI is a *synchronous* interface, while Tornado's concurrency model is based on single-threaded asynchronous execution. This means that running a WSGI app with Tornado's *WSGIContainer* is *less scalable* than running the same app in a multi-threaded WSGI server like `gunicorn` or `uwsgi`. Use *WSGIContainer* only when there are benefits to combining Tornado and WSGI in the same process that outweigh the reduced scalability.

Wrap a WSGI function in a *WSGIContainer* and pass it to *HTTPServer* to run it. For example:

```
def simple_app(environ, start_response):
    status = "200 OK"
    response_headers = [("Content-type", "text/plain")]
    start_response(status, response_headers)
    return ["Hello world!\n"]

container = tornado.wsgi.WSGIContainer(simple_app)
http_server = tornado.httpserver.HTTPServer(container)
http_server.listen(8888)
tornado.ioloop.IOLoop.current().start()
```

This class is intended to let other frameworks (Django, web.py, etc) run on the Tornado HTTP server and I/O loop.

The *tornado.web.FallbackHandler* class is often useful for mixing Tornado and WSGI apps in the same server. See https://github.com/bdarnell/django-tornado-demo for a complete example.

static environ(*request*)

Converts a *tornado.httputil.HTTPServerRequest* to a WSGI environment.

4.6.3 tornado.platform.asyncio — Bridge between asyncio and Tornado

Bridges between the `asyncio` module and Tornado IOLoop.

New in version 3.2.

This module integrates Tornado with the `asyncio` module introduced in Python 3.4 (and available as a separate download for Python 3.3). This makes it possible to combine the two libraries on the same event loop.

Most applications should use *AsyncIOMainLoop* to run Tornado on the default `asyncio` event loop. Applications that need to run event loops on multiple threads may use *AsyncIOLoop* to create multiple loops.

Note: Tornado requires the `add_reader` family of methods, so it is not compatible with the `ProactorEventLoop` on Windows. Use the `SelectorEventLoop` instead.

class `tornado.platform.asyncio.`**`AsyncIOMainLoop`**

`AsyncIOMainLoop` creates an *`IOLoop`* that corresponds to the current `asyncio` event loop (i.e. the one returned by `asyncio.get_event_loop()`). Recommended usage:

```
from tornado.platform.asyncio import AsyncIOMainLoop
import asyncio
AsyncIOMainLoop().install()
asyncio.get_event_loop().run_forever()
```

See also *`tornado.ioloop.IOLoop.install()`* for general notes on installing alternative IOLoops.

class `tornado.platform.asyncio.`**`AsyncIOLoop`**

`AsyncIOLoop` is an *`IOLoop`* that runs on an `asyncio` event loop. This class follows the usual Tornado semantics for creating new `IOLoops`; these loops are not necessarily related to the `asyncio` default event loop. Recommended usage:

```
from tornado.ioloop import IOLoop
IOLoop.configure('tornado.platform.asyncio.AsyncIOLoop')
IOLoop.current().start()
```

Each `AsyncIOLoop` creates a new `asyncio.EventLoop`; this object can be accessed with the `asyncio_loop` attribute.

`tornado.platform.asyncio.`**`to_tornado_future`**(*asyncio_future*)

Convert an `asyncio.Future` to a *`tornado.concurrent.Future`*.

New in version 4.1.

`tornado.platform.asyncio.`**`to_asyncio_future`**(*tornado_future*)

Convert a Tornado yieldable object to an `asyncio.Future`.

New in version 4.1.

Changed in version 4.3: Now accepts any yieldable object, not just *`tornado.concurrent.Future`*.

4.6.4 `tornado.platform.caresresolver` — Asynchronous DNS Resolver using C-Ares

This module contains a DNS resolver using the c-ares library (and its wrapper `pycares`).

class `tornado.platform.caresresolver.`**`CaresResolver`**

Name resolver based on the c-ares library.

This is a non-blocking and non-threaded resolver. It may not produce the same results as the system resolver, but can be used for non-blocking resolution when threads cannot be used.

c-ares fails to resolve some names when `family` is `AF_UNSPEC`, so it is only recommended for use in `AF_INET` (i.e. IPv4). This is the default for `tornado.simple_httpclient`, but other libraries may default to `AF_UNSPEC`.

4.6.5 `tornado.platform.twisted` — Bridges between Twisted and Tornado

Bridges between the Twisted reactor and Tornado IOLoop.

This module lets you run applications and libraries written for Twisted in a Tornado application. It can be used in two modes, depending on which library's underlying event loop you want to use.

This module has been tested with Twisted versions 11.0.0 and newer.

Twisted on Tornado

class tornado.platform.twisted.**TornadoReactor**(*io_loop=None*)
Twisted reactor built on the Tornado IOLoop.

TornadoReactor implements the Twisted reactor interface on top of the Tornado IOLoop. To use it, simply call *install* at the beginning of the application:

```
import tornado.platform.twisted
tornado.platform.twisted.install()
from twisted.internet import reactor
```

When the app is ready to start, call `IOLoop.current().start()` instead of `reactor.run()`.

It is also possible to create a non-global reactor by calling `tornado.platform.twisted.TornadoReactor(io_loop)`. However, if the *IOLoop* and reactor are to be short-lived (such as those used in unit tests), additional cleanup may be required. Specifically, it is recommended to call:

```
reactor.fireSystemEvent('shutdown')
reactor.disconnectAll()
```

before closing the *IOLoop*.

Changed in version 4.1: The `io_loop` argument is deprecated.

tornado.platform.twisted.**install**(*io_loop=None*)
Install this package as the default Twisted reactor.

`install()` must be called very early in the startup process, before most other twisted-related imports. Conversely, because it initializes the *IOLoop*, it cannot be called before *fork_processes* or multi-process *start*. These conflicting requirements make it difficult to use *TornadoReactor* in multi-process mode, and an external process manager such as `supervisord` is recommended instead.

Changed in version 4.1: The `io_loop` argument is deprecated.

Tornado on Twisted

class tornado.platform.twisted.**TwistedIOLoop**
IOLoop implementation that runs on Twisted.

TwistedIOLoop implements the Tornado IOLoop interface on top of the Twisted reactor. Recommended usage:

```
from tornado.platform.twisted import TwistedIOLoop
from twisted.internet import reactor
TwistedIOLoop().install()
# Set up your tornado application as usual using `IOLoop.instance`
reactor.run()
```

Uses the global Twisted reactor by default. To create multiple `TwistedIOLoops` in the same process, you must pass a unique reactor when constructing each one.

Not compatible with *tornado.process.Subprocess.set_exit_callback* because the `SIGCHLD` handlers used by Tornado and Twisted conflict with each other.

See also *tornado.ioloop.IOLoop.install()* for general notes on installing alternative IOLoops.

Twisted DNS resolver

class tornado.platform.twisted.**TwistedResolver**
Twisted-based asynchronous resolver.

This is a non-blocking and non-threaded resolver. It is recommended only when threads cannot be used, since it has limitations compared to the standard `getaddrinfo`-based *Resolver* and *ThreadedResolver*. Specifically, it returns at most one result, and arguments other than `host` and `family` are ignored. It may fail to resolve when `family` is not `socket.AF_UNSPEC`.

Requires Twisted 12.1 or newer.

Changed in version 4.1: The `io_loop` argument is deprecated.

4.7 Utilities

4.7.1 tornado.autoreload — Automatically detect code changes in development

Automatically restart the server when a source file is modified.

Most applications should not access this module directly. Instead, pass the keyword argument `autoreload=True` to the *tornado.web.Application* constructor (or `debug=True`, which enables this setting and several others). This will enable autoreload mode as well as checking for changes to templates and static resources. Note that restarting is a destructive operation and any requests in progress will be aborted when the process restarts. (If you want to disable autoreload while using other debug-mode features, pass both `debug=True` and `autoreload=False`).

This module can also be used as a command-line wrapper around scripts such as unit test runners. See the *main* method for details.

The command-line wrapper and Application debug modes can be used together. This combination is encouraged as the wrapper catches syntax errors and other import-time failures, while debug mode catches changes once the server has started.

This module depends on *IOLoop*, so it will not work in WSGI applications and Google App Engine. It also will not work correctly when *HTTPServer*'s multi-process mode is used.

Reloading loses any Python interpreter command-line arguments (e.g. `-u`) because it re-executes Python using `sys.executable` and `sys.argv`. Additionally, modifying these variables will cause reloading to behave incorrectly.

tornado.autoreload.**start**(*io_loop=None*, *check_time=500*)
Begins watching source files for changes.

Changed in version 4.1: The `io_loop` argument is deprecated.

tornado.autoreload.**wait**()
Wait for a watched file to change, then restart the process.

Intended to be used at the end of scripts like unit test runners, to run the tests again after any source file changes (but see also the command-line interface in *main*)

tornado.autoreload.**watch**(*filename*)
Add a file to the watch list.

All imported modules are watched by default.

tornado.autoreload.**add_reload_hook**(*fn*)
Add a function to be called before reloading the process.

Note that for open file and socket handles it is generally preferable to set the FD_CLOEXEC flag (using fcntl or tornado.platform.auto.set_close_exec) instead of using a reload hook to close them.

tornado.autoreload.**main**()
Command-line wrapper to re-run a script whenever its source changes.

Scripts may be specified by filename or module name:

```
python -m tornado.autoreload -m tornado.test.runtests
python -m tornado.autoreload tornado/test/runtests.py
```

Running a script with this wrapper is similar to calling *tornado.autoreload.wait* at the end of the script, but this wrapper can catch import-time problems like syntax errors that would otherwise prevent the script from reaching its call to *wait*.

4.7.2 tornado.log — Logging support

Logging support for Tornado.

Tornado uses three logger streams:

- tornado.access: Per-request logging for Tornado's HTTP servers (and potentially other servers in the future)
- tornado.application: Logging of errors from application code (i.e. uncaught exceptions from callbacks)
- tornado.general: General-purpose logging, including any errors or warnings from Tornado itself.

These streams may be configured independently using the standard library's logging module. For example, you may wish to send tornado.access logs to a separate file for analysis.

class tornado.log.**LogFormatter**(*color=True, fmt='%(color)s[%(levelname)1.1s %(asctime)s %(module)s:%(lineno)d]%(end_color)s %(message)s', datefmt='%y%m%d %H:%M:%S', colors={40: 1, 10: 4, 20: 2, 30: 3}*)
Log formatter used in Tornado.

Key features of this formatter are:

•Color support when logging to a terminal that supports it.

•Timestamps on every log line.

•Robust against str/bytes encoding problems.

This formatter is enabled automatically by *tornado.options.parse_command_line* or *tornado.options.parse_config_file* (unless --logging=none is used).

Parameters

- **color** (*bool*) – Enables color support.
- **fmt** (*string*) – Log message format. It will be applied to the attributes dict of log records. The text between %(color)s and %(end_color)s will be colored depending on the level if color support is on.
- **colors** (*dict*) – color mappings from logging level to terminal color code
- **datefmt** (*string*) – Datetime format. Used for formatting (asctime) placeholder in prefix_fmt.

Changed in version 3.2: Added `fmt` and `datefmt` arguments.

`tornado.log.`**`enable_pretty_logging`**(*options=None*, *logger=None*)
Turns on formatted logging output as configured.

This is called automatically by *tornado.options.parse_command_line* and *tornado.options.parse_config_file*.

`tornado.log.`**`define_logging_options`**(*options=None*)
Add logging-related flags to `options`.

These options are present automatically on the default options instance; this method is only necessary if you have created your own *OptionParser*.

New in version 4.2: This function existed in prior versions but was broken and undocumented until 4.2.

4.7.3 `tornado.options` — Command-line parsing

A command line parsing module that lets modules define their own options.

Each module defines its own options which are added to the global option namespace, e.g.:

```
from tornado.options import define, options

define("mysql_host", default="127.0.0.1:3306", help="Main user DB")
define("memcache_hosts", default="127.0.0.1:11011", multiple=True,
       help="Main user memcache servers")

def connect():
    db = database.Connection(options.mysql_host)
    ...
```

The `main()` method of your application does not need to be aware of all of the options used throughout your program; they are all automatically loaded when the modules are loaded. However, all modules that define options must have been imported before the command line is parsed.

Your `main()` method can parse the command line or parse a config file with either:

```
tornado.options.parse_command_line()
# or
tornado.options.parse_config_file("/etc/server.conf")
```

Command line formats are what you would expect (`--myoption=myvalue`). Config files are just Python files. Global names become options, e.g.:

```
myoption = "myvalue"
myotheroption = "myothervalue"
```

We support `datetimes`, `timedeltas`, ints, and floats (just pass a `type` kwarg to *define*). We also accept multi-value options. See the documentation for *define()* below.

tornado.options.options is a singleton instance of *OptionParser*, and the top-level functions in this module (*define*, *parse_command_line*, etc) simply call methods on it. You may create additional *OptionParser* instances to define isolated sets of options, such as for subcommands.

Note: By default, several options are defined that will configure the standard `logging` module when *parse_command_line* or *parse_config_file* are called. If you want Tornado to leave the logging configuration alone so you can manage it yourself, either pass `--logging=none` on the command line or do the following to disable it in code:

```
from tornado.options import options, parse_command_line
options.logging = None
parse_command_line()
```

Changed in version 4.3: Dashes and underscores are fully interchangeable in option names; options can be defined, set, and read with any mix of the two. Dashes are typical for command-line usage while config files require underscores.

Global functions

tornado.options.**define**(*name*, *default=None*, *type=None*, *help=None*, *metavar=None*, *multiple=False*, *group=None*, *callback=None*)
Defines an option in the global namespace.

See *OptionParser.define*.

tornado.options.**options**
Global options object. All defined options are available as attributes on this object.

tornado.options.**parse_command_line**(*args=None*, *final=True*)
Parses global options from the command line.

See *OptionParser.parse_command_line*.

tornado.options.**parse_config_file**(*path*, *final=True*)
Parses global options from a config file.

See *OptionParser.parse_config_file*.

tornado.options.**print_help**(*file=sys.stderr*)
Prints all the command line options to stderr (or another file).

See *OptionParser.print_help*.

tornado.options.**add_parse_callback**(*callback*)
Adds a parse callback, to be invoked when option parsing is done.

See *OptionParser.add_parse_callback*

exception tornado.options.**Error**
Exception raised by errors in the options module.

OptionParser class

class tornado.options.**OptionParser**
A collection of options, a dictionary with object-like access.

Normally accessed via static functions in the *tornado.options* module, which reference a global instance.

items()
A sequence of (name, value) pairs.

New in version 3.1.

groups()
The set of option-groups created by `define`.

New in version 3.1.

group_dict(*group*)
The names and values of options in a group.

Useful for copying options into Application settings:

```
from tornado.options import define, parse_command_line, options

define('template_path', group='application')
define('static_path', group='application')

parse_command_line()

application = Application(
    handlers, **options.group_dict('application'))
```

New in version 3.1.

as_dict()
The names and values of all options.

New in version 3.1.

define(*name*, *default=None*, *type=None*, *help=None*, *metavar=None*, *multiple=False*, *group=None*, *callback=None*)
Defines a new command line option.

If `type` is given (one of str, float, int, datetime, or timedelta) or can be inferred from the `default`, we parse the command line arguments based on the given type. If `multiple` is True, we accept comma-separated values, and the option value is always a list.

For multi-value integers, we also accept the syntax `x:y`, which turns into `range(x, y)` - very useful for long integer ranges.

`help` and `metavar` are used to construct the automatically generated command line help string. The help message is formatted like:

```
--name=METAVAR      help string
```

`group` is used to group the defined options in logical groups. By default, command line options are grouped by the file in which they are defined.

Command line option names must be unique globally. They can be parsed from the command line with *parse_command_line* or parsed from a config file with *parse_config_file*.

If a `callback` is given, it will be run with the new value whenever the option is changed. This can be used to combine command-line and file-based options:

```
define("config", type=str, help="path to config file",
       callback=lambda path: parse_config_file(path, final=False))
```

With this definition, options in the file specified by `--config` will override options set earlier on the command line, but can be overridden by later flags.

parse_command_line(*args=None*, *final=True*)
Parses all options given on the command line (defaults to `sys.argv`).

Note that `args[0]` is ignored since it is the program name in `sys.argv`.

We return a list of all arguments that are not parsed as options.

If `final` is `False`, parse callbacks will not be run. This is useful for applications that wish to combine configurations from multiple sources.

parse_config_file(*path*, *final=True*)
Parses and loads the Python config file at the given path.

If `final` is `False`, parse callbacks will not be run. This is useful for applications that wish to combine configurations from multiple sources.

Changed in version 4.1: Config files are now always interpreted as utf-8 instead of the system default encoding.

Changed in version 4.4: The special variable `__file__` is available inside config files, specifying the absolute path to the config file itself.

print_help(*file=None*)
Prints all the command line options to stderr (or another file).

add_parse_callback(*callback*)
Adds a parse callback, to be invoked when option parsing is done.

mockable()
Returns a wrapper around self that is compatible with `mock.patch`.

The `mock.patch` function (included in the standard library `unittest.mock` package since Python 3.3, or in the third-party `mock` package for older versions of Python) is incompatible with objects like `options` that override `__getattr__` and `__setattr__`. This function returns an object that can be used with `mock.patch.object` to modify option values:

```
with mock.patch.object(options.mockable(), 'name', value):
    assert options.name == value
```

4.7.4 tornado.stack_context — Exception handling across asynchronous callbacks

StackContext allows applications to maintain threadlocal-like state that follows execution as it moves to other execution contexts.

The motivating examples are to eliminate the need for explicit `async_callback` wrappers (as in *tornado.web.RequestHandler*), and to allow some additional context to be kept for logging.

This is slightly magic, but it's an extension of the idea that an exception handler is a kind of stack-local state and when that stack is suspended and resumed in a new context that state needs to be preserved. *StackContext* shifts the burden of restoring that state from each call site (e.g. wrapping each *AsyncHTTPClient* callback in `async_callback`) to the mechanisms that transfer control from one context to another (e.g. *AsyncHTTPClient* itself, *IOLoop*, thread pools, etc).

Example usage:

```
@contextlib.contextmanager
def die_on_error():
    try:
        yield
    except Exception:
        logging.error("exception in asynchronous operation",exc_info=True)
        sys.exit(1)

with StackContext(die_on_error):
    # Any exception thrown here *or in callback and its descendants*
    # will cause the process to exit instead of spinning endlessly
    # in the ioloop.
```

```
    http_client.fetch(url, callback)
ioloop.start()
```

Most applications shouldn't have to work with *StackContext* directly. Here are a few rules of thumb for when it's necessary:

- If you're writing an asynchronous library that doesn't rely on a stack_context-aware library like *tornado.ioloop* or *tornado.iostream* (for example, if you're writing a thread pool), use *stack_context.wrap()* before any asynchronous operations to capture the stack context from where the operation was started.
- If you're writing an asynchronous library that has some shared resources (such as a connection pool), create those shared resources within a `with stack_context.NullContext():` block. This will prevent `StackContexts` from leaking from one request to another.
- If you want to write something like an exception handler that will persist across asynchronous calls, create a new *StackContext* (or *ExceptionStackContext*), and make your asynchronous calls in a `with` block that references your *StackContext*.

class `tornado.stack_context.`**`StackContext`**(*context_factory*)
Establishes the given context as a StackContext that will be transferred.

Note that the parameter is a callable that returns a context manager, not the context itself. That is, where for a non-transferable context manager you would say:

```
    with my_context():
```

StackContext takes the function itself rather than its result:

```
    with StackContext(my_context):
```

The result of `with StackContext() as cb:` is a deactivation callback. Run this callback when the StackContext is no longer needed to ensure that it is not propagated any further (note that deactivating a context does not affect any instances of that context that are currently pending). This is an advanced feature and not necessary in most applications.

class `tornado.stack_context.`**`ExceptionStackContext`**(*exception_handler*)
Specialization of StackContext for exception handling.

The supplied `exception_handler` function will be called in the event of an uncaught exception in this context. The semantics are similar to a try/finally clause, and intended use cases are to log an error, close a socket, or similar cleanup actions. The `exc_info` triple `(type, value, traceback)` will be passed to the exception_handler function.

If the exception handler returns true, the exception will be consumed and will not be propagated to other exception handlers.

class `tornado.stack_context.`**`NullContext`**
Resets the *StackContext*.

Useful when creating a shared resource on demand (e.g. an *AsyncHTTPClient*) where the stack that caused the creating is not relevant to future operations.

`tornado.stack_context.`**`wrap`**(*fn*)
Returns a callable object that will restore the current *StackContext* when executed.

Use this whenever saving a callback to be executed later in a different execution context (either in a different thread or asynchronously in the same thread).

`tornado.stack_context.`**`run_with_stack_context`**(*context*, *func*)
Run a coroutine `func` in the given *StackContext*.

It is not safe to have a `yield` statement within a `with StackContext` block, so it is difficult to use stack context with *gen.coroutine*. This helper function runs the function in the correct context while keeping the `yield` and `with` statements syntactically separate.

Example:

```
@gen.coroutine
def incorrect():
    with StackContext(ctx):
        # ERROR: this will raise StackContextInconsistentError
        yield other_coroutine()

@gen.coroutine
def correct():
    yield run_with_stack_context(StackContext(ctx), other_coroutine)
```

New in version 3.1.

4.7.5 `tornado.testing` — Unit testing support for asynchronous code

Support classes for automated testing.

- *AsyncTestCase* and *AsyncHTTPTestCase*: Subclasses of unittest.TestCase with additional support for testing asynchronous (*IOLoop* based) code.
- *ExpectLog* and *LogTrapTestCase*: Make test logs less spammy.
- *main()*: A simple test runner (wrapper around unittest.main()) with support for the tornado.autoreload module to rerun the tests when code changes.

Asynchronous test cases

class `tornado.testing.`**`AsyncTestCase`**(*methodName='runTest'*)

`TestCase` subclass for testing *IOLoop*-based asynchronous code.

The unittest framework is synchronous, so the test must be complete by the time the test method returns. This means that asynchronous code cannot be used in quite the same way as usual. To write test functions that use the same `yield`-based patterns used with the *tornado.gen* module, decorate your test methods with *tornado.testing.gen_test* instead of *tornado.gen.coroutine*. This class also provides the *stop()* and *wait()* methods for a more manual style of testing. The test method itself must call `self.wait()`, and asynchronous callbacks should call `self.stop()` to signal completion.

By default, a new *IOLoop* is constructed for each test and is available as `self.io_loop`. This *IOLoop* should be used in the construction of HTTP clients/servers, etc. If the code being tested requires a global *IOLoop*, subclasses should override *get_new_ioloop* to return it.

The *IOLoop*'s `start` and `stop` methods should not be called directly. Instead, use *self.stop* and *self.wait*. Arguments passed to `self.stop` are returned from `self.wait`. It is possible to have multiple `wait`/`stop` cycles in the same test.

Example:

```
# This test uses coroutine style.
class MyTestCase(AsyncTestCase):
    @tornado.testing.gen_test
    def test_http_fetch(self):
        client = AsyncHTTPClient(self.io_loop)
        response = yield client.fetch("http://www.tornadoweb.org")
```

```
            # Test contents of response
            self.assertIn("FriendFeed", response.body)

    # This test uses argument passing between self.stop and self.wait.
    class MyTestCase2(AsyncTestCase):
        def test_http_fetch(self):
            client = AsyncHTTPClient(self.io_loop)
            client.fetch("http://www.tornadoweb.org/", self.stop)
            response = self.wait()
            # Test contents of response
            self.assertIn("FriendFeed", response.body)

    # This test uses an explicit callback-based style.
    class MyTestCase3(AsyncTestCase):
        def test_http_fetch(self):
            client = AsyncHTTPClient(self.io_loop)
            client.fetch("http://www.tornadoweb.org/", self.handle_fetch)
            self.wait()

        def handle_fetch(self, response):
            # Test contents of response (failures and exceptions here
            # will cause self.wait() to throw an exception and end the
            # test).
            # Exceptions thrown here are magically propagated to
            # self.wait() in test_http_fetch() via stack_context.
            self.assertIn("FriendFeed", response.body)
            self.stop()
```

get_new_ioloop()
Creates a new *IOLoop* for this test. May be overridden in subclasses for tests that require a specific *IOLoop* (usually the singleton *IOLoop.instance()*).

stop(*_arg=None*, ***kwargs*)
Stops the *IOLoop*, causing one pending (or future) call to *wait()* to return.

Keyword arguments or a single positional argument passed to *stop()* are saved and will be returned by *wait()*.

wait(*condition=None*, *timeout=None*)
Runs the *IOLoop* until stop is called or timeout has passed.

In the event of a timeout, an exception will be thrown. The default timeout is 5 seconds; it may be overridden with a `timeout` keyword argument or globally with the `ASYNC_TEST_TIMEOUT` environment variable.

If `condition` is not None, the *IOLoop* will be restarted after *stop()* until `condition()` returns true.

Changed in version 3.1: Added the `ASYNC_TEST_TIMEOUT` environment variable.

class `tornado.testing.`**AsyncHTTPTestCase**(*methodName='runTest'*)
A test case that starts up an HTTP server.

Subclasses must override *get_app()*, which returns the *tornado.web.Application* (or other *HTTPServer* callback) to be tested. Tests will typically use the provided `self.http_client` to fetch URLs from this server.

Example, assuming the "Hello, world" example from the user guide is in `hello.py`:

```
    import hello

    class TestHelloApp(AsyncHTTPTestCase):
        def get_app(self):
            return hello.make_app()

        def test_homepage(self):
            response = self.fetch('/')
            self.assertEqual(response.code, 200)
            self.assertEqual(response.body, 'Hello, world')
```

That call to `self.fetch()` is equivalent to

```
    self.http_client.fetch(self.get_url('/'), self.stop)
    response = self.wait()
```

which illustrates how AsyncTestCase can turn an asynchronous operation, like `http_client.fetch()`, into a synchronous operation. If you need to do other asynchronous operations in tests, you'll probably need to use `stop()` and `wait()` yourself.

get_app()
Should be overridden by subclasses to return a *tornado.web.Application* or other *HTTPServer* callback.

fetch(*path*, ***kwargs*)
Convenience method to synchronously fetch a url.

The given path will be appended to the local server's host and port. Any additional kwargs will be passed directly to *AsyncHTTPClient.fetch* (and so could be used to pass `method="POST"`, `body="..."`, etc).

get_httpserver_options()
May be overridden by subclasses to return additional keyword arguments for the server.

get_http_port()
Returns the port used by the server.

A new port is chosen for each test.

get_url(*path*)
Returns an absolute url for the given path on the test server.

class tornado.testing.**AsyncHTTPSTestCase**(*methodName='runTest'*)
A test case that starts an HTTPS server.

Interface is generally the same as *AsyncHTTPTestCase*.

get_ssl_options()
May be overridden by subclasses to select SSL options.

By default includes a self-signed testing certificate.

tornado.testing.**gen_test**(*func=None*, *timeout=None*)
Testing equivalent of `@gen.coroutine`, to be applied to test methods.

`@gen.coroutine` cannot be used on tests because the *IOLoop* is not already running. `@gen_test` should be applied to test methods on subclasses of *AsyncTestCase*.

Example:

```
    class MyTest(AsyncHTTPTestCase):
        @gen_test
```

```
        def test_something(self):
            response = yield gen.Task(self.fetch('/'))
```

By default, `@gen_test` times out after 5 seconds. The timeout may be overridden globally with the `ASYNC_TEST_TIMEOUT` environment variable, or for each test with the `timeout` keyword argument:

```
    class MyTest(AsyncHTTPTestCase):
        @gen_test(timeout=10)
        def test_something_slow(self):
            response = yield gen.Task(self.fetch('/'))
```

New in version 3.1: The `timeout` argument and `ASYNC_TEST_TIMEOUT` environment variable.

Changed in version 4.0: The wrapper now passes along `*args, **kwargs` so it can be used on functions with arguments.

Controlling log output

class `tornado.testing.`**`ExpectLog`**(*logger*, *regex*, *required=True*)

Context manager to capture and suppress expected log output.

Useful to make tests of error conditions less noisy, while still leaving unexpected log entries visible. *Not thread safe.*

The attribute `logged_stack` is set to true if any exception stack trace was logged.

Usage:

```
    with ExpectLog('tornado.application', "Uncaught exception"):
        error_response = self.fetch("/some_page")
```

Changed in version 4.3: Added the `logged_stack` attribute.

Constructs an ExpectLog context manager.

Parameters

- **`logger`** – Logger object (or name of logger) to watch. Pass an empty string to watch the root logger.
- **`regex`** – Regular expression to match. Any log entries on the specified logger that match this regex will be suppressed.
- **`required`** – If true, an exeption will be raised if the end of the `with` statement is reached without matching any log entries.

class `tornado.testing.`**`LogTrapTestCase`**(*methodName='runTest'*)

A test case that captures and discards all logging output if the test passes.

Some libraries can produce a lot of logging output even when the test succeeds, so this class can be useful to minimize the noise. Simply use it as a base class for your test case. It is safe to combine with AsyncTestCase via multiple inheritance (`class MyTestCase(AsyncHTTPTestCase, LogTrapTestCase):`)

This class assumes that only one log handler is configured and that it is a `StreamHandler`. This is true for both `logging.basicConfig` and the "pretty logging" configured by *`tornado.options`*. It is not compatible with other log buffering mechanisms, such as those provided by some test runners.

Deprecated since version 4.1: Use the unittest module's `--buffer` option instead, or *`ExpectLog`*.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Test runner

tornado.testing.**main**(***kwargs*)
A simple test runner.

This test runner is essentially equivalent to `unittest.main` from the standard library, but adds support for tornado-style option parsing and log formatting.

The easiest way to run a test is via the command line:

```
python -m tornado.testing tornado.test.stack_context_test
```

See the standard library unittest module for ways in which tests can be specified.

Projects with many tests may wish to define a test script like `tornado/test/runtests.py`. This script should define a method `all()` which returns a test suite and then call *`tornado.testing.main()`*. Note that even when a test script is used, the `all()` test suite may be overridden by naming a single test on the command line:

```
# Runs all tests
python -m tornado.test.runtests
# Runs one test
python -m tornado.test.runtests tornado.test.stack_context_test
```

Additional keyword arguments passed through to `unittest.main()`. For example, use `tornado.testing.main(verbosity=2)` to show many test details as they are run. See http://docs.python.org/library/unittest.html#unittest.main for full argument list.

Helper functions

tornado.testing.**bind_unused_port**(*reuse_port=False*)
Binds a server socket to an available port on localhost.

Returns a tuple (socket, port).

Changed in version 4.4: Always binds to `127.0.0.1` without resolving the name `localhost`.

tornado.testing.**get_unused_port**()
Returns a (hopefully) unused port number.

This function does not guarantee that the port it returns is available, only that a series of get_unused_port calls in a single process return distinct ports.

Deprecated since version Use: bind_unused_port instead, which is guaranteed to find an unused port.

tornado.testing.**get_async_test_timeout**()
Get the global timeout setting for async tests.

Returns a float, the timeout in seconds.

New in version 3.1.

4.7.6 `tornado.util` — General-purpose utilities

Miscellaneous utility functions and classes.

This module is used internally by Tornado. It is not necessarily expected that the functions and classes defined here will be useful to other applications, but they are documented here in case they are.

The one public-facing part of this module is the *Configurable* class and its *configure* method, which becomes a part of the interface of its subclasses, including *AsyncHTTPClient*, *IOLoop*, and *Resolver*.

class tornado.util.**ObjectDict**
Makes a dictionary behave like an object, with attribute-style access.

class tornado.util.**GzipDecompressor**
Streaming gzip decompressor.

The interface is like that of `zlib.decompressobj` (without some of the optional arguments, but it understands gzip headers and checksums.

decompress(*value*, *max_length=None*)
Decompress a chunk, returning newly-available data.

Some data may be buffered for later processing; *flush* must be called when there is no more input data to ensure that all data was processed.

If `max_length` is given, some input data may be left over in `unconsumed_tail`; you must retrieve this value and pass it back to a future call to *decompress* if it is not empty.

unconsumed_tail
Returns the unconsumed portion left over

flush()
Return any remaining buffered data not yet returned by decompress.

Also checks for errors such as truncated input. No other methods may be called on this object after *flush*.

tornado.util.**import_object**(*name*)
Imports an object by name.

import_object('x') is equivalent to 'import x'. import_object('x.y.z') is equivalent to 'from x.y import z'.

```
>>> import tornado.escape
>>> import_object('tornado.escape') is tornado.escape
True
>>> import_object('tornado.escape.utf8') is tornado.escape.utf8
True
>>> import_object('tornado') is tornado
True
>>> import_object('tornado.missing_module')
Traceback (most recent call last):
    ...
ImportError: No module named missing_module
```

tornado.util.**errno_from_exception**(*e*)
Provides the errno from an Exception object.

There are cases that the errno attribute was not set so we pull the errno out of the args but if someone instantiates an Exception without any args you will get a tuple error. So this function abstracts all that behavior to give you a safe way to get the errno.

tornado.util.**re_unescape**(*s*)
Unescape a string escaped by `re.escape`.

May raise `ValueError` for regular expressions which could not have been produced by `re.escape` (for example, strings containing `\d` cannot be unescaped).

New in version 4.4.

class tornado.util.**Configurable**
Base class for configurable interfaces.

A configurable interface is an (abstract) class whose constructor acts as a factory function for one of its implementation subclasses. The implementation subclass as well as optional keyword arguments to its initializer can be set globally at runtime with *configure*.

By using the constructor as the factory method, the interface looks like a normal class, `isinstance` works as usual, etc. This pattern is most useful when the choice of implementation is likely to be a global decision (e.g. when `epoll` is available, always use it instead of `select`), or when a previously-monolithic class has been split into specialized subclasses.

Configurable subclasses must define the class methods *configurable_base* and *configurable_default*, and use the instance method *initialize* instead of `__init__`.

classmethod configurable_base()
Returns the base class of a configurable hierarchy.

This will normally return the class in which it is defined. (which is *not* necessarily the same as the cls classmethod parameter).

classmethod configurable_default()
Returns the implementation class to be used if none is configured.

initialize()
Initialize a *Configurable* subclass instance.

Configurable classes should use *initialize* instead of `__init__`.

Changed in version 4.2: Now accepts positional arguments in addition to keyword arguments.

classmethod configure(*impl*, ***kwargs*)
Sets the class to use when the base class is instantiated.

Keyword arguments will be saved and added to the arguments passed to the constructor. This can be used to set global defaults for some parameters.

classmethod configured_class()
Returns the currently configured class.

class `tornado.util.`**ArgReplacer**(*func*, *name*)
Replaces one value in an `args, kwargs` pair.

Inspects the function signature to find an argument by name whether it is passed by position or keyword. For use in decorators and similar wrappers.

get_old_value(*args*, *kwargs*, *default=None*)
Returns the old value of the named argument without replacing it.

Returns `default` if the argument is not present.

replace(*new_value*, *args*, *kwargs*)
Replace the named argument in `args, kwargs` with `new_value`.

Returns `(old_value, args, kwargs)`. The returned `args` and `kwargs` objects may not be the same as the input objects, or the input objects may be mutated.

If the named argument was not found, `new_value` will be added to `kwargs` and None will be returned as `old_value`.

`tornado.util.`**timedelta_to_seconds**(*td*)
Equivalent to td.total_seconds() (introduced in python 2.7).

4.8 Frequently Asked Questions

4.8.1 Why isn't this example with `time.sleep()` running in parallel?

Many people's first foray into Tornado's concurrency looks something like this:

```
class BadExampleHandler(RequestHandler):
    def get(self):
        for i in range(5):
            print(i)
            time.sleep(1)
```

Fetch this handler twice at the same time and you'll see that the second five-second countdown doesn't start until the first one has completely finished. The reason for this is that `time.sleep` is a **blocking** function: it doesn't allow control to return to the *`IOLoop`* so that other handlers can be run.

Of course, `time.sleep` is really just a placeholder in these examples, the point is to show what happens when something in a handler gets slow. No matter what the real code is doing, to achieve concurrency blocking code must be replaced with non-blocking equivalents. This means one of three things:

1. *Find a coroutine-friendly equivalent.* For `time.sleep`, use *`tornado.gen.sleep`* instead:

```
    class CoroutineSleepHandler(RequestHandler):
        @gen.coroutine
        def get(self):
            for i in range(5):
                print(i)
                yield gen.sleep(1)
```

   When this option is available, it is usually the best approach. See the Tornado wiki for links to asynchronous libraries that may be useful.

2. *Find a callback-based equivalent.* Similar to the first option, callback-based libraries are available for many tasks, although they are slightly more complicated to use than a library designed for coroutines. These are typically used with *`tornado.gen.Task`* as an adapter:

```
    class CoroutineTimeoutHandler(RequestHandler):
        @gen.coroutine
        def get(self):
            io_loop = IOLoop.current()
            for i in range(5):
                print(i)
                yield gen.Task(io_loop.add_timeout, io_loop.time() + 1)
```

   Again, the Tornado wiki can be useful to find suitable libraries.

3. *Run the blocking code on another thread.* When asynchronous libraries are not available, `concurrent.futures.ThreadPoolExecutor` can be used to run any blocking code on another thread. This is a universal solution that can be used for any blocking function whether an asynchronous counterpart exists or not:

```
    executor = concurrent.futures.ThreadPoolExecutor(8)

    class ThreadPoolHandler(RequestHandler):
        @gen.coroutine
        def get(self):
```

```
            for i in range(5):
                print(i)
                yield executor.submit(time.sleep, 1)
```

See the Asynchronous I/O chapter of the Tornado user's guide for more on blocking and asynchronous functions.

4.8.2 My code is asynchronous, but it's not running in parallel in two browser tabs.

Even when a handler is asynchronous and non-blocking, it can be surprisingly tricky to verify this. Browsers will recognize that you are trying to load the same page in two different tabs and delay the second request until the first has finished. To work around this and see that the server is in fact working in parallel, do one of two things:

- Add something to your urls to make them unique. Instead of `http://localhost:8888` in both tabs, load `http://localhost:8888/?x=1` in one and `http://localhost:8888/?x=2` in the other.
- Use two different browsers. For example, Firefox will be able to load a url even while that same url is being loaded in a Chrome tab.

4.9 Release notes

4.9.1 What's new in Tornado 4.4.2

Oct 1, 2016

Security fixes

- A difference in cookie parsing between Tornado and web browsers (especially when combined with Google Analytics) could allow an attacker to set arbitrary cookies and bypass XSRF protection. The cookie parser has been rewritten to fix this attack.

Backwards-compatibility notes

- Cookies containing certain special characters (in particular semicolon and square brackets) are now parsed differently.
- If the cookie header contains a combination of valid and invalid cookies, the valid ones will be returned (older versions of Tornado would reject the entire header for a single invalid cookie).

4.9.2 What's new in Tornado 4.4.1

Jul 23, 2016

`tornado.web`

- Fixed a regression in Tornado 4.4 which caused URL regexes containing backslash escapes outside capturing groups to be rejected.

4.9.3 What's new in Tornado 4.4

Jul 15, 2016

General

- Tornado now requires Python 2.7 or 3.3+; versions 2.6 and 3.2 are no longer supported. Pypy3 is still supported even though its latest release is mainly based on Python 3.2.
- The monotonic package is now supported as an alternative to Monotime for monotonic clock support on Python 2.

`tornado.curl_httpclient`

- Failures in `_curl_setup_request` no longer cause the `max_clients` pool to be exhausted.
- Non-ascii header values are now handled correctly.

`tornado.gen`

- *`with_timeout`* now accepts any yieldable object (except *`YieldPoint`*), not just *`tornado.concurrent.Future`*.

`tornado.httpclient`

- The errors raised by timeouts now indicate what state the request was in; the error message is no longer simply "599 Timeout".
- Calling `repr` on a *`tornado.httpclient.HTTPError`* no longer raises an error.

`tornado.httpserver`

- Int-like enums (including `http.HTTPStatus`) can now be used as status codes.
- Responses with status code `204 No Content` no longer emit a `Content-Length: 0` header.

`tornado.ioloop`

- Improved performance when there are large numbers of active timeouts.

`tornado.netutil`

- All included *`Resolver`* implementations raise `IOError` (or a subclass) for any resolution failure.

`tornado.options`

- Options can now be modified with subscript syntax in addition to attribute syntax.
- The special variable `__file__` is now available inside config files.

tornado.simple_httpclient

- HTTP/1.0 (not 1.1) responses without a `Content-Length` header now work correctly.

tornado.tcpserver

- *`TCPServer.bind`* now accepts a `reuse_port` argument.

tornado.testing

- Test sockets now always use `127.0.0.1` instead of `localhost`. This avoids conflicts when the automatically-assigned port is available on IPv4 but not IPv6, or in unusual network configurations when `localhost` has multiple IP addresses.

tornado.web

- `image/svg+xml` is now on the list of compressible mime types.
- Fixed an error on Python 3 when compression is used with multiple `Vary` headers.

tornado.websocket

- `WebSocketHandler.__init__` now uses `super`, which improves support for multiple inheritance.

4.9.4 What's new in Tornado 4.3

Nov 6, 2015

Highlights

- The new async/await keywords in Python 3.5 are supported. In most cases, `async def` can be used in place of the `@gen.coroutine` decorator. Inside a function defined with `async def`, use `await` instead of `yield` to wait on an asynchronous operation. Coroutines defined with async/await will be faster than those defined with `@gen.coroutine` and `yield`, but do not support some features including *`Callback`*/*`Wait`* or the ability to yield a Twisted `Deferred`. See *the users' guide* for more.
- The async/await keywords are also available when compiling with Cython in older versions of Python.

Deprecation notice

- This will be the last release of Tornado to support Python 2.6 or 3.2. Note that PyPy3 will continue to be supported even though it implements a mix of Python 3.2 and 3.3 features.

Installation

- Tornado has several new dependencies: `ordereddict` on Python 2.6, `singledispatch` on all Python versions prior to 3.4 (This was an optional dependency in prior versions of Tornado, and is now mandatory), and `backports_abc>=0.4` on all versions prior to 3.5. These dependencies will be installed automatically when installing with `pip` or `setup.py install`. These dependencies will not be required when running on Google App Engine.
- Binary wheels are provided for Python 3.5 on Windows (32 and 64 bit).

tornado.auth

- New method *`OAuth2Mixin.oauth2_request`* can be used to make authenticated requests with an access token.
- Now compatible with callbacks that have been compiled with Cython.

tornado.autoreload

- Fixed an issue with the autoreload command-line wrapper in which imports would be incorrectly interpreted as relative.

tornado.curl_httpclient

- Fixed parsing of multi-line headers.
- `allow_nonstandard_methods=True` now bypasses body sanity checks, in the same way as in `simple_httpclient`.
- The `PATCH` method now allows a body without `allow_nonstandard_methods=True`.

tornado.gen

- *`WaitIterator`* now supports the `async for` statement on Python 3.5.
- `@gen.coroutine` can be applied to functions compiled with Cython. On python versions prior to 3.5, the `backports_abc` package must be installed for this functionality.
- `Multi` and *`multi_future`* are deprecated and replaced by a unified function *`multi`*.

tornado.httpclient

- *`tornado.httpclient.HTTPError`* is now copyable with the `copy` module.

tornado.httpserver

- Requests containing both `Content-Length` and `Transfer-Encoding` will be treated as an error.

tornado.httputil

- *HTTPHeaders* can now be pickled and unpickled.

tornado.ioloop

- `IOLoop(make_current=True)` now works as intended instead of raising an exception.
- The Twisted and asyncio IOLoop implementations now clear `current()` when they exit, like the standard IOLoops.
- *IOLoop.add_callback* is faster in the single-threaded case.
- *IOLoop.add_callback* no longer raises an error when called on a closed IOLoop, but the callback will not be invoked.

tornado.iostream

- Coroutine-style usage of *IOStream* now converts most errors into *StreamClosedError*, which has the effect of reducing log noise from exceptions that are outside the application's control (especially SSL errors).
- *StreamClosedError* now has a `real_error` attribute which indicates why the stream was closed. It is the same as the `error` attribute of *IOStream* but may be more easily accessible than the *IOStream* itself.
- Improved error handling in *read_until_close*.
- Logging is less noisy when an SSL server is port scanned.
- `EINTR` is now handled on all reads.

tornado.locale

- *tornado.locale.load_translations* now accepts encodings other than UTF-8. UTF-16 and UTF-8 will be detected automatically if a BOM is present; for other encodings *load_translations* has an `encoding` parameter.

tornado.locks

- *Lock* and *Semaphore* now support the `async with` statement on Python 3.5.

tornado.log

- A new time-based log rotation mode is available with `--log_rotate_mode=time`, `--log-rotate-when`, and `log-rotate-interval`.

tornado.netutil

- *bind_sockets* now supports `SO_REUSEPORT` with the `reuse_port=True` argument.

tornado.options

- Dashes and underscores are now fully interchangeable in option names.

tornado.queues

- *Queue* now supports the `async for` statement on Python 3.5.

tornado.simple_httpclient

- When following redirects, `streaming_callback` and `header_callback` will no longer be run on the redirect responses (only the final non-redirect).
- Responses containing both `Content-Length` and `Transfer-Encoding` will be treated as an error.

tornado.template

- *tornado.template.ParseError* now includes the filename in addition to line number.
- Whitespace handling has become more configurable. The *Loader* constructor now has a `whitespace` argument, there is a new `template_whitespace` *Application* setting, and there is a new `{% whitespace %}` template directive. All of these options take a mode name defined in the *tornado.template.filter_whitespace* function. The default mode is `single`, which is the same behavior as prior versions of Tornado.
- Non-ASCII filenames are now supported.

tornado.testing

- *ExpectLog* objects now have a boolean `logged_stack` attribute to make it easier to test whether an exception stack trace was logged.

tornado.web

- The hard limit of 4000 bytes per outgoing header has been removed.
- *StaticFileHandler* returns the correct `Content-Type` for files with `.gz`, `.bz2`, and `.xz` extensions.
- Responses smaller than 1000 bytes will no longer be compressed.
- The default gzip compression level is now 6 (was 9).
- Fixed a regression in Tornado 4.2.1 that broke *StaticFileHandler* with a `path` of `/`.
- *tornado.web.HTTPError* is now copyable with the `copy` module.
- The exception *Finish* now accepts an argument which will be passed to the method *RequestHandler.finish*.
- New *Application* setting `xsrf_cookie_kwargs` can be used to set additional attributes such as `secure` or `httponly` on the XSRF cookie.
- *Application.listen* now returns the *HTTPServer* it created.

tornado.websocket

- Fixed handling of continuation frames when compression is enabled.

4.9.5 What's new in Tornado 4.2.1

Jul 17, 2015

Security fix

- This release fixes a path traversal vulnerability in *StaticFileHandler*, in which files whose names *started with* the static_path directory but were not actually *in* that directory could be accessed.

4.9.6 What's new in Tornado 4.2

May 26, 2015

Backwards-compatibility notes

- SSLIOStream.connect and *IOStream.start_tls* now validate certificates by default.
- Certificate validation will now use the system CA root certificates instead of certifi when possible (i.e. Python 2.7.9+ or 3.4+). This includes *IOStream* and simple_httpclient, but not curl_httpclient.
- The default SSL configuration has become stricter, using ssl.create_default_context where available on the client side. (On the server side, applications are encouraged to migrate from the ssl_options dict-based API to pass an ssl.SSLContext instead).
- The deprecated classes in the *tornado.auth* module, GoogleMixin, FacebookMixin, and FriendFeedMixin have been removed.

New modules: tornado.locks and tornado.queues

These modules provide classes for coordinating coroutines, merged from Toro.

To port your code from Toro's queues to Tornado 4.2, import *Queue*, *PriorityQueue*, or *LifoQueue* from *tornado.queues* instead of from toro.

Use *Queue* instead of Toro's JoinableQueue. In Tornado the methods *join* and *task_done* are available on all queues, not on a special JoinableQueue.

Tornado queues raise exceptions specific to Tornado instead of reusing exceptions from the Python standard library. Therefore instead of catching the standard queue.Empty exception from *Queue.get_nowait*, catch the special *tornado.queues.QueueEmpty* exception, and instead of catching the standard queue.Full from *Queue.get_nowait*, catch *tornado.queues.QueueFull*.

To port from Toro's locks to Tornado 4.2, import *Condition*, *Event*, *Semaphore*, *BoundedSemaphore*, or *Lock* from *tornado.locks* instead of from toro.

Toro's Semaphore.wait allowed a coroutine to wait for the semaphore to be unlocked *without* acquiring it. This encouraged unorthodox patterns; in Tornado, just use *acquire*.

Toro's Event.wait raised a Timeout exception after a timeout. In Tornado, *Event.wait* raises *tornado.gen.TimeoutError*.

Toro's `Condition.wait` also raised `Timeout`, but in Tornado, the *`Future`* returned by *`Condition.wait`* resolves to False after a timeout:

```
@gen.coroutine
def await_notification():
    if not (yield condition.wait(timeout=timedelta(seconds=1))):
        print('timed out')
    else:
        print('condition is true')
```

In lock and queue methods, wherever Toro accepted `deadline` as a keyword argument, Tornado names the argument `timeout` instead.

Toro's `AsyncResult` is not merged into Tornado, nor its exceptions `NotReady` and `AlreadySet`. Use a *`Future`* instead. If you wrote code like this:

```
from tornado import gen
import toro

result = toro.AsyncResult()

@gen.coroutine
def setter():
    result.set(1)

@gen.coroutine
def getter():
    value = yield result.get()
    print(value)  # Prints "1".
```

Then the Tornado equivalent is:

```
from tornado import gen
from tornado.concurrent import Future

result = Future()

@gen.coroutine
def setter():
    result.set_result(1)

@gen.coroutine
def getter():
    value = yield result
    print(value)  # Prints "1".
```

`tornado.autoreload`

- Improved compatibility with Windows.
- Fixed a bug in Python 3 if a module was imported during a reload check.

`tornado.concurrent`

- *`run_on_executor`* now accepts arguments to control which attributes it uses to find the *`IOLoop`* and executor.

tornado.curl_httpclient

- Fixed a bug that would cause the client to stop processing requests if an exception occurred in certain places while there is a queue.

tornado.escape

- *xhtml_escape* now supports numeric character references in hex format (`&#x20;`)

tornado.gen

- *WaitIterator* no longer uses weak references, which fixes several garbage-collection-related bugs.
- `tornado.gen.Multi` and *tornado.gen.multi_future* (which are used when yielding a list or dict in a coroutine) now log any exceptions after the first if more than one *Future* fails (previously they would be logged when the *Future* was garbage-collected, but this is more reliable). Both have a new keyword argument `quiet_exceptions` to suppress logging of certain exception types; to use this argument you must call `Multi` or `multi_future` directly instead of simply yielding a list.
- *multi_future* now works when given multiple copies of the same *Future*.
- On Python 3, catching an exception in a coroutine no longer leads to leaks via `Exception.__context__`.

tornado.httpclient

- The `raise_error` argument now works correctly with the synchronous *HTTPClient*.
- The synchronous *HTTPClient* no longer interferes with *IOLoop.current()*.

tornado.httpserver

- *HTTPServer* is now a subclass of *tornado.util.Configurable*.

tornado.httputil

- *HTTPHeaders* can now be copied with `copy.copy` and `copy.deepcopy`.

tornado.ioloop

- The *IOLoop* constructor now has a `make_current` keyword argument to control whether the new *IOLoop* becomes *IOLoop.current()*.
- Third-party implementations of *IOLoop* should accept `**kwargs` in their *initialize* methods and pass them to the superclass implementation.
- *PeriodicCallback* is now more efficient when the clock jumps forward by a large amount.

tornado.iostream

- `SSLIOStream.connect` and *IOStream.start_tls* now validate certificates by default.
- New method *SSLIOStream.wait_for_handshake* allows server-side applications to wait for the handshake to complete in order to verify client certificates or use NPN/ALPN.
- The *Future* returned by `SSLIOStream.connect` now resolves after the handshake is complete instead of as soon as the TCP connection is established.
- Reduced logging of SSL errors.
- *BaseIOStream.read_until_close* now works correctly when a `streaming_callback` is given but `callback` is None (i.e. when it returns a *Future*)

tornado.locale

- New method *GettextLocale.pgettext* allows additional context to be supplied for gettext translations.

tornado.log

- *define_logging_options* now works correctly when given a non-default `options` object.

tornado.process

- New method *Subprocess.wait_for_exit* is a coroutine-friendly version of *Subprocess.set_exit_callback*.

tornado.simple_httpclient

- Improved performance on Python 3 by reusing a single `ssl.SSLContext`.
- New constructor argument `max_body_size` controls the maximum response size the client is willing to accept. It may be bigger than `max_buffer_size` if `streaming_callback` is used.

tornado.tcpserver

- *TCPServer.handle_stream* may be a coroutine (so that any exceptions it raises will be logged).

tornado.util

- *import_object* now supports unicode strings on Python 2.
- *Configurable.initialize* now supports positional arguments.

tornado.web

- Key versioning support for cookie signing. `cookie_secret` application setting can now contain a dict of valid keys with version as key. The current signing key then must be specified via `key_version` setting.
- Parsing of the `If-None-Match` header now follows the RFC and supports weak validators.

- Passing `secure=False` or `httponly=False` to *`RequestHandler.set_cookie`* now works as expected (previously only the presence of the argument was considered and its value was ignored).
- *`RequestHandler.get_arguments`* now requires that its `strip` argument be of type bool. This helps prevent errors caused by the slightly dissimilar interfaces between the singular and plural methods.
- Errors raised in `_handle_request_exception` are now logged more reliably.
- *`RequestHandler.redirect`* now works correctly when called from a handler whose path begins with two slashes.
- Passing messages containing `%` characters to *`tornado.web.HTTPError`* no longer causes broken error messages.

`tornado.websocket`

- The `on_close` method will no longer be called more than once.
- When the other side closes a connection, we now echo the received close code back instead of sending an empty close frame.

4.9.7 What's new in Tornado 4.1

Feb 7, 2015

Highlights

- If a *`Future`* contains an exception but that exception is never examined or re-raised (e.g. by yielding the *`Future`*), a stack trace will be logged when the *`Future`* is garbage-collected.
- New class *`tornado.gen.WaitIterator`* provides a way to iterate over `Futures` in the order they resolve.
- The *`tornado.websocket`* module now supports compression via the "permessage-deflate" extension. Override *`WebSocketHandler.get_compression_options`* to enable on the server side, and use the `compression_options` keyword argument to *`websocket_connect`* on the client side.
- When the appropriate packages are installed, it is possible to yield `asyncio.Future` or Twisted `Defered` objects in Tornado coroutines.

Backwards-compatibility notes

- *`HTTPServer`* now calls `start_request` with the correct arguments. This change is backwards-incompatible, afffecting any application which implemented *`HTTPServerConnectionDelegate`* by following the example of *`Application`* instead of the documented method signatures.

`tornado.concurrent`

- If a *`Future`* contains an exception but that exception is never examined or re-raised (e.g. by yielding the *`Future`*), a stack trace will be logged when the *`Future`* is garbage-collected.
- *`Future`* now catches and logs exceptions in its callbacks.

tornado.curl_httpclient

- `tornado.curl_httpclient` now supports request bodies for `PATCH` and custom methods.
- `tornado.curl_httpclient` now supports resubmitting bodies after following redirects for methods other than `POST`.
- `curl_httpclient` now runs the streaming and header callbacks on the IOLoop.
- `tornado.curl_httpclient` now uses its own logger for debug output so it can be filtered more easily.

tornado.gen

- New class *tornado.gen.WaitIterator* provides a way to iterate over `Futures` in the order they resolve.
- When the `singledispatch` library is available (standard on Python 3.4, available via `pip install singledispatch` on older versions), the *convert_yielded* function can be used to make other kinds of objects yieldable in coroutines.
- New function *tornado.gen.sleep* is a coroutine-friendly analogue to `time.sleep`.
- *gen.engine* now correctly captures the stack context for its callbacks.

tornado.httpclient

- *tornado.httpclient.HTTPRequest* accepts a new argument `raise_error=False` to suppress the default behavior of raising an error for non-200 response codes.

tornado.httpserver

- *HTTPServer* now calls `start_request` with the correct arguments. This change is backwards-incompatible, afffecting any application which implemented *HTTPServerConnectionDelegate* by following the example of *Application* instead of the documented method signatures.
- *HTTPServer* now tolerates extra newlines which are sometimes inserted between requests on keep-alive connections.
- *HTTPServer* can now use keep-alive connections after a request with a chunked body.
- *HTTPServer* now always reports `HTTP/1.1` instead of echoing the request version.

tornado.httputil

- New function *tornado.httputil.split_host_and_port* for parsing the `netloc` portion of URLs.
- The `context` argument to *HTTPServerRequest* is now optional, and if a context is supplied the `remote_ip` attribute is also optional.
- *HTTPServerRequest.body* is now always a byte string (previously the default empty body would be a unicode string on python 3).
- Header parsing now works correctly when newline-like unicode characters are present.
- Header parsing again supports both CRLF and bare LF line separators.

- Malformed `multipart/form-data` bodies will always be logged quietly instead of raising an unhandled exception; previously the behavior was inconsistent depending on the exact error.

tornado.ioloop

- The `kqueue` and `select` IOLoop implementations now report writeability correctly, fixing flow control in IOStream.
- When a new *IOLoop* is created, it automatically becomes "current" for the thread if there is not already a current instance.
- New method *PeriodicCallback.is_running* can be used to see whether the *PeriodicCallback* has been started.

tornado.iostream

- *IOStream.start_tls* now uses the `server_hostname` parameter for certificate validation.
- *SSLIOStream* will no longer consume 100% CPU after certain error conditions.
- *SSLIOStream* no longer logs `EBADF` errors during the handshake as they can result from nmap scans in certain modes.

tornado.options

- *parse_config_file* now always decodes the config file as utf8 on Python 3.
- *tornado.options.define* more accurately finds the module defining the option.

tornado.platform.asyncio

- It is now possible to yield `asyncio.Future` objects in coroutines when the `singledispatch` library is available and `tornado.platform.asyncio` has been imported.
- New methods *tornado.platform.asyncio.to_tornado_future* and *to_asyncio_future* convert between the two libraries' *Future* classes.

tornado.platform.twisted

- It is now possible to yield `Deferred` objects in coroutines when the `singledispatch` library is available and `tornado.platform.twisted` has been imported.

tornado.tcpclient

- *TCPClient* will no longer raise an exception due to an ill-timed timeout.

tornado.tcpserver

- *TCPServer* no longer ignores its `read_chunk_size` argument.

tornado.testing

- *AsyncTestCase* has better support for multiple exceptions. Previously it would silently swallow all but the last; now it raises the first and logs all the rest.
- *AsyncTestCase* now cleans up *Subprocess* state on `tearDown` when necessary.

tornado.web

- The *asynchronous* decorator now understands `concurrent.futures.Future` in addition to *tornado.concurrent.Future*.
- *StaticFileHandler* no longer logs a stack trace if the connection is closed while sending the file.
- *RequestHandler.send_error* now supports a `reason` keyword argument, similar to *tornado.web.HTTPError*.
- *RequestHandler.locale* now has a property setter.
- *Application.add_handlers* hostname matching now works correctly with IPv6 literals.
- Redirects for the *Application* `default_host` setting now match the request protocol instead of redirecting HTTPS to HTTP.
- Malformed `_xsrf` cookies are now ignored instead of causing uncaught exceptions.
- `Application.start_request` now has the same signature as *HTTPServerConnectionDelegate.start_reques*

tornado.websocket

- The *tornado.websocket* module now supports compression via the "permessage-deflate" extension. Override *WebSocketHandler.get_compression_options* to enable on the server side, and use the `compression_options` keyword argument to *websocket_connect* on the client side.
- *WebSocketHandler* no longer logs stack traces when the connection is closed.
- *WebSocketHandler.open* now accepts `*args, **kw` for consistency with `RequestHandler.get` and related methods.
- The `Sec-WebSocket-Version` header now includes all supported versions.
- *websocket_connect* now has a `on_message_callback` keyword argument for callback-style use without `read_message()`.

4.9.8 What's new in Tornado 4.0.2

Sept 10, 2014

Bug fixes

- Fixed a bug that could sometimes cause a timeout to fire after being cancelled.
- *AsyncTestCase* once again passes along arguments to test methods, making it compatible with extensions such as Nose's test generators.
- *StaticFileHandler* can again compress its responses when gzip is enabled.
- `simple_httpclient` passes its `max_buffer_size` argument to the underlying stream.

- Fixed a reference cycle that can lead to increased memory consumption.
- *add_accept_handler* will now limit the number of times it will call accept per *IOLoop* iteration, addressing a potential starvation issue.
- Improved error handling in *IOStream.connect* (primarily for FreeBSD systems)

4.9.9 What's new in Tornado 4.0.1

Aug 12, 2014

- The build will now fall back to pure-python mode if the C extension fails to build for any reason (previously it would fall back for some errors but not others).
- *IOLoop.call_at* and *IOLoop.call_later* now always return a timeout handle for use with *IOLoop.remove_timeout*.
- If any callback of a *PeriodicCallback* or *IOStream* returns a *Future*, any error raised in that future will now be logged (similar to the behavior of *IOLoop.add_callback*).
- Fixed an exception in client-side websocket connections when the connection is closed.
- simple_httpclient once again correctly handles 204 status codes with no content-length header.
- Fixed a regression in simple_httpclient that would result in timeouts for certain kinds of errors.

4.9.10 What's new in Tornado 4.0

July 15, 2014

Highlights

- The *tornado.web.stream_request_body* decorator allows large files to be uploaded with limited memory usage.
- Coroutines are now faster and are used extensively throughout Tornado itself. More methods now return *Futures*, including most *IOStream* methods and *RequestHandler.flush*.
- Many user-overridden methods are now allowed to return a *Future* for flow control.
- HTTP-related code is now shared between the *tornado.httpserver*, tornado.simple_httpclient and *tornado.wsgi* modules, making support for features such as chunked and gzip encoding more consistent. *HTTPServer* now uses new delegate interfaces defined in *tornado.httputil* in addition to its old single-callback interface.
- New module *tornado.tcpclient* creates TCP connections with non-blocking DNS, SSL handshaking, and support for IPv6.

Backwards-compatibility notes

- *tornado.concurrent.Future* is no longer thread-safe; use concurrent.futures.Future when thread-safety is needed.
- Tornado now depends on the certifi package instead of bundling its own copy of the Mozilla CA list. This will be installed automatically when using pip or easy_install.

- This version includes the changes to the secure cookie format first introduced in version 3.2.1, and the xsrf token change in version 3.2.2. If you are upgrading from an earlier version, see those versions' release notes.
- WebSocket connections from other origin sites are now rejected by default. To accept cross-origin websocket connections, override the new method *WebSocketHandler.check_origin*.
- *WebSocketHandler* no longer supports the old `draft 76` protocol (this mainly affects Safari 5.x browsers). Applications should use non-websocket workarounds for these browsers.
- Authors of alternative *IOLoop* implementations should see the changes to *IOLoop.add_handler* in this release.
- The `RequestHandler.async_callback` and `WebSocketHandler.async_callback` wrapper functions have been removed; they have been obsolete for a long time due to stack contexts (and more recently coroutines).
- `curl_httpclient` now requires a minimum of libcurl version 7.21.1 and pycurl 7.18.2.
- Support for `RequestHandler.get_error_html` has been removed; override *RequestHandler.write_error* instead.

Other notes

- The git repository has moved to https://github.com/tornadoweb/tornado. All old links should be redirected to the new location.
- An announcement mailing list is now available.
- All Tornado modules are now importable on Google App Engine (although the App Engine environment does not allow the system calls used by *IOLoop* so many modules are still unusable).

tornado.auth

- Fixed a bug in `.FacebookMixin` on Python 3.
- When using the *Future* interface, exceptions are more reliably delivered to the caller.

tornado.concurrent

- *tornado.concurrent.Future* is now always thread-unsafe (previously it would be thread-safe if the `concurrent.futures` package was available). This improves performance and provides more consistent semantics. The parts of Tornado that accept Futures will accept both Tornado's thread-unsafe Futures and the thread-safe `concurrent.futures.Future`.
- *tornado.concurrent.Future* now includes all the functionality of the old `TracebackFuture` class. `TracebackFuture` is now simply an alias for `Future`.

tornado.curl_httpclient

- `curl_httpclient` now passes along the HTTP "reason" string in `response.reason`.

tornado.gen

- Performance of coroutines has been improved.
- Coroutines no longer generate `StackContexts` by default, but they will be created on demand when needed.
- The internals of the *tornado.gen* module have been rewritten to improve performance when using `Futures`, at the expense of some performance degradation for the older *YieldPoint* interfaces.
- New function *with_timeout* wraps a *Future* and raises an exception if it doesn't complete in a given amount of time.
- New object *moment* can be yielded to allow the IOLoop to run for one iteration before resuming.
- *Task* is now a function returning a *Future* instead of a *YieldPoint* subclass. This change should be transparent to application code, but allows *Task* to take advantage of the newly-optimized *Future* handling.

tornado.http1connection

- New module contains the HTTP implementation shared by *tornado.httpserver* and `tornado.simple_httpclient`.

tornado.httpclient

- The command-line HTTP client (`python -m tornado.httpclient $URL`) now works on Python 3.
- Fixed a memory leak in *AsyncHTTPClient* shutdown that affected applications that created many HTTP clients and IOLoops.
- New client request parameter `decompress_response` replaces the existing `use_gzip` parameter; both names are accepted.

tornado.httpserver

- `tornado.httpserver.HTTPRequest` has moved to *tornado.httputil.HTTPServerRequest*.
- HTTP implementation has been unified with `tornado.simple_httpclient` in *tornado.http1connection*.
- Now supports `Transfer-Encoding: chunked` for request bodies.
- Now supports `Content-Encoding: gzip` for request bodies if `decompress_request=True` is passed to the *HTTPServer* constructor.
- The `connection` attribute of *HTTPServerRequest* is now documented for public use; applications are expected to write their responses via the *HTTPConnection* interface.
- The *HTTPServerRequest.write* and *HTTPServerRequest.finish* methods are now deprecated. (*RequestHandler.write* and *RequestHandler.finish* are *not* deprecated; this only applies to the methods on *HTTPServerRequest*)
- *HTTPServer* now supports *HTTPServerConnectionDelegate* in addition to the old `request_callback` interface. The delegate interface supports streaming of request bodies.
- *HTTPServer* now detects the error of an application sending a `Content-Length` error that is inconsistent with the actual content.

- New constructor arguments `max_header_size` and `max_body_size` allow separate limits to be set for different parts of the request. `max_body_size` is applied even in streaming mode.
- New constructor argument `chunk_size` can be used to limit the amount of data read into memory at one time per request.
- New constructor arguments `idle_connection_timeout` and `body_timeout` allow time limits to be placed on the reading of requests.
- Form-encoded message bodies are now parsed for all HTTP methods, not just `POST`, `PUT`, and `PATCH`.

tornado.httputil

- *HTTPServerRequest* was moved to this module from *tornado.httpserver*.
- New base classes *HTTPConnection*, *HTTPServerConnectionDelegate*, and *HTTPMessageDelegate* define the interaction between applications and the HTTP implementation.

tornado.ioloop

- *IOLoop.add_handler* and related methods now accept file-like objects in addition to raw file descriptors. Passing the objects is recommended (when possible) to avoid a garbage-collection-related problem in unit tests.
- New method *IOLoop.clear_instance* makes it possible to uninstall the singleton instance.
- Timeout scheduling is now more robust against slow callbacks.
- *IOLoop.add_timeout* is now a bit more efficient.
- When a function run by the *IOLoop* returns a *Future* and that *Future* has an exception, the *IOLoop* will log the exception.
- New method *IOLoop.spawn_callback* simplifies the process of launching a fire-and-forget callback that is separated from the caller's stack context.
- New methods *IOLoop.call_later* and *IOLoop.call_at* simplify the specification of relative or absolute timeouts (as opposed to *add_timeout*, which used the type of its argument).

tornado.iostream

- The `callback` argument to most *IOStream* methods is now optional. When called without a callback the method will return a *Future* for use with coroutines.
- New method *IOStream.start_tls* converts an *IOStream* to an *SSLIOStream*.
- No longer gets confused when an `IOError` or `OSError` without an `errno` attribute is raised.
- *BaseIOStream.read_bytes* now accepts a `partial` keyword argument, which can be used to return before the full amount has been read. This is a more coroutine-friendly alternative to `streaming_callback`.
- *BaseIOStream.read_until* and `read_until_regex` now acept a `max_bytes` keyword argument which will cause the request to fail if it cannot be satisfied from the given number of bytes.
- *IOStream* no longer reads from the socket into memory if it does not need data to satisfy a pending read. As a side effect, the close callback will not be run immediately if the other side closes the connection while there is unconsumed data in the buffer.
- The default `chunk_size` has been increased to 64KB (from 4KB)

- The *IOStream* constructor takes a new keyword argument `max_write_buffer_size` (defaults to unlimited). Calls to *BaseIOStream.write* will raise *StreamBufferFullError* if the amount of unsent buffered data exceeds this limit.
- `ETIMEDOUT` errors are no longer logged. If you need to distinguish timeouts from other forms of closed connections, examine `stream.error` from a close callback.

tornado.netutil

- When *bind_sockets* chooses a port automatically, it will now use the same port for IPv4 and IPv6.
- TLS compression is now disabled by default on Python 3.3 and higher (it is not possible to change this option in older versions).

tornado.options

- It is now possible to disable the default logging configuration by setting `options.logging` to `None` instead of the string `"none"`.

tornado.platform.asyncio

- Now works on Python 2.6.
- Now works with Trollius version 0.3.

tornado.platform.twisted

- *TwistedIOLoop* now works on Python 3.3+ (with Twisted 14.0.0+).

tornado.simple_httpclient

- `simple_httpclient` has better support for IPv6, which is now enabled by default.
- Improved default cipher suite selection (Python 2.7+).
- HTTP implementation has been unified with `tornado.httpserver` in *tornado.http1connection*
- Streaming request bodies are now supported via the `body_producer` keyword argument to *tornado.httpclient.HTTPRequest*.
- The `expect_100_continue` keyword argument to *tornado.httpclient.HTTPRequest* allows the use of the HTTP `Expect: 100-continue` feature.
- `simple_httpclient` now raises the original exception (e.g. an `IOError`) in more cases, instead of converting everything to `HTTPError`.

tornado.stack_context

- The stack context system now has less performance overhead when no stack contexts are active.

tornado.tcpclient

- New module which creates TCP connections and IOStreams, including name resolution, connecting, and SSL handshakes.

tornado.testing

- *AsyncTestCase* now attempts to detect test methods that are generators but were not run with `@gen_test` or any similar decorator (this would previously result in the test silently being skipped).
- Better stack traces are now displayed when a test times out.
- The `@gen_test` decorator now passes along `*args, **kwargs` so it can be used on functions with arguments.
- Fixed the test suite when `unittest2` is installed on Python 3.

tornado.web

- It is now possible to support streaming request bodies with the *stream_request_body* decorator and the new *RequestHandler.data_received* method.
- *RequestHandler.flush* now returns a *Future* if no callback is given.
- New exception *Finish* may be raised to finish a request without triggering error handling.
- When gzip support is enabled, all `text/*` mime types will be compressed, not just those on a whitelist.
- *Application* now implements the *HTTPMessageDelegate* interface.
- `HEAD` requests in *StaticFileHandler* no longer read the entire file.
- *StaticFileHandler* now streams response bodies to the client.
- New setting `compress_response` replaces the existing `gzip` setting; both names are accepted.
- XSRF cookies that were not generated by this module (i.e. strings without any particular formatting) are once again accepted (as long as the cookie and body/header match). This pattern was common for testing and non-browser clients but was broken by the changes in Tornado 3.2.2.

tornado.websocket

- WebSocket connections from other origin sites are now rejected by default. Browsers do not use the same-origin policy for WebSocket connections as they do for most other browser-initiated communications. This can be surprising and a security risk, so we disallow these connections on the server side by default. To accept cross-origin websocket connections, override the new method *WebSocketHandler.check_origin*.
- *WebSocketHandler.close* and *WebSocketClientConnection.close* now support `code` and `reason` arguments to send a status code and message to the other side of the connection when closing. Both classes also have `close_code` and `close_reason` attributes to receive these values when the other side closes.
- The C speedup module now builds correctly with MSVC, and can support messages larger than 2GB on 64-bit systems.
- The fallback mechanism for detecting a missing C compiler now works correctly on Mac OS X.

- Arguments to *WebSocketHandler.open* are now decoded in the same way as arguments to *RequestHandler.get* and similar methods.
- It is now allowed to override `prepare` in a *WebSocketHandler*, and this method may generate HTTP responses (error pages) in the usual way. The HTTP response methods are still not allowed once the WebSocket handshake has completed.

tornado.wsgi

- New class *WSGIAdapter* supports running a Tornado *Application* on a WSGI server in a way that is more compatible with Tornado's non-WSGI *HTTPServer*. *WSGIApplication* is deprecated in favor of using *WSGIAdapter* with a regular *Application*.
- *WSGIAdapter* now supports gzipped output.

4.9.11 What's new in Tornado 3.2.2

June 3, 2014

Security fixes

- The XSRF token is now encoded with a random mask on each request. This makes it safe to include in compressed pages without being vulnerable to the BREACH attack. This applies to most applications that use both the `xsrf_cookies` and `gzip` options (or have gzip applied by a proxy).

Backwards-compatibility notes

- If Tornado 3.2.2 is run at the same time as older versions on the same domain, there is some potential for issues with the differing cookie versions. The *Application* setting `xsrf_cookie_version=1` can be used for a transitional period to generate the older cookie format on newer servers.

Other changes

- `tornado.platform.asyncio` is now compatible with `trollius` version 0.3.

4.9.12 What's new in Tornado 3.2.1

May 5, 2014

Security fixes

- The signed-value format used by *RequestHandler.set_secure_cookie* and *RequestHandler.get_secure_cookie* has changed to be more secure. **This is a disruptive change**. The `secure_cookie` functions take new `version` parameters to support transitions between cookie formats.
- The new cookie format fixes a vulnerability that may be present in applications that use multiple cookies where the name of one cookie is a prefix of the name of another.

- To minimize disruption, cookies in the older format will be accepted by default until they expire. Applications that may be vulnerable can reject all cookies in the older format by passing `min_version=2` to *`RequestHandler.get_secure_cookie`*.
- Thanks to Joost Pol of Certified Secure for reporting this issue.

Backwards-compatibility notes

- Signed cookies issued by *`RequestHandler.set_secure_cookie`* in Tornado 3.2.1 cannot be read by older releases. If you need to run 3.2.1 in parallel with older releases, you can pass `version=1` to *`RequestHandler.set_secure_cookie`* to issue cookies that are backwards-compatible (but have a known weakness, so this option should only be used for a transitional period).

Other changes

- The C extension used to speed up the websocket module now compiles correctly on Windows with MSVC and 64-bit mode. The fallback to the pure-Python alternative now works correctly on Mac OS X machines with no C compiler installed.

4.9.13 What's new in Tornado 3.2

Jan 14, 2014

Installation

- Tornado now depends on the backports.ssl_match_hostname when running on Python 2. This will be installed automatically when using `pip` or `easy_install`
- Tornado now includes an optional C extension module, which greatly improves performance of websockets. This extension will be built automatically if a C compiler is found at install time.

New modules

- The *`tornado.platform.asyncio`* module provides integration with the `asyncio` module introduced in Python 3.4 (also available for Python 3.3 with `pip install asyncio`).

`tornado.auth`

- Added *`GoogleOAuth2Mixin`* support authentication to Google services with OAuth 2 instead of OpenID and OAuth 1.
- *`FacebookGraphMixin`* has been updated to use the current Facebook login URL, which saves a redirect.

`tornado.concurrent`

- `TracebackFuture` now accepts a `timeout` keyword argument (although it is still incorrect to use a non-zero timeout in non-blocking code).

tornado.curl_httpclient

- `tornado.curl_httpclient` now works on Python 3 with the soon-to-be-released pycurl 7.19.3, which will officially support Python 3 for the first time. Note that there are some unofficial Python 3 ports of pycurl (Ubuntu has included one for its past several releases); these are not supported for use with Tornado.

tornado.escape

- *xhtml_escape* now escapes apostrophes as well.
- *tornado.escape.utf8*, *to_unicode*, and *native_str* now raise `TypeError` instead of `AssertionError` when given an invalid value.

tornado.gen

- Coroutines may now yield dicts in addition to lists to wait for multiple tasks in parallel.
- Improved performance of *tornado.gen* when yielding a *Future* that is already done.

tornado.httpclient

- *tornado.httpclient.HTTPRequest* now uses property setters so that setting attributes after construction applies the same conversions as `__init__` (e.g. converting the body attribute to bytes).

tornado.httpserver

- Malformed `x-www-form-urlencoded` request bodies will now log a warning and continue instead of causing the request to fail (similar to the existing handling of malformed `multipart/form-data` bodies. This is done mainly because some libraries send this content type by default even when the data is not form-encoded.
- Fix some error messages for unix sockets (and other non-IP sockets)

tornado.ioloop

- *IOLoop* now uses *handle_callback_exception* consistently for error logging.
- *IOLoop* now frees callback objects earlier, reducing memory usage while idle.
- *IOLoop* will no longer call `logging.basicConfig` if there is a handler defined for the root logger or for the `tornado` or `tornado.application` loggers (previously it only looked at the root logger).

tornado.iostream

- *IOStream* now recognizes `ECONNABORTED` error codes in more places (which was mainly an issue on Windows).
- *IOStream* now frees memory earlier if a connection is closed while there is data in the write buffer.
- *PipeIOStream* now handles `EAGAIN` error codes correctly.
- *SSLIOStream* now initiates the SSL handshake automatically without waiting for the application to try and read or write to the connection.

- Swallow a spurious exception from `set_nodelay` when a connection has been reset.

tornado.locale

- *Locale.format_date* no longer forces the use of absolute dates in Russian.

tornado.log

- Fix an error from *tornado.log.enable_pretty_logging* when `sys.stderr` does not have an `isatty` method.
- *tornado.log.LogFormatter* now accepts keyword arguments `fmt` and `datefmt`.

tornado.netutil

- *is_valid_ip* (and therefore `HTTPRequest.remote_ip`) now rejects empty strings.
- Synchronously using *ThreadedResolver* at import time to resolve a unicode hostname no longer deadlocks.

tornado.platform.twisted

- *TwistedResolver* now has better error handling.

tornado.process

- *Subprocess* no longer leaks file descriptors if `subprocess.Popen` fails.

tornado.simple_httpclient

- `simple_httpclient` now applies the `connect_timeout` to requests that are queued and have not yet started.
- On Python 2.6, `simple_httpclient` now uses TLSv1 instead of SSLv3.
- `simple_httpclient` now enforces the connect timeout during DNS resolution.
- The embedded `ca-certificates.crt` file has been updated with the current Mozilla CA list.

tornado.web

- *StaticFileHandler* no longer fails if the client requests a `Range` that is larger than the entire file (Facebook has a crawler that does this).
- *RequestHandler.on_connection_close* now works correctly on subsequent requests of a keep-alive connection.
- New application setting `default_handler_class` can be used to easily set up custom 404 pages.
- New application settings `autoreload`, `compiled_template_cache`, `static_hash_cache`, and `serve_traceback` can be used to control individual aspects of debug mode.

- New methods *RequestHandler.get_query_argument* and *RequestHandler.get_body_argument* and new attributes `HTTPRequest.query_arguments` and `HTTPRequest.body_arguments` allow access to arguments without intermingling those from the query string with those from the request body.
- *RequestHandler.decode_argument* and related methods now raise an `HTTPError(400)` instead of `UnicodeDecodeError` when the argument could not be decoded.
- *RequestHandler.clear_all_cookies* now accepts `domain` and `path` arguments, just like *clear_cookie*.
- It is now possible to specify handlers by name when using the *URLSpec* class.
- *Application* now accepts 4-tuples to specify the `name` parameter (which previously required constructing a *URLSpec* object instead of a tuple).
- Fixed an incorrect error message when handler methods return a value other than None or a Future.
- Exceptions will no longer be logged twice when using both `@asynchronous` and `@gen.coroutine`

tornado.websocket

- *WebSocketHandler.write_message* now raises *WebSocketClosedError* instead of `AttributeError` when the connection has been closed.
- *websocket_connect* now accepts preconstructed `HTTPRequest` objects.
- Fix a bug with *WebSocketHandler* when used with some proxies that unconditionally modify the `Connection` header.
- *websocket_connect* now returns an error immediately for refused connections instead of waiting for the timeout.
- *WebSocketClientConnection* now has a `close` method.

tornado.wsgi

- *WSGIContainer* now calls the iterable's `close()` method even if an error is raised, in compliance with the spec.

4.9.14 What's new in Tornado 3.1.1

Sep 1, 2013

- *StaticFileHandler* no longer fails if the client requests a `Range` that is larger than the entire file (Facebook has a crawler that does this).
- *RequestHandler.on_connection_close* now works correctly on subsequent requests of a keep-alive connection.

4.9.15 What's new in Tornado 3.1

Jun 15, 2013

Multiple modules

- Many reference cycles have been broken up throughout the package, allowing for more efficient garbage collection on CPython.
- Silenced some log messages when connections are opened and immediately closed (i.e. port scans), or other situations related to closed connections.
- Various small speedups: *HTTPHeaders* case normalization, *UIModule* proxy objects, precompile some regexes.

tornado.auth

- *OAuthMixin* always sends `oauth_version=1.0` in its request as required by the spec.
- *FacebookGraphMixin* now uses `self._FACEBOOK_BASE_URL` in *facebook_request* to allow the base url to be overridden.
- The `authenticate_redirect` and `authorize_redirect` methods in the *tornado.auth* mixin classes all now return Futures. These methods are asynchronous in *OAuthMixin* and derived classes, although they do not take a callback. The *Future* these methods return must be yielded if they are called from a function decorated with *gen.coroutine* (but not *gen.engine*).
- *TwitterMixin* now uses `/account/verify_credentials` to get information about the logged-in user, which is more robust against changing screen names.
- The `demos` directory (in the source distribution) has a new `twitter` demo using *TwitterMixin*.

tornado.escape

- *url_escape* and *url_unescape* have a new `plus` argument (defaulting to True for consistency with the previous behavior) which specifies whether they work like `urllib.parse.unquote` or `urllib.parse.unquote_plus`.

tornado.gen

- Fixed a potential memory leak with long chains of *tornado.gen* coroutines.

tornado.httpclient

- *tornado.httpclient.HTTPRequest* takes a new argument `auth_mode`, which can be either `basic` or `digest`. Digest authentication is only supported with `tornado.curl_httpclient`.
- `tornado.curl_httpclient` no longer goes into an infinite loop when pycurl returns a negative timeout.
- `curl_httpclient` now supports the `PATCH` and `OPTIONS` methods without the use of `allow_nonstandard_methods=True`.
- Worked around a class of bugs in libcurl that would result in errors from *IOLoop.update_handler* in various scenarios including digest authentication and socks proxies.

- The TCP_NODELAY flag is now set when appropriate in simple_httpclient.
- simple_httpclient no longer logs exceptions, since those exceptions are made available to the caller as HTTPResponse.error.

tornado.httpserver

- *tornado.httpserver.HTTPServer* handles malformed HTTP headers more gracefully.
- *HTTPServer* now supports lists of IPs in X-Forwarded-For (it chooses the last, i.e. nearest one).
- Memory is now reclaimed promptly on CPython when an HTTP request fails because it exceeded the maximum upload size.
- The TCP_NODELAY flag is now set when appropriate in *HTTPServer*.
- The *HTTPServer* no_keep_alive option is now respected with HTTP 1.0 connections that explicitly pass Connection: keep-alive.
- The Connection: keep-alive check for HTTP 1.0 connections is now case-insensitive.
- The str and repr of tornado.httpserver.HTTPRequest no longer include the request body, reducing log spam on errors (and potential exposure/retention of private data).

tornado.httputil

- The cache used in *HTTPHeaders* will no longer grow without bound.

tornado.ioloop

- Some *IOLoop* implementations (such as pyzmq) accept objects other than integer file descriptors; these objects will now have their .close() method called when the IOLoop` is closed with ``all_fds=True.
- The stub handles left behind by *IOLoop.remove_timeout* will now get cleaned up instead of waiting to expire.

tornado.iostream

- Fixed a bug in *BaseIOStream.read_until_close* that would sometimes cause data to be passed to the final callback instead of the streaming callback.
- The *IOStream* close callback is now run more reliably if there is an exception in _try_inline_read.
- New method *BaseIOStream.set_nodelay* can be used to set the TCP_NODELAY flag.
- Fixed a case where errors in SSLIOStream.connect (and SimpleAsyncHTTPClient) were not being reported correctly.

tornado.locale

- *Locale.format_date* now works on Python 3.

tornado.netutil

- The default *Resolver* implementation now works on Solaris.
- *Resolver* now has a *close* method.
- Fixed a potential CPU DoS when `tornado.netutil.ssl_match_hostname` is used on certificates with an abusive wildcard pattern.
- All instances of *ThreadedResolver* now share a single thread pool, whose size is set by the first one to be created (or the static `Resolver.configure` method).
- *ExecutorResolver* is now documented for public use.
- *bind_sockets* now works in configurations with incomplete IPv6 support.

tornado.options

- *tornado.options.define* with `multiple=True` now works on Python 3.
- *tornado.options.options* and other *OptionParser* instances support some new dict-like methods: *items()*, iteration over keys, and (read-only) access to options with square braket syntax. *OptionParser.group_dict* returns all options with a given group name, and *OptionParser.as_dict* returns all options.

tornado.process

- *tornado.process.Subprocess* no longer leaks file descriptors into the child process, which fixes a problem in which the child could not detect that the parent process had closed its stdin pipe.
- *Subprocess.set_exit_callback* now works for subprocesses created without an explicit `io_loop` parameter.

tornado.stack_context

- *tornado.stack_context* has been rewritten and is now much faster.
- New function *run_with_stack_context* facilitates the use of stack contexts with coroutines.

tornado.tcpserver

- The constructors of *TCPServer* and *HTTPServer* now take a `max_buffer_size` keyword argument.

tornado.template

- Some internal names used by the template system have been changed; now all "reserved" names in templates start with `_tt_`.

tornado.testing

- *tornado.testing.AsyncTestCase.wait* now raises the correct exception when it has been modified by *tornado.stack_context*.
- *tornado.testing.gen_test* can now be called as `@gen_test(timeout=60)` to give some tests a longer timeout than others.
- The environment variable `ASYNC_TEST_TIMEOUT` can now be set to override the default timeout for *AsyncTestCase.wait* and *gen_test*.
- *bind_unused_port* now passes `None` instead of `0` as the port to `getaddrinfo`, which works better with some unusual network configurations.

tornado.util

- *tornado.util.import_object* now works with top-level module names that do not contain a dot.
- *tornado.util.import_object* now consistently raises `ImportError` instead of `AttributeError` when it fails.

tornado.web

- The `handlers` list passed to the *tornado.web.Application* constructor and *add_handlers* methods can now contain lists in addition to tuples and *URLSpec* objects.
- *tornado.web.StaticFileHandler* now works on Windows when the client passes an `If-Modified-Since` timestamp before 1970.
- New method *RequestHandler.log_exception* can be overridden to customize the logging behavior when an exception is uncaught. Most apps that currently override `_handle_request_exception` can now use a combination of *RequestHandler.log_exception* and *write_error*.
- *RequestHandler.get_argument* now raises *MissingArgumentError* (a subclass of *tornado.web.HTTPError*, which is what it raised previously) if the argument cannot be found.
- *Application.reverse_url* now uses *url_escape* with `plus=False`, i.e. spaces are encoded as `%20` instead of +.
- Arguments extracted from the url path are now decoded with *url_unescape* with `plus=False`, so plus signs are left as-is instead of being turned into spaces.
- *RequestHandler.send_error* will now only be called once per request, even if multiple exceptions are caught by the stack context.
- The *tornado.web.asynchronous* decorator is no longer necessary for methods that return a *Future* (i.e. those that use the *gen.coroutine* or *return_future* decorators)
- *RequestHandler.prepare* may now be asynchronous if it returns a *Future*. The *asynchronous* decorator is not used with `prepare`; one of the *Future*-related decorators should be used instead.
- `RequestHandler.current_user` may now be assigned to normally.
- *RequestHandler.redirect* no longer silently strips control characters and whitespace. It is now an error to pass control characters, newlines or tabs.
- *StaticFileHandler* has been reorganized internally and now has additional extension points that can be overridden in subclasses.

- *StaticFileHandler* now supports HTTP `Range` requests. *StaticFileHandler* is still not suitable for files too large to comfortably fit in memory, but `Range` support is necessary in some browsers to enable seeking of HTML5 audio and video.
- *StaticFileHandler* now uses longer hashes by default, and uses the same hashes for `Etag` as it does for versioned urls.
- *StaticFileHandler.make_static_url* and *RequestHandler.static_url* now have an additional keyword argument `include_version` to suppress the url versioning.
- *StaticFileHandler* now reads its file in chunks, which will reduce memory fragmentation.
- Fixed a problem with the `Date` header and cookie expiration dates when the system locale is set to a non-english configuration.

tornado.websocket

- *WebSocketHandler* now catches *StreamClosedError* and runs *on_close* immediately instead of logging a stack trace.
- New method *WebSocketHandler.set_nodelay* can be used to set the `TCP_NODELAY` flag.

tornado.wsgi

- Fixed an exception in *WSGIContainer* when the connection is closed while output is being written.

4.9.16 What's new in Tornado 3.0.2

Jun 2, 2013

- *tornado.auth.TwitterMixin* now defaults to version 1.1 of the Twitter API, instead of version 1.0 which is being discontinued on June 11. It also now uses HTTPS when talking to Twitter.
- Fixed a potential memory leak with a long chain of *gen.coroutine* or *gen.engine* functions.

4.9.17 What's new in Tornado 3.0.1

Apr 8, 2013

- The interface of *tornado.auth.FacebookGraphMixin* is now consistent with its documentation and the rest of the module. The `get_authenticated_user` and `facebook_request` methods return a `Future` and the `callback` argument is optional.
- The *tornado.testing.gen_test* decorator will no longer be recognized as a (broken) test by `nose`.
- Work around a bug in Ubuntu 13.04 betas involving an incomplete backport of the `ssl.match_hostname` function.
- *tornado.websocket.websocket_connect* now fails cleanly when it attempts to connect to a non-websocket url.
- *tornado.testing.LogTrapTestCase* once again works with byte strings on Python 2.
- The `request` attribute of *tornado.httpclient.HTTPResponse* is now always an *HTTPRequest*, never a `_RequestProxy`.

- Exceptions raised by the *tornado.gen* module now have better messages when tuples are used as callback keys.

4.9.18 What's new in Tornado 3.0

Mar 29, 2013

Highlights

- The `callback` argument to many asynchronous methods is now optional, and these methods return a *Future*. The *tornado.gen* module now understands `Futures`, and these methods can be used directly without a *gen.Task* wrapper.
- New function *IOLoop.current* returns the *IOLoop* that is running on the current thread (as opposed to *IOLoop.instance*, which returns a specific thread's (usually the main thread's) IOLoop.
- New class *tornado.netutil.Resolver* provides an asynchronous interface to DNS resolution. The default implementation is still blocking, but non-blocking implementations are available using one of three optional dependencies: *ThreadedResolver* using the `concurrent.futures` thread pool, `tornado.platform.caresresolver.CaresResolver` using the `pycares` library, or `tornado.platform.twisted.TwistedResolver` using `twisted`
- Tornado's logging is now less noisy, and it no longer goes directly to the root logger, allowing for finer-grained configuration.
- New class *tornado.process.Subprocess* wraps `subprocess.Popen` with *PipeIOStream* access to the child's file descriptors.
- *IOLoop* now has a static *configure* method like the one on *AsyncHTTPClient*, which can be used to select an *IOLoop* implementation other than the default.
- *IOLoop* can now optionally use a monotonic clock if available (see below for more details).

Backwards-incompatible changes

- Python 2.5 is no longer supported. Python 3 is now supported in a single codebase instead of using `2to3`
- The `tornado.database` module has been removed. It is now available as a separate package, torndb
- Functions that take an `io_loop` parameter now default to *IOLoop.current()* instead of *IOLoop.instance()*.
- Empty HTTP request arguments are no longer ignored. This applies to `HTTPRequest.arguments` and `RequestHandler.get_argument[s]` in WSGI and non-WSGI modes.
- On Python 3, *tornado.escape.json_encode* no longer accepts byte strings.
- On Python 3, the `get_authenticated_user` methods in *tornado.auth* now return character strings instead of byte strings.
- `tornado.netutil.TCPServer` has moved to its own module, *tornado.tcpserver*.
- The Tornado test suite now requires `unittest2` when run on Python 2.6.
- *tornado.options.options* is no longer a subclass of `dict`; attribute-style access is now required.

Detailed changes by module

Multiple modules

- Tornado no longer logs to the root logger. Details on the new logging scheme can be found under the *tornado.log* module. Note that in some cases this will require that you add an explicit logging configuration in order to see any output (perhaps just calling `logging.basicConfig()`), although both *IOLoop.start()* and *tornado.options.parse_command_line* will do this for you.
- On python 3.2+, methods that take an `ssl_options` argument (on *SSLIOStream*, *TCPServer*, and *HTTPServer*) now accept either a dictionary of options or an `ssl.SSLContext` object.
- New optional dependency on `concurrent.futures` to provide better support for working with threads. `concurrent.futures` is in the standard library for Python 3.2+, and can be installed on older versions with `pip install futures`.

tornado.autoreload

- *tornado.autoreload* is now more reliable when there are errors at import time.
- Calling *tornado.autoreload.start* (or creating an *Application* with `debug=True`) twice on the same *IOLoop* now does nothing (instead of creating multiple periodic callbacks). Starting autoreload on more than one *IOLoop* in the same process now logs a warning.
- Scripts run by autoreload no longer inherit `__future__` imports used by Tornado.

tornado.auth

- On Python 3, the `get_authenticated_user` method family now returns character strings instead of byte strings.
- Asynchronous methods defined in *tornado.auth* now return a *Future*, and their `callback` argument is optional. The `Future` interface is preferred as it offers better error handling (the previous interface just logged a warning and returned None).
- The *tornado.auth* mixin classes now define a method `get_auth_http_client`, which can be overridden to use a non-default *AsyncHTTPClient* instance (e.g. to use a different *IOLoop*)
- Subclasses of *OAuthMixin* are encouraged to override *OAuthMixin._oauth_get_user_future* instead of `_oauth_get_user`, although both methods are still supported.

tornado.concurrent

- New module *tornado.concurrent* contains code to support working with `concurrent.futures`, or to emulate future-based interface when that module is not available.

tornado.curl_httpclient

- Preliminary support for `tornado.curl_httpclient` on Python 3. The latest official release of pycurl only supports Python 2, but Ubuntu has a port available in 12.10 (`apt-get install python3-pycurl`). This port currently has bugs that prevent it from handling arbitrary binary data but it should work for textual (utf8) resources.
- Fix a crash with libcurl 7.29.0 if a curl object is created and closed without being used.

tornado.escape

- On Python 3, *json_encode* no longer accepts byte strings. This mirrors the behavior of the underlying json module. Python 2 behavior is unchanged but should be faster.

tornado.gen

- New decorator `@gen.coroutine` is available as an alternative to `@gen.engine`. It automatically returns a *Future*, and within the function instead of calling a callback you return a value with `raise gen.Return(value)` (or simply `return value` in Python 3.3).
- Generators may now yield *Future* objects.
- Callbacks produced by *gen.Callback* and *gen.Task* are now automatically stack-context-wrapped, to minimize the risk of context leaks when used with asynchronous functions that don't do their own wrapping.
- Fixed a memory leak involving generators, *RequestHandler.flush*, and clients closing connections while output is being written.
- Yielding a large list no longer has quadratic performance.

tornado.httpclient

- *AsyncHTTPClient.fetch* now returns a *Future* and its callback argument is optional. When the future interface is used, any error will be raised automatically, as if *HTTPResponse.rethrow* was called.
- *AsyncHTTPClient.configure* and all *AsyncHTTPClient* constructors now take a `defaults` keyword argument. This argument should be a dictionary, and its values will be used in place of corresponding attributes of *HTTPRequest* that are not set.
- All unset attributes of *tornado.httpclient.HTTPRequest* are now `None`. The default values of some attributes (`connect_timeout`, `request_timeout`, `follow_redirects`, `max_redirects`, `use_gzip`, `proxy_password`, `allow_nonstandard_methods`, and `validate_cert` have been moved from *HTTPRequest* to the client implementations.
- The `max_clients` argument to *AsyncHTTPClient* is now a keyword-only argument.
- Keyword arguments to *AsyncHTTPClient.configure* are no longer used when instantiating an implementation subclass directly.
- Secondary *AsyncHTTPClient* callbacks (`streaming_callback`, `header_callback`, and `prepare_curl_callback`) now respect *StackContext*.

tornado.httpserver

- *HTTPServer* no longer logs an error when it is unable to read a second request from an HTTP 1.1 keep-alive connection.
- *HTTPServer* now takes a `protocol` keyword argument which can be set to `https` if the server is behind an SSL-decoding proxy that does not set any supported X-headers.
- `tornado.httpserver.HTTPConnection` now has a `set_close_callback` method that should be used instead of reaching into its `stream` attribute.
- Empty HTTP request arguments are no longer ignored. This applies to `HTTPRequest.arguments` and `RequestHandler.get_argument[s]` in WSGI and non-WSGI modes.

tornado.ioloop

- New function *IOLoop.current* returns the IOLoop that is running on the current thread (as opposed to *IOLoop.instance*, which returns a specific thread's (usually the main thread's) IOLoop).
- New method *IOLoop.add_future* to run a callback on the IOLoop when an asynchronous *Future* finishes.
- *IOLoop* now has a static *configure* method like the one on *AsyncHTTPClient*, which can be used to select an *IOLoop* implementation other than the default.
- The *IOLoop* poller implementations (select, epoll, kqueue) are now available as distinct subclasses of *IOLoop*. Instantiating *IOLoop* will continue to automatically choose the best available implementation.
- The *IOLoop* constructor has a new keyword argument time_func, which can be used to set the time function used when scheduling callbacks. This is most useful with the time.monotonic function, introduced in Python 3.3 and backported to older versions via the monotime module. Using a monotonic clock here avoids problems when the system clock is changed.
- New function *IOLoop.time* returns the current time according to the IOLoop. To use the new monotonic clock functionality, all calls to *IOLoop.add_timeout* must be either pass a datetime.timedelta or a time relative to *IOLoop.time*, not time.time. (time.time will continue to work only as long as the IOLoop's time_func argument is not used).
- New convenience method *IOLoop.run_sync* can be used to start an IOLoop just long enough to run a single coroutine.
- New method *IOLoop.add_callback_from_signal* is safe to use in a signal handler (the regular *add_callback* method may deadlock).
- *IOLoop* now uses signal.set_wakeup_fd where available (Python 2.6+ on Unix) to avoid a race condition that could result in Python signal handlers being delayed.
- Method IOLoop.running() has been removed.
- *IOLoop* has been refactored to better support subclassing.
- *IOLoop.add_callback* and *add_callback_from_signal* now take *args, **kwargs to pass along to the callback.

tornado.iostream

- *IOStream.connect* now has an optional server_hostname argument which will be used for SSL certificate validation when applicable. Additionally, when supported (on Python 3.2+), this hostname will be sent via SNI (and this is supported by tornado.simple_httpclient)
- Much of *IOStream* has been refactored into a separate class *BaseIOStream*.
- New class *tornado.iostream.PipeIOStream* provides the IOStream interface on pipe file descriptors.
- *IOStream* now raises a new exception tornado.iostream.StreamClosedError when you attempt to read or write after the stream has been closed (by either side).
- *IOStream* now simply closes the connection when it gets an ECONNRESET error, rather than logging it as an error.
- IOStream.error no longer picks up unrelated exceptions.
- *BaseIOStream.close* now has an exc_info argument (similar to the one used in the logging module) that can be used to set the stream's error attribute when closing it.
- *BaseIOStream.read_until_close* now works correctly when it is called while there is buffered data.
- Fixed a major performance regression when run on PyPy (introduced in Tornado 2.3).

tornado.log

- New module containing *enable_pretty_logging* and *LogFormatter*, moved from the options module.
- *LogFormatter* now handles non-ascii data in messages and tracebacks better.

tornado.netutil

- New class *tornado.netutil.Resolver* provides an asynchronous interface to DNS resolution. The default implementation is still blocking, but non-blocking implementations are available using one of three optional dependencies: *ThreadedResolver* using the `concurrent.futures` thread pool, *tornado.platform.caresresolver.CaresResolver* using the `pycares` library, or *tornado.platform.twisted.TwistedResolver* using `twisted`
- New function *tornado.netutil.is_valid_ip* returns true if a given string is a valid IP (v4 or v6) address.
- *tornado.netutil.bind_sockets* has a new `flags` argument that can be used to pass additional flags to `getaddrinfo`.
- *tornado.netutil.bind_sockets* no longer sets `AI_ADDRCONFIG`; this will cause it to bind to both ipv4 and ipv6 more often than before.
- *tornado.netutil.bind_sockets* now works when Python was compiled with `--disable-ipv6` but IPv6 DNS resolution is available on the system.
- `tornado.netutil.TCPServer` has moved to its own module, *tornado.tcpserver*.

tornado.options

- The class underlying the functions in *tornado.options* is now public (*tornado.options.OptionParser*). This can be used to create multiple independent option sets, such as for subcommands.
- *tornado.options.parse_config_file* now configures logging automatically by default, in the same way that *parse_command_line* does.
- New function *tornado.options.add_parse_callback* schedules a callback to be run after the command line or config file has been parsed. The keyword argument `final=False` can be used on either parsing function to supress these callbacks.
- *tornado.options.define* now takes a `callback` argument. This callback will be run with the new value whenever the option is changed. This is especially useful for options that set other options, such as by reading from a config file.
- *tornado.options.parse_command_line* `--help` output now goes to `stderr` rather than `stdout`.
- *tornado.options.options* is no longer a subclass of `dict`; attribute-style access is now required.
- *tornado.options.options* (and *OptionParser* instances generally) now have a *mockable()* method that returns a wrapper object compatible with `mock.patch`.
- Function `tornado.options.enable_pretty_logging` has been moved to the *tornado.log* module.

tornado.platform.caresresolver

- New module containing an asynchronous implementation of the *Resolver* interface, using the `pycares` library.

tornado.platform.twisted

- New class *tornado.platform.twisted.TwistedIOLoop* allows Tornado code to be run on the Twisted reactor (as opposed to the existing *TornadoReactor*, which bridges the gap in the other direction).
- New class *tornado.platform.twisted.TwistedResolver* is an asynchronous implementation of the *Resolver* interface.

tornado.process

- New class *tornado.process.Subprocess* wraps `subprocess.Popen` with *PipeIOStream* access to the child's file descriptors.

tornado.simple_httpclient

- `SimpleAsyncHTTPClient` now takes a `resolver` keyword argument (which may be passed to either the constructor or *configure*), to allow it to use the new non-blocking *tornado.netutil.Resolver*.
- When following redirects, `SimpleAsyncHTTPClient` now treats a 302 response code the same as a 303. This is contrary to the HTTP spec but consistent with all browsers and other major HTTP clients (including `CurlAsyncHTTPClient`).
- The behavior of `header_callback` with `SimpleAsyncHTTPClient` has changed and is now the same as that of `CurlAsyncHTTPClient`. The header callback now receives the first line of the response (e.g. `HTTP/1.0 200 OK`) and the final empty line.
- `tornado.simple_httpclient` now accepts responses with a 304 status code that include a `Content-Length` header.
- Fixed a bug in which `SimpleAsyncHTTPClient` callbacks were being run in the client's `stack_context`.

tornado.stack_context

- *stack_context.wrap* now runs the wrapped callback in a more consistent environment by recreating contexts even if they already exist on the stack.
- Fixed a bug in which stack contexts could leak from one callback chain to another.
- Yield statements inside a `with` statement can cause stack contexts to become inconsistent; an exception will now be raised when this case is detected.

tornado.template

- Errors while rendering templates no longer log the generated code, since the enhanced stack traces (from version 2.1) should make this unnecessary.
- The `{% apply %}` directive now works properly with functions that return both unicode strings and byte strings (previously only byte strings were supported).
- Code in templates is no longer affected by Tornado's `__future__` imports (which previously included `absolute_import` and `division`).

tornado.testing

- New function *tornado.testing.bind_unused_port* both chooses a port and binds a socket to it, so there is no risk of another process using the same port. `get_unused_port` is now deprecated.

- New decorator *tornado.testing.gen_test* can be used to allow for yielding *tornado.gen* objects in tests, as an alternative to the `stop` and `wait` methods of *AsyncTestCase*.
- *tornado.testing.AsyncTestCase* and friends now extend `unittest2.TestCase` when it is available (and continue to use the standard `unittest` module when `unittest2` is not available)
- *tornado.testing.ExpectLog* can be used as a finer-grained alternative to *tornado.testing.LogTrapTestCase*
- The command-line interface to *tornado.testing.main* now supports additional arguments from the underlying `unittest` module: `verbose`, `quiet`, `failfast`, `catch`, `buffer`.
- The deprecated `--autoreload` option of *tornado.testing.main* has been removed. Use `python -m tornado.autoreload` as a prefix command instead.
- The `--httpclient` option of *tornado.testing.main* has been moved to `tornado.test.runtests` so as not to pollute the application option namespace. The *tornado.options* module's new callback support now makes it easy to add options from a wrapper script instead of putting all possible options in *tornado.testing.main*.
- *AsyncHTTPTestCase* no longer calls *AsyncHTTPClient.close* for tests that use the singleton *IOLoop.instance*.
- *LogTrapTestCase* no longer fails when run in unknown logging configurations. This allows tests to be run under nose, which does its own log buffering (*LogTrapTestCase* doesn't do anything useful in this case, but at least it doesn't break things any more).

tornado.util

- `tornado.util.b` (which was only intended for internal use) is gone.

tornado.web

- *RequestHandler.set_header* now overwrites previous header values case-insensitively.
- *tornado.web.RequestHandler* has new attributes `path_args` and `path_kwargs`, which contain the positional and keyword arguments that are passed to the `get`/`post`/etc method. These attributes are set before those methods are called, so they are available during `prepare()`
- *tornado.web.ErrorHandler* no longer requires XSRF tokens on `POST` requests, so posts to an unknown url will always return 404 instead of complaining about XSRF tokens.
- Several methods related to HTTP status codes now take a `reason` keyword argument to specify an alternate "reason" string (i.e. the "Not Found" in "HTTP/1.1 404 Not Found"). It is now possible to set status codes other than those defined in the spec, as long as a reason string is given.
- The `Date` HTTP header is now set by default on all responses.
- `Etag`/`If-None-Match` requests now work with *StaticFileHandler*.
- *StaticFileHandler* no longer sets `Cache-Control: public` unnecessarily.
- When gzip is enabled in a *tornado.web.Application*, appropriate `Vary: Accept-Encoding` headers are now sent.
- It is no longer necessary to pass all handlers for a host in a single *Application.add_handlers* call. Now the request will be matched against the handlers for any `host_pattern` that includes the request's `Host` header.

tornado.websocket

- Client-side WebSocket support is now available: *tornado.websocket.websocket_connect*
- *WebSocketHandler* has new methods *ping* and *on_pong* to send pings to the browser (not supported on the draft76 protocol)

4.9.19 What's new in Tornado 2.4.1

Nov 24, 2012

Bug fixes

- Fixed a memory leak in *tornado.stack_context* that was especially likely with long-running @gen.engine functions.
- *tornado.auth.TwitterMixin* now works on Python 3.
- Fixed a bug in which IOStream.read_until_close with a streaming callback would sometimes pass the last chunk of data to the final callback instead of the streaming callback.

4.9.20 What's new in Tornado 2.4

Sep 4, 2012

General

- Fixed Python 3 bugs in *tornado.auth*, *tornado.locale*, and *tornado.wsgi*.

HTTP clients

- Removed max_simultaneous_connections argument from *tornado.httpclient* (both implementations). This argument hasn't been useful for some time (if you were using it you probably want max_clients instead)
- tornado.simple_httpclient now accepts and ignores HTTP 1xx status responses.

tornado.ioloop and tornado.iostream

- Fixed a bug introduced in 2.3 that would cause *IOStream* close callbacks to not run if there were pending reads.
- Improved error handling in *SSLIOStream* and SSL-enabled *TCPServer*.
- SSLIOStream.get_ssl_certificate now has a binary_form argument which is passed to SSLSocket.getpeercert.
- SSLIOStream.write can now be called while the connection is in progress, same as non-SSL *IOStream* (but be careful not to send sensitive data until the connection has completed and the certificate has been verified).
- *IOLoop.add_handler* cannot be called more than once with the same file descriptor. This was always true for epoll, but now the other implementations enforce it too.
- On Windows, *TCPServer* uses SO_EXCLUSIVEADDRUSER instead of SO_REUSEADDR.

tornado.template

- `{% break %}` and `{% continue %}` can now be used looping constructs in templates.
- It is no longer an error for an if/else/for/etc block in a template to have an empty body.

tornado.testing

- New class *tornado.testing.AsyncHTTPSTestCase* is like *AsyncHTTPTestCase*. but enables SSL for the testing server (by default using a self-signed testing certificate).
- *tornado.testing.main* now accepts additional keyword arguments and forwards them to `unittest.main`.

tornado.web

- New method *RequestHandler.get_template_namespace* can be overridden to add additional variables without modifying keyword arguments to `render_string`.
- *RequestHandler.add_header* now works with *WSGIApplication*.
- *RequestHandler.get_secure_cookie* now handles a potential error case.
- `RequestHandler.__init__` now calls `super().__init__` to ensure that all constructors are called when multiple inheritance is used.
- Docs have been updated with a description of all available *Application settings*

Other modules

- *OAuthMixin* now accepts `"oob"` as a `callback_uri`.
- *OpenIdMixin* now also returns the `claimed_id` field for the user.
- *tornado.platform.twisted* shutdown sequence is now more compatible.
- The logging configuration used in *tornado.options* is now more tolerant of non-ascii byte strings.

4.9.21 What's new in Tornado 2.3

May 31, 2012

HTTP clients

- *tornado.httpclient.HTTPClient* now supports the same constructor keyword arguments as *AsyncHTTPClient*.
- The `max_clients` keyword argument to *AsyncHTTPClient.configure* now works.
- `tornado.simple_httpclient` now supports the `OPTIONS` and `PATCH` HTTP methods.
- `tornado.simple_httpclient` is better about closing its sockets instead of leaving them for garbage collection.
- `tornado.simple_httpclient` correctly verifies SSL certificates for URLs containing IPv6 literals (This bug affected Python 2.5 and 2.6).

- `tornado.simple_httpclient` no longer includes basic auth credentials in the `Host` header when those credentials are extracted from the URL.
- `tornado.simple_httpclient` no longer modifies the caller-supplied header dictionary, which caused problems when following redirects.
- `tornado.curl_httpclient` now supports client SSL certificates (using the same `client_cert` and `client_key` arguments as `tornado.simple_httpclient`)

HTTP Server

- *`HTTPServer`* now works correctly with paths starting with `//`
- `HTTPHeaders.copy` (inherited from `dict.copy`) now works correctly.
- `HTTPConnection.address` is now always the socket address, even for non-IP sockets. `HTTPRequest.remote_ip` is still always an IP-style address (fake data is used for non-IP sockets)
- Extra data at the end of multipart form bodies is now ignored, which fixes a compatibility problem with an iOS HTTP client library.

`IOLoop` and `IOStream`

- *`IOStream`* now has an `error` attribute that can be used to determine why a socket was closed.
- `tornado.iostream.IOStream.read_until` and `read_until_regex` are much faster with large input.
- `IOStream.write` performs better when given very large strings.
- *`IOLoop.instance()`* is now thread-safe.

`tornado.options`

- *`tornado.options`* options with `multiple=True` that are set more than once now overwrite rather than append. This makes it possible to override values set in `parse_config_file` with `parse_command_line`.
- *`tornado.options`* `--help` output is now prettier.
- *`tornado.options.options`* now supports attribute assignment.

`tornado.template`

- Template files containing non-ASCII (utf8) characters now work on Python 3 regardless of the locale environment variables.
- Templates now support `else` clauses in `try`/`except`/`finally`/`else` blocks.

`tornado.web`

- *`tornado.web.RequestHandler`* now supports the `PATCH` HTTP method. Note that this means any existing methods named `patch` in `RequestHandler` subclasses will need to be renamed.

- *tornado.web.addslash* and removeslash decorators now send permanent redirects (301) instead of temporary (302).
- *RequestHandler.flush* now invokes its callback whether there was any data to flush or not.
- Repeated calls to *RequestHandler.set_cookie* with the same name now overwrite the previous cookie instead of producing additional copies.
- tornado.web.OutputTransform.transform_first_chunk now takes and returns a status code in addition to the headers and chunk. This is a backwards-incompatible change to an interface that was never technically private, but was not included in the documentation and does not appear to have been used outside Tornado itself.
- Fixed a bug on python versions before 2.6.5 when *URLSpec* regexes are constructed from unicode strings and keyword arguments are extracted.
- The reverse_url function in the template namespace now comes from the *RequestHandler* rather than the *Application*. (Unless overridden, *RequestHandler.reverse_url* is just an alias for the *Application* method).
- The Etag header is now returned on 304 responses to an If-None-Match request, improving compatibility with some caches.
- *tornado.web* will no longer produce responses with status code 304 that also have entity headers such as Content-Length.

Other modules

- *tornado.auth.FacebookGraphMixin* no longer sends post_args redundantly in the url.
- The extra_params argument to *tornado.escape.linkify* may now be a callable, to allow parameters to be chosen separately for each link.
- *tornado.gen* no longer leaks StackContexts when a @gen.engine wrapped function is called repeatedly.
- *tornado.locale.get_supported_locales* no longer takes a meaningless cls argument.
- *StackContext* instances now have a deactivation callback that can be used to prevent further propagation.
- *tornado.testing.AsyncTestCase.wait* now resets its timeout on each call.
- *tornado.wsgi.WSGIApplication* now parses arguments correctly on Python 3.
- Exception handling on Python 3 has been improved; previously some exceptions such as UnicodeDecodeError would generate TypeErrors

4.9.22 What's new in Tornado 2.2.1

Apr 23, 2012

Security fixes

- *tornado.web.RequestHandler.set_header* now properly sanitizes input values to protect against header injection, response splitting, etc. (it has always attempted to do this, but the check was incorrect). Note that redirects, the most likely source of such bugs, are protected by a separate check in *RequestHandler.redirect*.

Bug fixes

- Colored logging configuration in *tornado.options* is compatible with Python 3.2.3 (and 3.3).

4.9.23 What's new in Tornado 2.2

Jan 30, 2012

Highlights

- Updated and expanded WebSocket support.
- Improved compatibility in the Twisted/Tornado bridge.
- Template errors now generate better stack traces.
- Better exception handling in *tornado.gen*.

Security fixes

- `tornado.simple_httpclient` now disables SSLv2 in all cases. Previously SSLv2 would be allowed if the Python interpreter was linked against a pre-1.0 version of OpenSSL.

Backwards-incompatible changes

- *tornado.process.fork_processes* now raises `SystemExit` if all child processes exit cleanly rather than returning `None`. The old behavior was surprising and inconsistent with most of the documented examples of this function (which did not check the return value).
- On Python 2.6, `tornado.simple_httpclient` only supports SSLv3. This is because Python 2.6 does not expose a way to support both SSLv3 and TLSv1 without also supporting the insecure SSLv2.
- *tornado.websocket* no longer supports the older "draft 76" version of the websocket protocol by default, although this version can be enabled by overriding `tornado.websocket.WebSocketHandler.allow_draft76`.

tornado.httpclient

- `SimpleAsyncHTTPClient` no longer hangs on `HEAD` requests, responses with no content, or empty `POST`/`PUT` response bodies.
- `SimpleAsyncHTTPClient` now supports 303 and 307 redirect codes.
- `tornado.curl_httpclient` now accepts non-integer timeouts.
- `tornado.curl_httpclient` now supports basic authentication with an empty password.

tornado.httpserver

- *HTTPServer* with `xheaders=True` will no longer accept `X-Real-IP` headers that don't look like valid IP addresses.
- *HTTPServer* now treats the `Connection` request header as case-insensitive.

tornado.ioloop and tornado.iostream

- `IOStream.write` now works correctly when given an empty string.
- `IOStream.read_until` (and `read_until_regex`) now perform better when there is a lot of buffered data, which improves peformance of `SimpleAsyncHTTPClient` when downloading files with lots of chunks.
- *`SSLIOStream`* now works correctly when `ssl_version` is set to a value other than `SSLv23`.
- Idle `IOLoops` no longer wake up several times a second.
- *`tornado.ioloop.PeriodicCallback`* no longer triggers duplicate callbacks when stopped and started repeatedly.

tornado.template

- Exceptions in template code will now show better stack traces that reference lines from the original template file.
- `{#` and `#}` can now be used for comments (and unlike the old `{% comment %}` directive, these can wrap other template directives).
- Template directives may now span multiple lines.

tornado.web

- Now behaves better when given malformed `Cookie` headers
- *`RequestHandler.redirect`* now has a `status` argument to send status codes other than 301 and 302.
- New method *`RequestHandler.on_finish`* may be overridden for post-request processing (as a counterpart to *`RequestHandler.prepare`*)
- *`StaticFileHandler`* now outputs `Content-Length` and `Etag` headers on `HEAD` requests.
- *`StaticFileHandler`* now has overridable `get_version` and `parse_url_path` methods for use in subclasses.
- *`RequestHandler.static_url`* now takes an `include_host` parameter (in addition to the old support for the `RequestHandler.include_host` attribute).

tornado.websocket

- Updated to support the latest version of the protocol, as finalized in RFC 6455.
- Many bugs were fixed in all supported protocol versions.
- *`tornado.websocket`* no longer supports the older "draft 76" version of the websocket protocol by default, although this version can be enabled by overriding `tornado.websocket.WebSocketHandler.allow_draft76`.
- *`WebSocketHandler.write_message`* now accepts a `binary` argument to send binary messages.
- Subprotocols (i.e. the `Sec-WebSocket-Protocol` header) are now supported; see the *`WebSocketHandler.select_subprotocol`* method for details.
- `.WebSocketHandler.get_websocket_scheme` can be used to select the appropriate url scheme (`ws://` or `wss://`) in cases where `HTTPRequest.protocol` is not set correctly.

Other modules

- *tornado.auth.TwitterMixin.authenticate_redirect* now takes a `callback_uri` parameter.
- *tornado.auth.TwitterMixin.twitter_request* now accepts both URLs and partial paths (complete URLs are useful for the search API which follows different patterns).
- Exception handling in *tornado.gen* has been improved. It is now possible to catch exceptions thrown by a `Task`.
- *tornado.netutil.bind_sockets* now works when `getaddrinfo` returns duplicate addresses.
- *tornado.platform.twisted* compatibility has been significantly improved. Twisted version 11.1.0 is now supported in addition to 11.0.0.
- *tornado.process.fork_processes* correctly reseeds the `random` module even when `os.urandom` is not implemented.
- *tornado.testing.main* supports a new flag `--exception_on_interrupt`, which can be set to false to make `Ctrl-C` kill the process more reliably (at the expense of stack traces when it does so).
- `tornado.version_info` is now a four-tuple so official releases can be distinguished from development branches.

4.9.24 What's new in Tornado 2.1.1

Oct 4, 2011

Bug fixes

- Fixed handling of closed connections with the `epoll` (i.e. Linux) `IOLoop`. Previously, closed connections could be shut down too early, which most often manifested as "Stream is closed" exceptions in `SimpleAsyncHTTPClient`.
- Fixed a case in which chunked responses could be closed prematurely, leading to truncated output.
- `IOStream.connect` now reports errors more consistently via logging and the close callback (this affects e.g. connections to localhost on FreeBSD).
- `IOStream.read_bytes` again accepts both `int` and `long` arguments.
- `PeriodicCallback` no longer runs repeatedly when `IOLoop` iterations complete faster than the resolution of `time.time()` (mainly a problem on Windows).

Backwards-compatibility note

- Listening for `IOLoop.ERROR` alone is no longer sufficient for detecting closed connections on an otherwise unused socket. `IOLoop.ERROR` must always be used in combination with `READ` or `WRITE`.

4.9.25 What's new in Tornado 2.1

Sep 20, 2011

Backwards-incompatible changes

- Support for secure cookies written by pre-1.0 releases of Tornado has been removed. The *RequestHandler.get_secure_cookie* method no longer takes an `include_name` parameter.
- The `debug` application setting now causes stack traces to be displayed in the browser on uncaught exceptions. Since this may leak sensitive information, debug mode is not recommended for public-facing servers.

Security fixes

- Diginotar has been removed from the default CA certificates file used by `SimpleAsyncHTTPClient`.

New modules

- *tornado.gen*: A generator-based interface to simplify writing asynchronous functions.
- *tornado.netutil*: Parts of *tornado.httpserver* have been extracted into a new module for use with non-HTTP protocols.
- *tornado.platform.twisted*: A bridge between the Tornado IOLoop and the Twisted Reactor, allowing code written for Twisted to be run on Tornado.
- *tornado.process*: Multi-process mode has been improved, and can now restart crashed child processes. A new entry point has been added at *tornado.process.fork_processes*, although `tornado.httpserver.HTTPServer.start` is still supported.

tornado.web

- *tornado.web.RequestHandler.write_error* replaces `get_error_html` as the preferred way to generate custom error pages (`get_error_html` is still supported, but deprecated)
- In *tornado.web.Application*, handlers may be specified by (fully-qualified) name instead of importing and passing the class object itself.
- It is now possible to use a custom subclass of `StaticFileHandler` with the `static_handler_class` application setting, and this subclass can override the behavior of the `static_url` method.
- *StaticFileHandler* subclasses can now override `get_cache_time` to customize cache control behavior.
- *tornado.web.RequestHandler.get_secure_cookie* now has a `max_age_days` parameter to allow applications to override the default one-month expiration.
- *set_cookie* now accepts a `max_age` keyword argument to set the `max-age` cookie attribute (note underscore vs dash)
- *tornado.web.RequestHandler.set_default_headers* may be overridden to set headers in a way that does not get reset during error handling.
- *RequestHandler.add_header* can now be used to set a header that can appear multiple times in the response.
- *RequestHandler.flush* can now take a callback for flow control.

- The `application/json` content type can now be gzipped.
- The cookie-signing functions are now accessible as static functions `tornado.web.create_signed_value` and `tornado.web.decode_signed_value`.

`tornado.httpserver`

- To facilitate some advanced multi-process scenarios, `HTTPServer` has a new method `add_sockets`, and socket-opening code is available separately as *`tornado.netutil.bind_sockets`*.
- The `cookies` property is now available on `tornado.httpserver.HTTPRequest` (it is also available in its old location as a property of *`RequestHandler`*)
- `tornado.httpserver.HTTPServer.bind` now takes a backlog argument with the same meaning as `socket.listen`.
- *`HTTPServer`* can now be run on a unix socket as well as TCP.
- Fixed exception at startup when `socket.AI_ADDRCONFIG` is not available, as on Windows XP

`IOLoop` and `IOStream`

- *`IOStream`* performance has been improved, especially for small synchronous requests.
- New methods `tornado.iostream.IOStream.read_until_close` and `tornado.iostream.IOStream.read_until_regex`.
- `IOStream.read_bytes` and `IOStream.read_until_close` now take a `streaming_callback` argument to return data as it is received rather than all at once.
- *`IOLoop.add_timeout`* now accepts `datetime.timedelta` objects in addition to absolute timestamps.
- *`PeriodicCallback`* now sticks to the specified period instead of creeping later due to accumulated errors.
- *`tornado.ioloop.IOLoop`* and *`tornado.httpclient.HTTPClient`* now have `close()` methods that should be used in applications that create and destroy many of these objects.
- *`IOLoop.install`* can now be used to use a custom subclass of IOLoop as the singleton without monkey-patching.
- *`IOStream`* should now always call the close callback instead of the connect callback on a connection error.
- The *`IOStream`* close callback will no longer be called while there are pending read callbacks that can be satisfied with buffered data.

`tornado.simple_httpclient`

- Now supports client SSL certificates with the `client_key` and `client_cert` parameters to *`tornado.httpclient.HTTPRequest`*
- Now takes a maximum buffer size, to allow reading files larger than 100MB
- Now works with HTTP 1.0 servers that don't send a Content-Length header
- The `allow_nonstandard_methods` flag on HTTP client requests now permits methods other than `POST` and `PUT` to contain bodies.
- Fixed file descriptor leaks and multiple callback invocations in `SimpleAsyncHTTPClient`
- No longer consumes extra connection resources when following redirects.

- Now works with buggy web servers that separate headers with `\n` instead of `\r\n\r\n`.
- Now sets `response.request_time` correctly.
- Connect timeouts now work correctly.

Other modules

- *tornado.auth.OpenIdMixin* now uses the correct realm when the callback URI is on a different domain.
- *tornado.autoreload* has a new command-line interface which can be used to wrap any script. This replaces the `--autoreload` argument to *tornado.testing.main* and is more robust against syntax errors.
- *tornado.autoreload.watch* can be used to watch files other than the sources of imported modules.
- `tornado.database.Connection` has new variants of `execute` and `executemany` that return the number of rows affected instead of the last inserted row id.
- *tornado.locale.load_translations* now accepts any properly-formatted locale name, not just those in the predefined `LOCALE_NAMES` list.
- *tornado.options.define* now takes a `group` parameter to group options in `--help` output.
- Template loaders now take a `namespace` constructor argument to add entries to the template namespace.
- *tornado.websocket* now supports the latest ("hybi-10") version of the protocol (the old version, "hixie-76" is still supported; the correct version is detected automatically).
- *tornado.websocket* now works on Python 3

Bug fixes

- Windows support has been improved. Windows is still not an officially supported platform, but the test suite now passes and *tornado.autoreload* works.
- Uploading files whose names contain special characters will now work.
- Cookie values containing special characters are now properly quoted and unquoted.
- Multi-line headers are now supported.
- Repeated Content-Length headers (which may be added by certain proxies) are now supported in *HTTPServer*.
- Unicode string literals now work in template expressions.
- The template `{% module %}` directive now works even if applications use a template variable named `modules`.
- Requests with "Expect: 100-continue" now work on python 3

4.9.26 What's new in Tornado 2.0

Jun 21, 2011

```
Major changes:
* Template output is automatically escaped by default; see backwards
  compatibility note below.
* The default AsyncHTTPClient implementation is now simple_httpclient.
* Python 3.2 is now supported.

Backwards compatibility:
* Template autoescaping is enabled by default.  Applications upgrading from
  a previous release of Tornado must either disable autoescaping or adapt
  their templates to work with it.  For most applications, the simplest
  way to do this is to pass autoescape=None to the Application constructor.
  Note that this affects certain built-in methods, e.g. xsrf_form_html
  and linkify, which must now be called with {% raw %} instead of {}
* Applications that wish to continue using curl_httpclient instead of
  simple_httpclient may do so by calling
    AsyncHTTPClient.configure("tornado.curl_httpclient.CurlAsyncHTTPClient")
  at the beginning of the process.  Users of Python 2.5 will probably want
  to use curl_httpclient as simple_httpclient only supports ssl on Python 2.6+.
* Python 3 compatibility involved many changes throughout the codebase,
  so users are encouraged to test their applications more thoroughly than
  usual when upgrading to this release.

Other changes in this release:
* Templates support several new directives:
  - {% autoescape ...%} to control escaping behavior
  - {% raw ... %} for unescaped output
  - {% module ... %} for calling UIModules
* {% module Template(path, **kwargs) %} may now be used to call another
  template with an independent namespace
* All IOStream callbacks are now run directly on the IOLoop via add_callback.
* HTTPServer now supports IPv6 where available.  To disable, pass
  family=socket.AF_INET to HTTPServer.bind().
* HTTPClient now supports IPv6, configurable via allow_ipv6=bool on the
  HTTPRequest.  allow_ipv6 defaults to false on simple_httpclient and true
  on curl_httpclient.
* RequestHandlers can use an encoding other than utf-8 for query parameters
  by overriding decode_argument()
* Performance improvements, especially for applications that use a lot of
  IOLoop timeouts
* HTTP OPTIONS method no longer requires an XSRF token.
* JSON output (RequestHandler.write(dict)) now sets Content-Type to
  application/json
* Etag computation can now be customized or disabled by overriding
  RequestHandler.compute_etag
* USE_SIMPLE_HTTPCLIENT environment variable is no longer supported.
  Use AsyncHTTPClient.configure instead.
```

4.9.27 What's new in Tornado 1.2.1

Mar 3, 2011

```
We are pleased to announce the release of Tornado 1.2.1, available from
https://github.com/downloads/facebook/tornado/tornado-1.2.1.tar.gz

This release contains only two small changes relative to version 1.2:
* FacebookGraphMixin has been updated to work with a recent change to the
```

```
  Facebook API.
* Running "setup.py install" will no longer attempt to automatically
  install pycurl.  This wasn't working well on platforms where the best way
  to install pycurl is via something like apt-get instead of easy_install.

This is an important upgrade if you are using FacebookGraphMixin, but
otherwise it can be safely ignored.
```

4.9.28 What's new in Tornado 1.2

Feb 20, 2011

```
We are pleased to announce the release of Tornado 1.2, available from
https://github.com/downloads/facebook/tornado/tornado-1.2.tar.gz

Backwards compatibility notes:
* This release includes the backwards-incompatible security change from
  version 1.1.1.  Users upgrading from 1.1 or earlier should read the
  release notes from that release:
  http://groups.google.com/group/python-tornado/browse_thread/thread/b36191c781580cde
* StackContexts that do something other than catch exceptions may need to
  be modified to be reentrant.
  https://github.com/tornadoweb/tornado/commit/7a7e24143e77481d140fb5579bc67e4c45cbcfad
* When XSRF tokens are used, the token must also be present on PUT and
  DELETE requests (anything but GET and HEAD)

New features:
* A new HTTP client implementation is available in the module
  tornado.simple_httpclient.  This HTTP client does not depend on pycurl.
  It has not yet been tested extensively in production, but is intended
  to eventually replace the pycurl-based HTTP client in a future release of
  Tornado.  To transparently replace tornado.httpclient.AsyncHTTPClient with
  this new implementation, you can set the environment variable
  USE_SIMPLE_HTTPCLIENT=1 (note that the next release of Tornado will
  likely include a different way to select HTTP client implementations)
* Request logging is now done by the Application rather than the
  RequestHandler.  Logging behavior may be customized by either overriding
  Application.log_request in a subclass or by passing log_function
  as an Application setting
* Application.listen(port): Convenience method as an alternative to
  explicitly creating an HTTPServer
* tornado.escape.linkify(): Wrap urls in <a> tags
* RequestHandler.create_signed_value(): Create signatures like the
  secure_cookie methods without setting cookies.
* tornado.testing.get_unused_port(): Returns a port selected in the same
  way as inAsyncHTTPTestCase
* AsyncHTTPTestCase.fetch(): Convenience method for synchronous fetches
* IOLoop.set_blocking_signal_threshold(): Set a callback to be run when
  the IOLoop is blocked.
* IOStream.connect(): Asynchronously connect a client socket
* AsyncHTTPClient.handle_callback_exception(): May be overridden
  in subclass for custom error handling
* httpclient.HTTPRequest has two new keyword arguments, validate_cert and
  ca_certs. Setting validate_cert=False will disable all certificate checks
  when fetching https urls.  ca_certs may be set to a filename containing
  trusted certificate authorities (defaults will be used if this is
```

```
  unspecified)
* HTTPRequest.get_ssl_certificate(): Returns the client's SSL certificate
  (if client certificates were requested in the server's ssl_options
* StaticFileHandler can be configured to return a default file (e.g.
  index.html) when a directory is requested
* Template directives of the form "{% from x import y %}" are now
  supported (in addition to the existing support for "{% import x
  %}"
* FacebookGraphMixin.get_authenticated_user now accepts a new
  parameter 'extra_fields' which may be used to request additional
  information about the user

Bug fixes:
* auth: Fixed KeyError with Facebook offline_access
* auth: Uses request.uri instead of request.path as the default redirect
  so that parameters are preserved.
* escape: xhtml_escape() now returns a unicode string, not
  utf8-encoded bytes
* ioloop: Callbacks added with add_callback are now run in the order they
  were added
* ioloop: PeriodicCallback.stop can now be called from inside the callback.
* iostream: Fixed several bugs in SSLIOStream
* iostream: Detect when the other side has closed the connection even with
  the select()-based IOLoop
* iostream: read_bytes(0) now works as expected
* iostream: Fixed bug when writing large amounts of data on windows
* iostream: Fixed infinite loop that could occur with unhandled exceptions
* httpclient: Fix bugs when some requests use proxies and others don't
* httpserver: HTTPRequest.protocol is now set correctly when using the
  built-in SSL support
* httpserver: When using multiple processes, the standard library's
  random number generator is re-seeded in each child process
* httpserver: With xheaders enabled, X-Forwarded-Proto is supported as an
  alternative to X-Scheme
* httpserver: Fixed bugs in multipart/form-data parsing
* locale: format_date() now behaves sanely with dates in the future
* locale: Updates to the language list
* stack_context: Fixed bug with contexts leaking through reused IOStreams
* stack_context: Simplified semantics and improved performance
* web: The order of css_files from UIModules is now preserved
* web: Fixed error with default_host redirect
* web: StaticFileHandler works when os.path.sep != '/' (i.e. on Windows)
* web: Fixed a caching-related bug in StaticFileHandler when a file's
  timestamp has changed but its contents have not.
* web: Fixed bugs with HEAD requests and e.g. Etag headers
* web: Fix bugs when different handlers have different static_paths
* web: @removeslash will no longer cause a redirect loop when applied to the
  root path
* websocket: Now works over SSL
* websocket: Improved compatibility with proxies

Many thanks to everyone who contributed patches, bug reports, and feedback
that went into this release!

-Ben
```

4.9.29 What's new in Tornado 1.1.1

Feb 8, 2011

```
Tornado 1.1.1 is a BACKWARDS-INCOMPATIBLE security update that fixes an
XSRF vulnerability.  It is available at
https://github.com/downloads/facebook/tornado/tornado-1.1.1.tar.gz

This is a backwards-incompatible change.  Applications that previously
relied on a blanket exception for XMLHTTPRequest may need to be modified
to explicitly include the XSRF token when making ajax requests.

The tornado chat demo application demonstrates one way of adding this
token (specifically the function postJSON in demos/chat/static/chat.js).

More information about this change and its justification can be found at
http://www.djangoproject.com/weblog/2011/feb/08/security/
http://weblog.rubyonrails.org/2011/2/8/csrf-protection-bypass-in-ruby-on-rails
```

4.9.30 What's new in Tornado 1.1

Sep 7, 2010

```
We are pleased to announce the release of Tornado 1.1, available from
https://github.com/downloads/facebook/tornado/tornado-1.1.tar.gz

Changes in this release:
* RequestHandler.async_callback and related functions in other classes
  are no longer needed in most cases (although it's harmless to continue
  using them).  Uncaught exceptions will now cause the request to be closed
  even in a callback.  If you're curious how this works, see the new
  tornado.stack_context module.
* The new tornado.testing module contains support for unit testing
  asynchronous IOLoop-based code.
* AsyncHTTPClient has been rewritten (the new implementation was
  available as AsyncHTTPClient2 in Tornado 1.0; both names are
  supported for backwards compatibility).
* The tornado.auth module has had a number of updates, including support
  for OAuth 2.0 and the Facebook Graph API, and upgrading Twitter and
  Google support to OAuth 1.0a.
* The websocket module is back and supports the latest version (76) of the
  websocket protocol.  Note that this module's interface is different
  from the websocket module that appeared in pre-1.0 versions of Tornado.
* New method RequestHandler.initialize() can be overridden in subclasses
  to simplify handling arguments from URLSpecs.  The sequence of methods
  called during initialization is documented at
  http://tornadoweb.org/documentation#overriding-requesthandler-methods
* get_argument() and related methods now work on PUT requests in addition
  to POST.
* The httpclient module now supports HTTP proxies.
* When HTTPServer is run in SSL mode, the SSL handshake is now non-blocking.
* Many smaller bug fixes and documentation updates

Backwards-compatibility notes:
* While most users of Tornado should not have to deal with the stack_context
  module directly, users of worker thread pools and similar constructs may
```

```
  need to use stack_context.wrap and/or NullContext to avoid memory leaks.
* The new AsyncHTTPClient still works with libcurl version 7.16.x, but it
  performs better when both libcurl and pycurl are at least version 7.18.2.
* OAuth transactions started under previous versions of the auth module
  cannot be completed under the new module.  This applies only to the
  initial authorization process; once an authorized token is issued that
  token works with either version.

Many thanks to everyone who contributed patches, bug reports, and feedback
that went into this release!

-Ben
```

4.9.31 What's new in Tornado 1.0.1

Aug 13, 2010

```
This release fixes a bug with RequestHandler.get_secure_cookie, which would
in some circumstances allow an attacker to tamper with data stored in the
cookie.
```

4.9.32 What's new in Tornado 1.0

July 22, 2010

```
We are pleased to announce the release of Tornado 1.0, available
from
https://github.com/downloads/facebook/tornado/tornado-1.0.tar.gz.
There have been many changes since version 0.2; here are some of
the highlights:

New features:
* Improved support for running other WSGI applications in a
  Tornado server (tested with Django and CherryPy)
* Improved performance on Mac OS X and BSD (kqueue-based IOLoop),
  and experimental support for win32
* Rewritten AsyncHTTPClient available as
  tornado.httpclient.AsyncHTTPClient2 (this will become the
  default in a future release)
* Support for standard .mo files in addition to .csv in the locale
  module
* Pre-forking support for running multiple Tornado processes at
  once (see HTTPServer.start())
* SSL and gzip support in HTTPServer
* reverse_url() function refers to urls from the Application
  config by name from templates and RequestHandlers
* RequestHandler.on_connection_close() callback is called when the
  client has closed the connection (subject to limitations of the
  underlying network stack, any proxies, etc)
* Static files can now be served somewhere other than /static/ via
  the static_url_prefix application setting
* URL regexes can now use named groups ("(?P<name>)") to pass
  arguments to get()/post() via keyword instead of position
* HTTP header dictionary-like objects now support multiple values
```

```
  for the same header via the get_all() and add() methods.
* Several new options in the httpclient module, including
  prepare_curl_callback and header_callback
* Improved logging configuration in tornado.options.
* UIModule.html_body() can be used to return html to be inserted
  at the end of the document body.

Backwards-incompatible changes:
* RequestHandler.get_error_html() now receives the exception
  object as a keyword argument if the error was caused by an
  uncaught exception.
* Secure cookies are now more secure, but incompatible with
  cookies set by Tornado 0.2.  To read cookies set by older
  versions of Tornado, pass include_name=False to
  RequestHandler.get_secure_cookie()
* Parameters passed to RequestHandler.get/post() by extraction
  from the path now have %-escapes decoded, for consistency with
  the processing that was already done with other query
  parameters.

Many thanks to everyone who contributed patches, bug reports, and
feedback that went into this release!

-Ben
```

- genindex
- modindex
- search

CHAPTER 5

Discussion and support

You can discuss Tornado on the Tornado developer mailing list, and report bugs on the GitHub issue tracker. Links to additional resources can be found on the Tornado wiki. New releases are announced on the announcements mailing list.

t

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y