

---

# **Tornado Documentation**

*Release 6.0.dev1*

**The Tornado Authors**

**Oct 07, 2018**



---

## Contents

---

<b>1</b>	<b>Quick links</b>	<b>3</b>
<b>2</b>	<b>Hello, world</b>	<b>5</b>
<b>3</b>	<b>Threads and WSGI</b>	<b>7</b>
<b>4</b>	<b>Installation</b>	<b>9</b>
<b>5</b>	<b>Documentation</b>	<b>11</b>
5.1	User's guide . . . . .	11
5.2	Web framework . . . . .	37
5.3	HTTP servers and clients . . . . .	45
5.4	Asynchronous networking . . . . .	52
5.5	Coroutines and concurrency . . . . .	52
5.6	Integration with other services . . . . .	53
5.7	Utilities . . . . .	54
5.8	Frequently Asked Questions . . . . .	61
5.9	Release notes . . . . .	63
<b>6</b>	<b>Discussion and support</b>	<b>133</b>
	<b>Python Module Index</b>	<b>135</b>



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

---

- [Current version: 6.0.dev1](#) (download from PyPI, *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](#).



---

### Threads and WSGI

---

Tornado is different from most Python web frameworks. It is not based on WSGI, and it is typically run with only one thread per process. See the *User's guide* for more on Tornado's approach to asynchronous programming.

While some support of WSGI is available in the `tornado.wsgi` module, it is not a focus of development and most applications should be written to use Tornado's own interfaces (such as `tornado.web`) directly instead of using WSGI.

In general, Tornado code is not thread-safe. The only method in Tornado that is safe to call from other threads is `IOLoop.add_callback`. You can also use `IOLoop.run_in_executor` to asynchronously run a blocking function on another thread, but note that the function passed to `run_in_executor` should avoid referencing any Tornado objects. `run_in_executor` is the recommended way to interact with blocking code.



---

## Installation

---

```
pip install tornado
```

Tornado is listed in [PyPI](#) and can be installed with `pip`. 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 or clone the [git repository](#) as well.

**Prerequisites:** Tornado 5.x runs on Python 2.7, and 3.4+ (Tornado 6.0 will require Python 3.5+; Python 2 will no longer be supported). The updates to the `ssl` module in Python 2.7.9 are required (in some distributions, these updates may be available in older python versions). In addition to the requirements which will be installed automatically by `pip` or `setup.py install`, the following optional packages may be useful:

- `pycurl` is used by the optional `tornado.curl_httpclient`. Libcurl version 7.22 or higher is required.
- `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 adjustments are frequent. No longer needed in Python 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. Without reworking Tornado IOloop interface, it's not possible to add a native Tornado Windows IOloop implementation or leverage Windows' IOCP support from frameworks like AsyncIO or Twisted.



This documentation is also available in [PDF](#) and [Epub](#) formats.

## 5.1 User's guide

### 5.1.1 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 implementations of HTTP (`HTTPServer` and `AsyncHTTPClient`).
- An asynchronous networking library including the classes `IOLoop` and `IStream`, 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. This is similar to the native coroutine feature introduced in Python 3.5 (`async def`). Native coroutines are recommended in place of the `tornado.gen` module when available.

The Tornado web framework and HTTP server together offer a full-stack alternative to [WSGI](#). While it is possible to use the Tornado HTTP server as a container for other WSGI frameworks ([WSGIContainer](#)), this combination has limitations and to take full advantage of Tornado you will need to use the Tornado's web framework and HTTP server together.

## 5.1.2 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. 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).

Asynchronous operations in Tornado generally return placeholder objects (`Futures`), with the exception of some low-level components like the `IOLoop` that use callbacks. `Futures` are usually transformed into their result with the `await` or `yield` keywords.

### 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 asynchronously as a native coroutine:



```

from tornado.httpclient import AsyncHTTPClient

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

```

Or for compatibility with older versions of Python, using the `tornado.gen` module:

```

from tornado.httpclient import AsyncHTTPClient
from tornado import gen

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

```

Coroutines are a little magical, but what they do internally is something like this:

```

from tornado.concurrent import Future

def async_fetch_manual(url):
    http_client = AsyncHTTPClient()
    my_future = Future()
    fetch_future = http_client.fetch(url)
    def on_fetch(f):
        my_future.set_result(f.result().body)
    fetch_future.add_done_callback(on_fetch)
    return my_future

```

Notice that the coroutine returns its `Future` before the fetch is done. This is what makes coroutines *asynchronous*.

Anything you can do with coroutines you can also do by passing callback objects around, but coroutines provide an important simplification by letting you organize your code in the same way you would if it were synchronous. This is especially important for error handling, since `try/except` blocks work as you would expect in coroutines while this is difficult to achieve with callbacks. Coroutines will be discussed in depth in the next section of this guide.

### 5.1.3 Coroutines

**Coroutines** are the recommended way to write asynchronous code in Tornado. Coroutines use the Python `await` or `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:

```

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

```

## Native vs decorated coroutines

Python 3.5 introduced the `async` and `await` keywords (functions using these keywords are also called “native coroutines”). For compatibility with older versions of Python, you can use “decorated” or “yield-based” coroutines using the `tornado.gen.coroutine` decorator.

Native coroutines are the recommended form whenever possible. Only use decorated coroutines when compatibility with older versions of Python is required. Examples in the tornado documentation will generally use the native form.

Translation between the two forms is generally straightforward:

```
# Decorated:                                # Native:

# Normal function declaration                # "async def" keywords
# with decorator                             # "await" all async funcs
@gen.coroutine                               # "yield" all async funcs
def a():                                     async def a():
    # "yield" all async funcs                # "await" all async funcs
    b = yield c()                            b = await c()
    # "return" and "yield"                  # "return" normally
    # cannot be mixed in                    # Return normally
    # Python 2, so raise a                   # Return normally
    # special exception.                     # Return normally
    raise gen.Return(b)                     return b
```

Other differences between the two forms of coroutine are:

- Native coroutines are generally faster.
- Native coroutines can use `async for` and `async with` statements which make some patterns much simpler.
- Native coroutines do not run at all unless you `await` or `yield` them. Decorated coroutines can start running “in the background” as soon as they are called. Note that for both kinds of coroutines it is important to use `await` or `yield` so that any exceptions have somewhere to go.
- Decorated coroutines have additional integration with the `concurrent.futures` package, allowing the result of `executor.submit` to be yielded directly. For native coroutines, use `IOLoop.run_in_executor` instead.
- Decorated coroutines support some shorthand for waiting on multiple objects by yielding a list or dict. Use `tornado.gen.multi` to do this in native coroutines.
- Decorated coroutines can support integration with other packages including Twisted via a registry of conversion functions. To access this functionality in native coroutines, use `tornado.gen.convert_yielded`.
- Decorated coroutines always return a `Future` object. Native coroutines return an *awaitable* object that is not a `Future`. In Tornado the two are mostly interchangeable.

## How it works

This section explains the operation of decorated coroutines. Native coroutines are conceptually similar, but a little more complicated because of the extra integration with the Python runtime.

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 awaitable object until it is yielded. This means it is important to call coroutines in the right way, or you may have errors that go unnoticed:

```
async 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 `await` or `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`”):

```
async def good_call():
    # await will unwrap the object returned by divide() and raise
    # the exception.
    await 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)
```

Using `IOLoop.spawn_callback` in this way is *recommended* for functions using `@gen.coroutine`, but it is *required* for functions using `async def` (otherwise the coroutine runner will not start).

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

### Calling blocking functions

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

```
async def call_blocking():
    await IOLoop.current().run_in_executor(None, blocking_func, args)
```

## Parallelism

The `multi` function accepts lists and dicts whose values are Futures, and waits for all of those Futures in parallel:

```
from tornado.gen import multi

async def parallel_fetch(url1, url2):
    resp1, resp2 = await multi([http_client.fetch(url1),
                               http_client.fetch(url2)])

async def parallel_fetch_many(urls):
    responses = await multi ([http_client.fetch(url) for url in urls])
    # responses is a list of HTTPResponses in the same order

async def parallel_fetch_dict(urls):
    responses = await multi({url: http_client.fetch(url)
                             for url in urls})
    # responses is a dict {url: HTTPResponse}
```

In decorated coroutines, it is possible to yield the list or dict directly:

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

## Interleaving

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

```
from tornado.gen import convert_yielded

async def get(self):
    # convert_yielded() starts the native coroutine in the background.
    # This is equivalent to asyncio.ensure_future() (both work in Tornado).
```

(continues on next page)

(continued from previous page)

```

fetch_future = convert_yielded(self.fetch_next_chunk())
while True:
    chunk = yield fetch_future
    if chunk is None: break
    self.write(chunk)
    fetch_future = convert_yielded(self.fetch_next_chunk())
    yield self.flush()

```

This is a little easier to do with decorated coroutines, because they start immediately when called:

```

@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

In native coroutines, `async for` can be used. In older versions of Python, looping is tricky with coroutines since there is no way 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`:

```

async def minute_loop():
    while True:
        await do_something()
        await 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:

```

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

```

### 5.1.4 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 python3

import time
from datetime import timedelta

from html.parser import HTMLParser
from urllib.parse import urljoin, urldefrag

from tornado import gen, httpclient, ioloop, queues

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

async 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'.
    """
    response = await httpclient.AsyncHTTPClient().fetch(url)
    print('fetched %s' % url)

    html = response.body.decode(errors='ignore')
    return [urljoin(url, remove_fragment(new_url))
            for new_url in get_links(html)]

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

```

(continues on next page)

(continued from previous page)

```

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

async def main():
    q = queues.Queue()
    start = time.time()
    fetching, fetched = set(), set()

    async def fetch_url(current_url):
        if current_url in fetching:
            return

        print('fetching %s' % current_url)
        fetching.add(current_url)
        urls = await 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):
                await q.put(new_url)

    async def worker():
        async for url in q:
            if url is None:
                return
            try:
                await fetch_url(url)
            except Exception as e:
                print('Exception: %s %s' % (e, url))
            finally:
                q.task_done()

    await q.put(base_url)

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

    # Signal all the workers to exit.

```

(continues on next page)

(continued from previous page)

```

for _ in range(concurrency):
    await q.put(None)
await workers

if __name__ == '__main__':
    io_loop = ioloop.IOLoop.current()
    io_loop.run_sync(main)

```

### 5.1.5 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):

```

(continues on next page)



(continued from previous page)

```

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.

In the demos directory, `file_receiver.py` shows both methods of receiving file uploads.

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.get("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. This is generally after `get()` or another HTTP method returns.

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/{0}")),
])
```

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

Certain handler methods (including `prepare()` and the HTTP verb methods `get()/post()/etc`) may be overridden as coroutines to make the handler asynchronous.

For example, here is a simple handler using a coroutine:

```
class MainHandler(tornado.web.RequestHandler):
    async def get(self):
        http = tornado.httpclient.AsyncHTTPClient()
        response = await 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).

## 5.1.6 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`
- `xsrform_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):
```

(continues on next page)

(continued from previous page)

```

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.

## 5.1.7 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)
```

(continues on next page)

(continued from previous page)

```

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):
    async def get(self):

```

(continues on next page)

(continued from previous page)

```

if self.get_argument('code', False):
    user = await 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:
    await 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 `xsrp_cookies`:

```

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

```

If `xsrp_cookies` is set, the Tornado web application will set the `_xsrp` cookie for all users and reject all POST, PUT, and DELETE requests that do not contain a correct `_xsrp` 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 `_xsrp` value with each request. This is the `jQuery` function we use at FriendFeed for AJAX POST requests that automatically adds the `_xsrp` value to all requests:

```

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

```

(continues on next page)

```

}

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.

## DNS Rebinding

DNS rebinding is an attack that can bypass the same-origin policy and allow external sites to access resources on private networks. This attack involves a DNS name (with a short TTL) that alternates between returning an IP address controlled by the attacker and one controlled by the victim (often a guessable private IP address such as 127.0.0.1 or 192.168.1.1).

Applications that use TLS are *not* vulnerable to this attack (because the browser will display certificate mismatch warnings that block automated access to the target site).

Applications that cannot use TLS and rely on network-level access controls (for example, assuming that a server on 127.0.0.1 can only be accessed by the local machine) should guard against DNS rebinding by validating the Host HTTP header. This means passing a restrictive hostname pattern to either a `HostMatches` router or the first argument of `Application.add_handlers`:

```

# BAD: uses a default host pattern of r'.*'
app = Application(['/foo', FooHandler])

# GOOD: only matches localhost or its ip address.
app = Application()
app.add_handlers(r'(localhost|127\.0\.0\.1)',
    ['/foo', FooHandler])

# GOOD: same as previous example using tornado.routing.
app = Application([
    (HostMatches(r'(localhost|127\.0\.0\.1)'),
    ['/foo', FooHandler]),
])

```

In addition, the `default_host` argument to `Application` and the `DefaultHostMatches` router must not be used in applications that may be vulnerable to DNS rebinding, because it has a similar effect to a wildcard host pattern.

### 5.1.8 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;
```

(continues on next page)

```
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;
```

(continues on next page)

(continued from previous page)

```

    }

    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",
    "xcsrf_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. Tornado 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></div>

```

(continues on next page)

(continued from previous page)

```
</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 almost 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.



## 5.2 Web framework

### 5.2.1 tornado.web — RequestHandler and Application classes

### 5.2.2 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 ** 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 ** = *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*) → bytes

Generate this template with the given arguments.

**class** tornado.template.**BaseLoader** (*autoescape: str = 'html\_escape', namespace: Dict[str, Any] = None, whitespace: str = 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 “`html_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** () → None

Resets the cache of compiled templates.

**resolve\_path** (*name: str, parent\_path: str = None*) → str

Converts a possibly-relative path to absolute (used internally).

**load** (*name: str, parent\_path: str = None*) → tornado.template.Template

Loads a template.

**class** tornado.template.**Loader** (*root\_directory: str, \*\*kwargs*)

A template loader that loads from a single root directory.

**class** tornado.template.**DictLoader** (*dict: Dict[str, str], \*\*kwargs*)

A template loader that loads from a dictionary.

**exception** tornado.template.**ParseError** (*message: str, filename: str = None, lineno: int = 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: str, text: str*) → str

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.

### 5.2.3 `tornado.routing` — Basic routing implementation

### 5.2.4 `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: Union[str, bytes]*) → str

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: Union[str, bytes]*) → str

Un-escapes an XML-escaped string.

`tornado.escape.url_escape` (*value: Union[str, bytes]*, *plus: bool = True*) → str

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: Union[str, bytes]*, *encoding: Optional[str] = 'utf-8'*, *plus: bool = True*) → Union[str, bytes]

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: Any*) → str

JSON-encodes the given Python object.

`tornado.escape.json_decode` (*value: Union[str, bytes]*) → Any

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: Union[None, str, bytes]*) → Optional[bytes]

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: Union[None, str, bytes]*) → Optional[str]

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: Union[None, str, bytes]*) → Optional[str]

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: Any*) → Any

Walks a simple data structure, converting byte strings to unicode.

Supports lists, tuples, and dictionaries.

## Miscellaneous functions

`tornado.escape.linkify` (*text: Union[str, bytes]*, *shorten: bool = False*, *extra\_params: Union[str, Callable[[str], str]] = ''*, *require\_protocol: bool = False*, *permitted\_protocols: List[str] = ['http', 'https']*) → str

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: str*) → *str*

Replace all sequences of whitespace chars with a single space.

## 5.2.5 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)` → `tornado.locale.Locale`

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: str*) → `None`

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: str, encoding: str = None*) → `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*: *str*, *domain*: *str*) → None

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 your 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` () → Iterable[str]

Returns a list of all the supported locale codes.

**class** `tornado.locale.Locale` (*code*: *str*)

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*) → `tornado.locale.Locale`

Returns the closest match for the given locale code.

**classmethod** `get` (*code*: *str*) → `tornado.locale.Locale`

Returns the `Locale` for the given locale code.

If it is not supported, we raise an exception.

**translate** (*message*: *str*, *plural\_message*: *str* = None, *count*: *int* = None) → *str*

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*: *Union[int, float, datetime.datetime]*, *gmt\_offset*: *int* = 0, *relative*: *bool* = True,

*shorter*: *bool* = False, *full\_format*: *bool* = False) → *str*

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: datetime.datetime, gmt\_offset: int = 0, dow: bool = True*) → bool

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: Any*) → str

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: int*) → str

Returns a comma-separated number for the given integer.

**class** `tornado.locale.CSVLocale` (*code: str, translations: Dict[str, Dict[str, str]]*)

Locale implementation using tornado’s CSV translation format.

**class** `tornado.locale.GetTextLocale` (*code: str, translations: gettext.NullTranslations*)

Locale implementation using the `gettext` module.

**pgettext** (*context: str, message: str, plural\_message: str = None, count: int = None*) → str

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.

## 5.2.6 `tornado.websocket` — Bidirectional communication to the browser

## 5.3 HTTP servers and clients

### 5.3.1 `tornado.httpserver` — Non-blocking HTTP server

### 5.3.2 `tornado.httpclient` — Asynchronous HTTP client

#### Implementations

**class** `tornado.curl_httpclient.CurlAsyncHTTPClient` (*max\_clients=10, defaults=None*)

libcurl-based HTTP client.

## Example Code

- A simple webspider shows how to fetch URLs concurrently.
- The file uploader demo uses either HTTP POST or HTTP PUT to upload files to a server.

### 5.3.3 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: str, value: str) → None

Adds a new value for the given key.

**get\_list** (name: str) → List[str]

Returns all values for the given header as a list.

**get\_all** () → Iterable[Tuple[str, str]]

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: str) → None

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: str*) → tornado.httputil.HTTPHeaders

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')]
```

Changed in version 5.1: Raises `HTTPInputError` on malformed headers instead of a mix of `KeyError`, and `ValueError`.

**class** tornado.httputil.HTTPServerRequest (*method: str = None, uri: str = None, version: str = 'HTTP/1.0', headers: tornado.httputil.HTTPHeaders = None, body: bytes = None, host: str = None, files: Dict[str, List[HTTPFile]] = None, connection: Optional[tornado.httputil.HTTPConnection] = None, start\_line: Optional[tornado.httputil.RequestStartLine] = None, server\_connection: object = 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`.

**cookies**

A dictionary of `http.cookies.Morsel` objects.

**full\_url ()** → str

Reconstructs the full URL for this request.

**request\_time ()** → float

Returns the amount of time it took for this request to execute.

**get\_ssl\_certificate (binary\_form: bool = False)** → Union[None, Dict, bytes]

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*: object, *request\_conn*: tornado.httputil.HTTPConnection) → tornado.httputil.HTTPMessageDelegate

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*: object) → None

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*: Union[RequestStartLine, ResponseStartLine], *headers*: tornado.httputil.HTTPHeaders) → Optional[Awaitable[None]]

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*: bytes) → Optional[Awaitable[None]]

Called when a chunk of data has been received.

May return a *Future* for flow control.

**finish** () → None

Called after the last chunk of data has been received.

**on\_connection\_close** () → None

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*: Union[RequestStartLine, ResponseStartLine], *headers*: tornado.httputil.HTTPHeaders, *chunk*: bytes = None) → Future[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.

The `version` field of `start_line` is ignored.

Returns a future for flow control.

Changed in version 6.0: The `callback` argument was removed.

**write** (*chunk: bytes*) → Future[None]

Writes a chunk of body data.

Returns a future for flow control.

Changed in version 6.0: The `callback` argument was removed.

**finish** () → None

Indicates that the last body data has been written.

`tornado.httputil.url_concat` (*url: str, args: Union[None, Dict[str, str], List[Tuple[str, str]], Tuple[Tuple[str, str], ...]]*) → str

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: str, body: bytes, arguments: Dict[str, List[bytes]], files: Dict[str, List[tornado.httputil.HTTPFile]], headers: tornado.httputil.HTTPHeaders = None*) → 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: bytes, data: bytes, arguments: Dict[str, List[bytes]], files: Dict[str, List[tornado.httputil.HTTPFile]]*) → None

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.

Changed in version 5.1: Now recognizes non-ASCII filenames in RFC 2231/5987 (`filename*`) format.

`tornado.httputil.format_timestamp` (*ts: Union[int, float, tuple, datetime.datetime]*) → str  
 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)`

Create new instance of `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: str*) → `tornado.httputil.RequestStartLine`

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)`

Create new instance of `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: str*) → `tornado.httputil.ResponseStartLine`

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.encode_username_password` (*username: Union[str, bytes], password: Union[str, bytes]*) → bytes

Encodes a username/password pair in the format used by HTTP auth.

The return value is a byte string in the form `username:password`.

New in version 5.1.

`tornado.httputil.split_host_and_port` (*netloc: str*) → Tuple[str, Optional[int]]

Returns (host, port) tuple from netloc.

Returned port will be None if not present.

New in version 4.1.

`tornado.httputil.qs_to_qs1` (*qs: Dict[str, List[AnyStr]]*) → Iterable[Tuple[str, AnyStr]]

Generator converting a result of `parse_qs` back to name-value pairs.

New in version 5.0.

`tornado.httputil.parse_cookie` (*cookie: str*) → Dict[str, str]

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.

### 5.3.4 `tornado.http1connection` – HTTP/1.x client/server implementation

## 5.4 Asynchronous networking

### 5.4.1 `tornado.ioloop` — Main event loop

### 5.4.2 `tornado.iostream` — Convenient wrappers for non-blocking sockets

### 5.4.3 `tornado.netutil` — Miscellaneous network utilities

### 5.4.4 `tornado.tcpclient` — `IOStream` connection factory

### 5.4.5 `tornado.tcpserver` — Basic `IOStream`-based TCP server

## 5.5 Coroutines and concurrency

### 5.5.1 `tornado.gen` — Generator-based coroutines

### 5.5.2 `tornado.locks` – Synchronization primitives

New in version 4.2.

Coordinate coroutines with synchronization primitives analogous to those the standard library provides to threads. These classes are very similar to those provided in the standard library's `asyncio` package.

**Warning:** 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 multithreaded app.



### 5.5.3 `tornado.queues` – Queues for coroutines

New in version 4.2.

### 5.5.4 `tornado.process` — Utilities for multiple processes

## 5.6 Integration with other services

### 5.6.1 `tornado.auth` — Third-party login with OpenID and OAuth

### 5.6.2 `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 via the `WSGIContainer` class, which makes it possible to run applications using other WSGI frameworks on the Tornado HTTP server. The reverse is not supported; the Tornado `Application` and `RequestHandler` classes are designed for use with the Tornado `HTTPServer` and cannot be used in a generic WSGI container.

**class** `tornado.wsgi.WSGIContainer` (*wsgi\_application*: `WSGIAppType`)  
 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*: `tornado.httputil.HTTPServerRequest`) → `Dict[str, Any]`  
 Converts a `tornado.httputil.HTTPServerRequest` to a WSGI environment.

### 5.6.3 `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`.

### 5.6.4 `tornado.platform.twisted` — Bridges between Twisted and Tornado

### 5.6.5 `tornado.platform.asyncio` — Bridge between `asyncio` and Tornado

## 5.7 Utilities

### 5.7.1 `tornado.autoreload` — Automatically detect code changes in development

### 5.7.2 `tornado.concurrent` — Work with `Future` objects

### 5.7.3 `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` (*fmt*: `str = '%(color)s[%(levelname)1.1s %(asctime)s %(module)s:%(lineno)d](end_color)s %(message)s'`, *datefmt*: `str = '%y%m%d %H:%M:%S'`, *style*: `str = '%'`, *color*: `bool = True`, *colors*: `Dict[int, int] = {10: 4, 20: 2, 30: 3, 40: 1}`)

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).

Color support on Windows versions that do not support ANSI color codes is enabled by use of the `colorama` library. Applications that wish to use this must first initialize `colorama` with a call to `colorama.init`. See the `colorama` documentation for details.

Changed in version 4.5: Added support for `colorama`. Changed the constructor signature to be compatible with `logging.config.dictConfig`.

#### Parameters

- **color** (*bool*) – Enables color support.
- **fmt** (*str*) – 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** (*str*) – 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: Any = None, logger: logging.Logger = None*) → 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: Any = None*) → 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.

## 5.7.4 tornado.options — Command-line parsing

A command line parsing module that lets modules define their own options.

This module is inspired by Google's `gflags`. The primary difference with libraries such as `argparse` is that a global registry is used so that options may be defined in any module (it also enables `tornado.log` by default). The rest of Tornado does not depend on this module, so feel free to use `argparse` or other configuration libraries if you prefer them.

Options must be defined with `tornado.options.define` before use, generally at the top level of a module. The options are then accessible as attributes of `tornado.options.options`:

```
# myapp/db.py
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)
    ...

# myapp/server.py
```

(continues on next page)

(continued from previous page)

```

from tornado.options import define, options

define("port", default=8080, help="port to listen on")

def start_server():
    app = make_app()
    app.listen(options.port)

```

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 `parse_command_line` or `parse_config_file`:

```

import myapp.db, myapp.server
import tornado.options

if __name__ == '__main__':
    tornado.options.parse_command_line()
    # or
    tornado.options.parse_config_file("/etc/server.conf")

```

**Note:** When using multiple `parse_*` functions, pass `final=False` to all but the last one, or side effects may occur twice (in particular, this can result in log messages being doubled).

`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*: str, *default*: Any = None, *type*: type = None, *help*: str = None, *metavar*: str = None, *multiple*: bool = False, *group*: str = None, *callback*: Callable[[Any], None] = None) → 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: List[str] = None, final: bool = True*) → List[str]  
Parses global options from the command line.

See `OptionParser.parse_command_line`.

`tornado.options.parse_config_file` (*path: str, final: bool = True*) → None  
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: Callable[[], None]*) → None  
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.

`OptionParser.define` (*name: str, default: Any = None, type: type = None, help: str = None, metavar: str = None, multiple: bool = False, group: str = None, callback: Callable[[Any], None] = None*) → None  
Defines a new command line option.

`type` can be any of `str`, `int`, `float`, `bool`, `datetime`, or `timedelta`. If no `type` is given but a default is, `type` is the type of default. Otherwise, `type` defaults to `str`.

If `multiple` is `True`, the option value is a list of `type` instead of an instance of `type`.

`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.

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.

`OptionParser.parse_command_line` (*args: List[str] = None, final: bool = True*) → List[str]  
Parses all options given on the command line (defaults to `sys.argv`).

Options look like `--option=value` and are parsed according to their type. For boolean options, `--option` is equivalent to `--option=true`

If the option has `multiple=True`, comma-separated values are accepted. For multi-value integer options, the syntax `x:y` is also accepted and equivalent to `range(x, y)`.

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.

`OptionParser.parse_config_file` (*path: str, final: bool = True*) → None

Parses and loads the config file at the given path.

The config file contains Python code that will be executed (so it is **not safe** to use untrusted config files). Anything in the global namespace that matches a defined option will be used to set that option's value.

Options may either be the specified type for the option or strings (in which case they will be parsed the same way as in `parse_command_line`)

Example (using the options defined in the top-level docs of this module):

```
port = 80
mysql_host = 'mydb.example.com:3306'
# Both lists and comma-separated strings are allowed for
# multiple=True.
memcache_hosts = ['cache1.example.com:11011',
                  'cache2.example.com:11011']
memcache_hosts = 'cache1.example.com:11011,cache2.example.com:11011'
```

If `final` is `False`, parse callbacks will not be run. This is useful for applications that wish to combine configurations from multiple sources.

---

**Note:** `tornado.options` is primarily a command-line library. Config file support is provided for applications that wish to use it, but applications that prefer config files may wish to look at other libraries instead.

---

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.

Changed in version 5.1: Added the ability to set options via strings in config files.

`OptionParser.print_help` (*file: TextIO = None*) → None

Prints all the command line options to stderr (or another file).

`OptionParser.add_parse_callback` (*callback: Callable[[], None]*) → None

Adds a parse callback, to be invoked when option parsing is done.

`OptionParser.mockable` () → `tornado.options._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
```

`OptionParser.items()` → `Iterable[Tuple[str, Any]]`  
 An iterable of (name, value) pairs.

New in version 3.1.

`OptionParser.as_dict()` → `Dict[str, Any]`  
 The names and values of all options.

New in version 3.1.

`OptionParser.groups()` → `Set[str]`  
 The set of option-groups created by `define`.

New in version 3.1.

`OptionParser.group_dict(group: str)` → `Dict[str, Any]`  
 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.

## 5.7.5 tornado.testing — Unit testing support for asynchronous code

## 5.7.6 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`.

**exception** `tornado.util.TimeoutError`

Exception raised by `with_timeout` and `IOLoop.run_sync`.

Changed in version 5.0: Unified `tornado.gen.TimeoutError` and `tornado.ioloop.TimeoutError` as `tornado.util.TimeoutError`. Both former names remain as aliases.

**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: bytes, max\_length: int = 0*) → bytes

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()** → bytes

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: str) → Any`

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: BaseException) → Optional[int]`

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: str) → str`

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__`.

Changed in version 5.0: It is now possible for configuration to be specified at multiple levels of a class hierarchy.



**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()** → None

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: Callable, name: str*)

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: Sequence[Any], kwargs: Dict[str, Any], default: Any = None*) → Any

Returns the old value of the named argument without replacing it.

Returns *default* if the argument is not present.

**replace** (*new\_value: Any, args: Sequence[Any], kwargs: Dict[str, Any]*) → Tuple[Any, Sequence[Any], Dict[str, Any]]

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).

## 5.8 Frequently Asked Questions

- *Why isn't this example with `time.sleep()` running in parallel?*
- *My code is asynchronous, but it's not running in parallel in two browser tabs.*

### 5.8.1 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` (or `asyncio.sleep`) instead:

```
class CoroutineSleepHandler(RequestHandler):
    async def get(self):
        for i in range(5):
            print(i)
            await 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. Adapt the callback-based function into a future:

```
class CoroutineTimeoutHandler(RequestHandler):
    async def get(self):
        io_loop = IOLoop.current()
        for i in range(5):
            print(i)
            f = tornado.concurrent.Future()
            do_something_with_callback(f.set_result)
            result = await f
```

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:

```
class ThreadPoolHandler(RequestHandler):
    async def get(self):
        for i in range(5):
            print(i)
            await IOLoop.current().run_in_executor(None, time.sleep, 1)
```

See the *Asynchronous I/O* chapter of the Tornado user's guide for more on blocking and asynchronous functions.

## 5.8.2 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.

## 5.9 Release notes

### 5.9.1 What's new in Tornado 5.1.1

Sep 16, 2018

#### Bug fixes

- Fixed an case in which the `Future` returned by `RequestHandler.finish` could fail to resolve.
- The `TwitterMixin.authenticate_redirect` method works again.
- Improved error handling in the `tornado.auth` module, fixing hanging requests when a network or other error occurs.

### 5.9.2 What's new in Tornado 5.1

July 12, 2018

#### Deprecation notice

- Tornado 6.0 will drop support for Python 2.7 and 3.4. The minimum supported Python version will be 3.5.2.
- The `tornado.stack_context` module is deprecated and will be removed in Tornado 6.0. The reason for this is that it is not feasible to provide this module's semantics in the presence of `async def` native coroutines. `ExceptionStackContext` is mainly obsolete thanks to coroutines. `StackContext` lacks a direct replacement although the new `contextvars` package (in the Python standard library beginning in Python 3.7) may be an alternative.
- Callback-oriented code often relies on `ExceptionStackContext` to handle errors and prevent leaked connections. In order to avoid the risk of silently introducing subtle leaks (and to consolidate all of Tornado's interfaces behind the coroutine pattern), `callback` arguments throughout the package are deprecated and will be removed in version 6.0. All functions that had a `callback` argument removed now return a `Future` which should be used instead.
- Where possible, deprecation warnings are emitted when any of these deprecated interfaces is used. However, Python does not display deprecation warnings by default. To prepare your application for Tornado 6.0, run Python with the `-Wd` argument or set the environment variable `PYTHONWARNINGS` to `d`. If your application runs on Python 3 without deprecation warnings, it should be able to move to Tornado 6.0 without disruption.

#### `tornado.auth`

- `OAuthMixin._oauth_get_user_future` may now be a native coroutine.

- All callback arguments in this package are deprecated and will be removed in 6.0. Use the coroutine interfaces instead.
- The `OAuthMixin._oauth_get_user` method is deprecated and will be removed in 6.0. Override `_oauth_get_user_future` instead.

### `tornado.autoreload`

- The command-line autoreload wrapper is now preserved if an internal autoreload fires.
- The command-line wrapper no longer starts duplicated processes on windows when combined with internal autoreload.

### `tornado.concurrent`

- `run_on_executor` now returns `Future` objects that are compatible with `await`.
- The callback argument to `run_on_executor` is deprecated and will be removed in 6.0.
- `return_future` is deprecated and will be removed in 6.0.

### `tornado.gen`

- Some older portions of this module are deprecated and will be removed in 6.0. This includes `engine`, `YieldPoint`, `Callback`, `Wait`, `WaitAll`, `MultiYieldPoint`, and `Task`.
- Functions decorated with `@gen.coroutine` will no longer accept callback arguments in 6.0.

### `tornado.httpclient`

- The behavior of `raise_error=False` is changing in 6.0. Currently it suppresses all errors; in 6.0 it will only suppress the errors raised due to completed responses with non-200 status codes.
- The callback argument to `AsyncHTTPClient.fetch` is deprecated and will be removed in 6.0.
- `tornado.httpclient.HTTPError` has been renamed to `HTTPClientError` to avoid ambiguity in code that also has to deal with `tornado.web.HTTPError`. The old name remains as an alias.
- `tornado.curl_httpclient` now supports non-ASCII characters in username and password arguments.
- `.HTTPResponse.request_time` now behaves consistently across `simple_httpclient` and `curl_httpclient`, excluding time spent in the `max_clients` queue in both cases (previously this time was included in `simple_httpclient` but excluded in `curl_httpclient`). In both cases the time is now computed using a monotonic clock where available.
- `HTTPResponse` now has a `start_time` attribute recording a wall-clock (`time.time`) timestamp at which the request started (after leaving the `max_clients` queue if applicable).

### `tornado.httplib`

- `parse_multipart_form_data` now recognizes non-ASCII filenames in RFC 2231/5987 (`filename*`) format.
- `HTTPServerRequest.write` is deprecated and will be removed in 6.0. Use the methods of `request.connection` instead.

- Malformed HTTP headers are now logged less noisily.

#### `tornado.ioloop`

- `PeriodicCallback` now supports a `jitter` argument to randomly vary the timeout.
- `IOLoop.set_blocking_signal_threshold`, `IOLoop.set_blocking_log_threshold`, `IOLoop.log_stack`, and `IOLoop.handle_callback_exception` are deprecated and will be removed in 6.0.
- Fixed a `KeyError` in `IOLoop.close` when `IOLoop` objects are being opened and closed in multiple threads.

#### `tornado.iostream`

- All callback arguments in this module are deprecated except for `BaseIOStream.set_close_callback`. They will be removed in 6.0.
- `streaming_callback` arguments to `BaseIOStream.read_bytes` and `BaseIOStream.read_until_close` are deprecated and will be removed in 6.0.

#### `tornado.netutil`

- Improved compatibility with GNU Hurd.

#### `tornado.options`

- `tornado.options.parse_config_file` now allows setting options to strings (which will be parsed the same way as `tornado.options.parse_command_line`) in addition to the specified type for the option.

#### `tornado.platform.twisted`

- `TornadoReactor` and `TwistedIOLoop` are deprecated and will be removed in 6.0. Instead, Tornado will always use the `asyncio` event loop and `twisted` can be configured to do so as well.

#### `tornado.stack_context`

- The `tornado.stack_context` module is deprecated and will be removed in 6.0.

#### `tornado.testing`

- `AsyncHTTPTestCase.fetch` now takes a `raise_error` argument. This argument has the same semantics as `AsyncHTTPClient.fetch`, but defaults to `false` because tests often need to deal with non-200 responses (and for backwards-compatibility).
- The `AsyncTestCase.stop` and `AsyncTestCase.wait` methods are deprecated.

### `tornado.web`

- New method `RequestHandler.detach` can be used from methods that are not decorated with `@asynchronous` (the decorator was required to use `self.request.connection.detach()`).
- `RequestHandler.finish` and `RequestHandler.render` now return `Futures` that can be used to wait for the last part of the response to be sent to the client.
- `FallbackHandler` now calls `on_finish` for the benefit of subclasses that may have overridden it.
- The `asynchronous` decorator is deprecated and will be removed in 6.0.
- The `callback` argument to `RequestHandler.flush` is deprecated and will be removed in 6.0.

### `tornado.websocket`

- When compression is enabled, memory limits now apply to the post-decompression size of the data, protecting against DoS attacks.
- `websocket_connect` now supports subprotocols.
- `WebSocketHandler` and `WebSocketClientConnection` now have `selected_subprotocol` attributes to see the subprotocol in use.
- The `WebSocketHandler.select_subprotocol` method is now called with an empty list instead of a list containing an empty string if no subprotocols were requested by the client.
- `WebSocketHandler.open` may now be a coroutine.
- The `data` argument to `WebSocketHandler.ping` is now optional.
- Client-side websocket connections no longer buffer more than one message in memory at a time.
- Exception logging now uses `RequestHandler.log_exception`.

### `tornado.wsgi`

- `WSGIApplication` and `WSGIAdapter` are deprecated and will be removed in Tornado 6.0.

## 5.9.3 What's new in Tornado 5.0.2

Apr 7, 2018

### Bug fixes

- Fixed a memory leak when `IOLoop` objects are created and destroyed.
- If `AsyncTestCase.get_new_ioloop` returns a reference to a preexisting event loop (typically when it has been overridden to return `IOLoop.current()`), the test's `tearDown` method will not close this loop.
- Fixed a confusing error message when the synchronous `HTTPClient` fails to initialize because an event loop is already running.
- `PeriodicCallback` no longer executes twice in a row due to backwards clock adjustments.

## 5.9.4 What's new in Tornado 5.0.1

Mar 18, 2018

### Bug fix

- This release restores support for versions of Python 3.4 prior to 3.4.4. This is important for compatibility with Debian Jessie which has 3.4.2 as its version of Python 3.

## 5.9.5 What's new in Tornado 5.0

Mar 5, 2018

### Highlights

- The focus of this release is improving integration with `asyncio`. On Python 3, the `IOLoop` is always a wrapper around the `asyncio` event loop, and `asyncio.Future` and `asyncio.Task` are used instead of their Tornado counterparts. This means that libraries based on `asyncio` can be mixed relatively seamlessly with those using Tornado. While care has been taken to minimize the disruption from this change, code changes may be required for compatibility with Tornado 5.0, as detailed in the following section.
- Tornado 5.0 supports Python 2.7.9+ and 3.4+. Python 2.7 and 3.4 are deprecated and support for them will be removed in Tornado 6.0, which will require Python 3.5+.

### Backwards-compatibility notes

- Python 3.3 is no longer supported.
- Versions of Python 2.7 that predate the `ssl` module update are no longer supported. (The `ssl` module was updated in version 2.7.9, although in some distributions the updates are present in builds with a lower version number. Tornado requires `ssl.SSLContext`, `ssl.create_default_context`, and `ssl.match_hostname`)
- Versions of Python 3.5 prior to 3.5.2 are no longer supported due to a change in the async iterator protocol in that version.
- The `trollius` project (`asyncio` backported to Python 2) is no longer supported.
- `tornado.concurrent.Future` is now an alias for `asyncio.Future` when running on Python 3. This results in a number of minor behavioral changes:
  - `Future` objects can only be created while there is a current `IOLoop`
  - The timing of callbacks scheduled with `Future.add_done_callback` has changed. `tornado.concurrent.future_add_done_callback` can be used to make the behavior more like older versions of Tornado (but not identical). Some of these changes are also present in the Python 2 version of `tornado.concurrent.Future` to minimize the difference between Python 2 and 3.
  - Cancellation is now partially supported, via `asyncio.Future.cancel`. A canceled `Future` can no longer have its result set. Applications that handle `Future` objects directly may want to use `tornado.concurrent.future_set_result_unless_cancelled`. In native coroutines, cancellation will cause an exception to be raised in the coroutine.
  - The `exc_info` and `set_exc_info` methods are no longer present. Use `tornado.concurrent.future_set_exc_info` to replace the latter, and raise the exception with `result` to replace the former.

- `io_loop` arguments to many Tornado functions have been removed. Use `IOLoop.current()` instead of passing `IOLoop` objects explicitly.
- On Python 3, `IOLoop` is always a wrapper around the `asyncio` event loop. `IOLoop.configure` is effectively removed on Python 3 (for compatibility, it may be called to redundantly specify the `asyncio`-backed `IOLoop`)
- `IOLoop.instance` is now a deprecated alias for `IOLoop.current`. Applications that need the cross-thread communication behavior facilitated by `IOLoop.instance` should use their own global variable instead.

### Other notes

- The futures (`concurrent.futures` backport) package is now required on Python 2.7.
- The `certifi` and `backports.ssl-match-hostname` packages are no longer required on Python 2.7.
- Python 3.6 or higher is recommended, because it features more efficient garbage collection of `asyncio.Future` objects.

### `tornado.auth`

- `GoogleOAuth2Mixin` now uses a newer set of URLs.

### `tornado.autoreload`

- On Python 3, uses `__main__.__spec__` to more reliably reconstruct the original command line and avoid modifying `PYTHONPATH`.
- The `io_loop` argument to `tornado.autoreload.start` has been removed.

### `tornado.concurrent`

- `tornado.concurrent.Future` is now an alias for `asyncio.Future` when running on Python 3. See “Backwards-compatibility notes” for more.
- Setting the result of a `Future` no longer blocks while callbacks are being run. Instead, the callbacks are scheduled on the next `IOLoop` iteration.
- The deprecated alias `tornado.concurrent.TracebackFuture` has been removed.
- `tornado.concurrent.chain_future` now works with all three kinds of Futures (`Tornado`, `asyncio`, and `concurrent.futures`)
- The `io_loop` argument to `tornado.concurrent.run_on_executor` has been removed.
- New functions `future_set_result_unless_cancelled`, `future_set_exc_info`, and `future_add_done_callback` help mask the difference between `asyncio.Future` and Tornado’s previous `Future` implementation.

### `tornado.curl_httpclient`

- Improved debug logging on Python 3.



- The `time_info` response attribute now includes `appconnect` in addition to other measurements.
- Closing a `CurlAsyncHTTPClient` now breaks circular references that could delay garbage collection.
- The `io_loop` argument to the `CurlAsyncHTTPClient` constructor has been removed.

#### `tornado.gen`

- `tornado.gen.TimeoutError` is now an alias for `tornado.util.TimeoutError`.
- Leak detection for `Futures` created by this module now attributes them to their proper caller instead of the coroutine machinery.
- Several circular references that could delay garbage collection have been broken up.
- On Python 3, `asyncio.Task` is used instead of the Tornado coroutine runner. This improves compatibility with some `asyncio` libraries and adds support for cancellation.
- The `io_loop` arguments to `YieldFuture` and `with_timeout` have been removed.

#### `tornado.httpclient`

- The `io_loop` argument to all `AsyncHTTPClient` constructors has been removed.

#### `tornado.httpserver`

- It is now possible for a client to reuse a connection after sending a chunked request.
- If a client sends a malformed request, the server now responds with a 400 error instead of simply closing the connection.
- `Content-Length` and `Transfer-Encoding` headers are no longer sent with 1xx or 204 responses (this was already true of 304 responses).
- When closing a connection to a HTTP/1.1 client, the `Connection: close` header is sent with the response.
- The `io_loop` argument to the `HTTPServer` constructor has been removed.
- If more than one `X-Scheme` or `X-Forwarded-Proto` header is present, only the last is used.

#### `tornado.httputil`

- The string representation of `HTTPServerRequest` objects (which are sometimes used in log messages) no longer includes the request headers.
- New function `qs_to_qs1` converts the result of `urllib.parse.parse_qs` to name-value pairs.

#### `tornado.ioloop`

- `tornado.ioloop.TimeoutError` is now an alias for `tornado.util.TimeoutError`.
- `IOLoop.instance` is now a deprecated alias for `IOLoop.current`.
- `IOLoop.install` and `IOLoop.clear_instance` are deprecated.
- The `IOLoop.initialized` method has been removed.

- On Python 3, the `asyncio`-backed `IOLoop` is always used and alternative `IOLoop` implementations cannot be configured. `IOLoop.current` and related methods pass through to `asyncio.get_event_loop`.
- `run_sync` cancels its argument on a timeout. This results in better stack traces (and avoids log messages about leaks) in native coroutines.
- New methods `IOLoop.run_in_executor` and `IOLoop.set_default_executor` make it easier to run functions in other threads from native coroutines (since `concurrent.futures.Future` does not support `await`).
- `PollIOLoop` (the default on Python 2) attempts to detect misuse of `IOLoop` instances across `os.fork`.
- The `io_loop` argument to `PeriodicCallback` has been removed.
- It is now possible to create a `PeriodicCallback` in one thread and start it in another without passing an explicit event loop.
- The `IOLoop.set_blocking_signal_threshold` and `IOLoop.set_blocking_log_threshold` methods are deprecated because they are not implemented for the `asyncio` event loop. Use the `PYTHONASYNCIODEBUG=1` environment variable instead.
- `IOLoop.clear_current` now works if it is called before any current loop is established.

### `tornado.iostream`

- The `io_loop` argument to the `IOStream` constructor has been removed.
- New method `BaseIOStream.read_into` provides a minimal-copy alternative to `BaseIOStream.read_bytes`.
- `BaseIOStream.write` is now much more efficient for very large amounts of data.
- Fixed some cases in which `IOStream.error` could be inaccurate.
- Writing a `memoryview` can no longer result in “`BufferError: Existing exports of data: object cannot be resized`”.

### `tornado.locks`

- As a side effect of the `Future` changes, waiters are always notified asynchronously with respect to `Condition.notify`.

### `tornado.netutil`

- The default `Resolver` now uses `IOLoop.run_in_executor`. `ExecutorResolver`, `BlockingResolver`, and `ThreadedResolver` are deprecated.
- The `io_loop` arguments to `add_accept_handler`, `ExecutorResolver`, and `ThreadedResolver` have been removed.
- `add_accept_handler` returns a callable which can be used to remove all handlers that were added.
- `OverrideResolver` now accepts per-family overrides.

### `tornado.options`

- Duplicate option names are now detected properly whether they use hyphens or underscores.

### `tornado.platform.asyncio`

- `AsyncIOLoop` and `AsyncIOMainLoop` are now used automatically when appropriate; referencing them explicitly is no longer recommended.
- Starting an `IOLoop` or making it current now also sets the `asyncio` event loop for the current thread. Closing an `IOLoop` closes the corresponding `asyncio` event loop.
- `to_tornado_future` and `to_asyncio_future` are deprecated since they are now no-ops.
- `AnyThreadEventLoopPolicy` can now be used to easily allow the creation of event loops on any thread (similar to Tornado's prior policy).

### `tornado.platform.caresresolver`

- The `io_loop` argument to `CaresResolver` has been removed.

### `tornado.platform.twisted`

- The `io_loop` arguments to `TornadoReactor`, `TwistedResolver`, and `tornado.platform.twisted.install` have been removed.

### `tornado.process`

- The `io_loop` argument to the `Subprocess` constructor and `Subprocess.initialize` has been removed.

### `tornado.routing`

- A default 404 response is now generated if no delegate is found for a request.

### `tornado.simple_httpclient`

- The `io_loop` argument to `SimpleAsyncHTTPClient` has been removed.
- TLS is now configured according to `ssl.create_default_context` by default.

### `tornado.tcpclient`

- The `io_loop` argument to the `TCPClient` constructor has been removed.
- `TCPClient.connect` has a new `timeout` argument.

### `tornado.tcpserver`

- The `io_loop` argument to the `TCPServer` constructor has been removed.
- `TCPServer` no longer logs `EBADF` errors during shutdown.

### `tornado.testing`

- The deprecated `tornado.testing.get_unused_port` and `tornado.testing.LogTrapTestCase` have been removed.
- `AsyncHTTPTestCase.fetch` now supports absolute URLs.
- `AsyncHTTPTestCase.fetch` now connects to `127.0.0.1` instead of `localhost` to be more robust against faulty ipv6 configurations.

### `tornado.util`

- `tornado.util.TimeoutError` replaces `tornado.gen.TimeoutError` and `tornado.ioloop.TimeoutError`.
- `Configurable` now supports configuration at multiple levels of an inheritance hierarchy.

### `tornado.web`

- `RequestHandler.set_status` no longer requires that the given status code appear in `http.client.responses`.
- It is no longer allowed to send a body with 1xx or 204 responses.
- Exception handling now breaks up reference cycles that could delay garbage collection.
- `RedirectHandler` now copies any query arguments from the request to the redirect location.
- If both `If-None-Match` and `If-Modified-Since` headers are present in a request to `StaticFileHandler`, the latter is now ignored.

### `tornado.websocket`

- The C accelerator now operates on multiple bytes at a time to improve performance.
- Requests with invalid websocket headers now get a response with status code 400 instead of a closed connection.
- `WebSocketHandler.write_message` now raises `WebSocketClosedError` if the connection closes while the write is in progress.
- The `io_loop` argument to `websocket_connect` has been removed.

## 5.9.6 What's new in Tornado 4.5.3

Jan 6, 2018

### `tornado.curl_httpclient`

- Improved debug logging on Python 3.

### `tornado.httpserver`

- Content-Length and Transfer-Encoding headers are no longer sent with 1xx or 204 responses (this was already true of 304 responses).
- Reading chunked requests no longer leaves the connection in a broken state.

### `tornado.iostream`

- Writing a `memoryview` can no longer result in “BufferError: Existing exports of data: object cannot be resized”.

### `tornado.options`

- Duplicate option names are now detected properly whether they use hyphens or underscores.

### `tornado.testing`

- `AsyncHTTPTestCase.fetch` now uses `127.0.0.1` instead of `localhost`, improving compatibility with systems that have partially-working ipv6 stacks.

### `tornado.web`

- It is no longer allowed to send a body with 1xx or 204 responses.

### `tornado.websocket`

- Requests with invalid websocket headers now get a response with status code 400 instead of a closed connection.

## 5.9.7 What’s new in Tornado 4.5.2

Aug 27, 2017

### Bug Fixes

- Tornado now sets the `FD_CLOEXEC` flag on all file descriptors it creates. This prevents hanging client connections and resource leaks when the `tornado.autoreload` module (or `Application(debug=True)`) is used.

## 5.9.8 What’s new in Tornado 4.5.1

Apr 20, 2017

### `tornado.log`

- Improved detection of libraries for colorized logging.

## `tornado.httputil`

- `url_concat` once again treats `None` as equivalent to an empty sequence.

## 5.9.9 What's new in Tornado 4.5

Apr 16, 2017

### Backwards-compatibility warning

- The `tornado.websocket` module now imposes a limit on the size of incoming messages, which defaults to 10MiB.

### New module

- `tornado.routing` provides a more flexible routing system than the one built in to `Application`.

### General changes

- Reduced the number of circular references, reducing memory usage and improving performance.

## `tornado.auth`

- The `tornado.auth` module has been updated for compatibility with a [change to Facebook's access\\_token endpoint](#). This includes both the changes initially released in Tornado 4.4.3 and an additional change to support the `session_expires` field in the new format. The `session_expires` field is currently a string; it should be accessed as `int(user['session_expires'])` because it will change from a string to an int in Tornado 5.0.

## `tornado.autoreload`

- Autoreload is now compatible with the `asyncio` event loop.
- Autoreload no longer attempts to close the `IOLoop` and all registered file descriptors before restarting; it relies on the `CLOEXEC` flag being set instead.

## `tornado.concurrent`

- Suppressed some “‘NoneType’ object not callable” messages that could be logged at shutdown.

## `tornado.gen`

- `yield None` is now equivalent to `yield gen.moment`. `moment` is deprecated. This improves compatibility with `asyncio`.
- Fixed an issue in which a generator object could be garbage collected prematurely (most often when weak references are used).

- New function `is_coroutine_function` identifies functions wrapped by `coroutine` or `engine`.

#### `tornado.http1connection`

- The `Transfer-Encoding` header is now parsed case-insensitively.

#### `tornado.httpclient`

- `SimpleAsyncHTTPClient` now follows 308 redirects.
- `CurlAsyncHTTPClient` will no longer accept protocols other than `http` and `https`. To override this, set `pycurl.PROTOCOLS` and `pycurl.REDIR_PROTOCOLS` in a `prepare_curl_callback`.
- `CurlAsyncHTTPClient` now supports digest authentication for proxies (in addition to basic auth) via the new `proxy_auth_mode` argument.
- The minimum supported version of `libcurl` is now `7.22.0`.

#### `tornado.httpserver`

- `HTTPServer` now accepts the keyword argument `trusted_downstream` which controls the parsing of `X-Forwarded-For` headers. This header may be a list or set of IP addresses of trusted proxies which will be skipped in the `X-Forwarded-For` list.
- The `no_keep_alive` argument works again.

#### `tornado.httputil`

- `url_concat` correctly handles fragments and existing query arguments.

#### `tornado.ioloop`

- Fixed 100% CPU usage after a callback returns an empty list or dict.
- `IOLoop.add_callback` now uses a lockless implementation which makes it safe for use from `__del__` methods. This improves performance of calls to `add_callback` from the `IOLoop` thread, and slightly decreases it for calls from other threads.

#### `tornado.iostream`

- `memoryview` objects are now permitted as arguments to `write`.
- The internal memory buffers used by `IOStream` now use `bytearray` instead of a list of `bytes`, improving performance.
- Futures returned by `write` are no longer orphaned if a second call to `write` occurs before the previous one is finished.

### `tornado.log`

- Colored log output is now supported on Windows if the `colorama` library is installed and the application calls `colorama.init()` at startup.
- The signature of the `LogFormatter` constructor has been changed to make it compatible with `logging.config.dictConfig`.

### `tornado.netutil`

- Worked around an issue that caused “LookupError: unknown encoding: latin1” errors on Solaris.

### `tornado.process`

- `Subprocess` no longer causes “subprocess still running” warnings on Python 3.6.
- Improved error handling in `cpu_count`.

### `tornado.tcpclient`

- `TCPClient` now supports a `source_ip` and `source_port` argument.
- Improved error handling for environments where IPv6 support is incomplete.

### `tornado.tcpserver`

- `TCPServer.handle_stream` implementations may now be native coroutines.
- Stopping a `TCPServer` twice no longer raises an exception.

### `tornado.web`

- `RedirectHandler` now supports substituting parts of the matched URL into the redirect location using `str.format` syntax.
- New methods `RequestHandler.render_linked_js`, `RequestHandler.render_embed_js`, `RequestHandler.render_linked_css`, and `RequestHandler.render_embed_css` can be overridden to customize the output of `UIModule`.

### `tornado.websocket`

- `WebSocketHandler.on_message` implementations may now be coroutines. New messages will not be processed until the previous `on_message` coroutine has finished.
- The `websocket_ping_interval` and `websocket_ping_timeout` application settings can now be used to enable a periodic ping of the websocket connection, allowing dropped connections to be detected and closed.
- The new `websocket_max_message_size` setting defaults to 10MiB. The connection will be closed if messages larger than this are received.



- Headers set by `RequestHandler.prepare` or `RequestHandler.set_default_headers` are now sent as a part of the websocket handshake.
- Return values from `WebSocketHandler.get_compression_options` may now include the keys `compression_level` and `mem_level` to set gzip parameters. The default compression level is now 6 instead of 9.

## Demos

- A new file upload demo is available in the `file_upload` directory.
- A new `TCPClient` and `TCPServer` demo is available in the `tcpecho` directory.
- Minor updates have been made to several existing demos, including updates to more recent versions of jquery.

## Credits

The following people contributed commits to this release:

- A. Jesse Jiryu Davis
- Aaron Opfer
- Akihiro Yamazaki
- Alexander
- Andreas Røsdal
- Andrew Rabert
- Andrew Sumin
- Antoine Pietri
- Antoine Pitrou
- Artur Stawiarski
- Ben Darnell
- Brian Mego
- Dario
- Doug Vargas
- Eugene Dubovoy
- Iver Jordal
- JZQT
- James Maier
- Jeff Hunter
- Leynos
- Mark Henderson
- Michael V. DePalatis
- Min RK

- Mircea Ulinic
- Ping
- Ping Yang
- Riccardo Magliocchetti
- Samuel Chen
- Samuel Dion-Girardeau
- Scott Meisburger
- Shawn Ding
- TaoBeier
- Thomas Kluyver
- Vadim Semenov
- matee
- mike820324
- stiletto
- zhimin
- 

### 5.9.10 What's new in Tornado 4.4.3

Mar 30, 2017

#### Bug fixes

- The `tornado.auth` module has been updated for compatibility with a change to Facebook's `access_token` endpoint.

### 5.9.11 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).

### 5.9.12 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.

### 5.9.13 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.

## 5.9.14 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`

- `HTTPHeader`s 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 `xsrp_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.

### 5.9.15 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.

### 5.9.16 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`

- `xml_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`

- `HTTPHeader` 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.

## 5.9.17 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 `Deferred` objects in Tornado coroutines.

### Backwards-compatibility notes

- `HTTPServer` now calls `start_request` with the correct arguments. This change is backwards-incompatible, affecting 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, affecting 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`

- `TCPCClient` 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_request`.

### `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()`.

## 5.9.18 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)

## 5.9.19 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.

## 5.9.20 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 accept 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.

## 5.9.21 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 `xsrp_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 `xsrp_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.

## 5.9.22 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.

## 5.9.23 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 `IOLoop.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 `tornado.web.URLSpec` class.
- Application now accepts 4-tuples to specify the name parameter (which previously required constructing a `tornado.web.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.

## 5.9.24 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.

## 5.9.25 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: `HTTPHeader` 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 `HTTPHeader`s 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 bracket 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 `tornado.web.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.

## 5.9.26 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.

## 5.9.27 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.

### 5.9.28 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 suppress 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)

**5.9.29 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.

**5.9.30 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.

## 5.9.31 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 `tornado.web.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 `TypeError`s

### 5.9.32 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).

### 5.9.33 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 performance 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.

## 5.9.34 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.
- `IStream.connect` now reports errors more consistently via logging and the close callback (this affects e.g. connections to localhost on FreeBSD).
- `IStream.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`.

## 5.9.35 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

## 5.9.36 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. `xsrif_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`.

(continues on next page)

(continued from previous page)

- \* 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.

### 5.9.37 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.

### 5.9.38 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](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)

(continues on next page)

(continued from previous page)

## 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 in `AsyncHTTPTestCase`
- \* `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

(continues on next page)

(continued from previous page)

- \* `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

## 5.9.39 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>

## 5.9.40 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

## 5.9.41 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.

## 5.9.42 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

(continues on next page)

(continued from previous page)

```
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 6

---

### 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](#).

Tornado is available under the [Apache License, Version 2.0](#).

This web site and all documentation is licensed under [Creative Commons 3.0](#).



**t**

tornado.curl\_httpclient, 45  
tornado.escape, 41  
tornado.httputil, 46  
tornado.locale, 43  
tornado.log, 54  
tornado.options, 55  
tornado.platform.caresresolver, 54  
tornado.template, 37  
tornado.util, 59  
tornado.wsgi, 53



**A**

add() (tornado.httputil.HTTPHeaders method), 46  
 add\_parse\_callback() (in module tornado.options), 57  
 add\_parse\_callback() (tornado.options.OptionParser method), 58  
 ArgReplacer (class in tornado.util), 61  
 arguments (tornado.httputil.HTTPServerRequest attribute), 47  
 as\_dict() (tornado.options.OptionParser method), 59

**B**

BaseLoader (class in tornado.template), 40  
 body (tornado.httputil.HTTPServerRequest attribute), 47  
 body\_arguments (tornado.httputil.HTTPServerRequest attribute), 48

**C**

CaresResolver (class in tornado.platform.caresresolver), 54  
 code (tornado.httputil.ResponseStartLine attribute), 51  
 Configurable (class in tornado.util), 60  
 configurable\_base() (tornado.util.Configurable class method), 60  
 configurable\_default() (tornado.util.Configurable class method), 61  
 configure() (tornado.util.Configurable class method), 61  
 configured\_class() (tornado.util.Configurable class method), 61  
 connection (tornado.httputil.HTTPServerRequest attribute), 48  
 cookies (tornado.httputil.HTTPServerRequest attribute), 48  
 CSVLocale (class in tornado.locale), 45  
 CurlAsyncHTTPClient (class in tornado.curl\_httpclient), 45

**D**

data\_received() (tornado.httputil.HTTPMessageDelegate method), 49

decompress() (tornado.util.GzipDecompressor method), 59  
 define() (in module tornado.options), 56  
 define() (tornado.options.OptionParser method), 57  
 define\_logging\_options() (in module tornado.log), 55  
 DictLoader (class in tornado.template), 40

**E**

enable\_pretty\_logging() (in module tornado.log), 55  
 encode\_username\_password() (in module tornado.httputil), 51  
 environ() (tornado.wsgi.WSGIContainer static method), 53  
 errno\_from\_exception() (in module tornado.util), 60  
 Error, 57

**F**

files (tornado.httputil.HTTPServerRequest attribute), 48  
 filter\_whitespace() (in module tornado.template), 40  
 finish() (tornado.httputil.HTTPConnection method), 50  
 finish() (tornado.httputil.HTTPMessageDelegate method), 49  
 flush() (tornado.util.GzipDecompressor method), 60  
 format\_date() (tornado.locale.Locale method), 44  
 format\_day() (tornado.locale.Locale method), 45  
 format\_timestamp() (in module tornado.httputil), 51  
 friendly\_number() (tornado.locale.Locale method), 45  
 full\_url() (tornado.httputil.HTTPServerRequest method), 48

**G**

generate() (tornado.template.Template method), 40  
 get() (in module tornado.locale), 43  
 get() (tornado.locale.Locale class method), 44  
 get\_all() (tornado.httputil.HTTPHeaders method), 46  
 get\_closest() (tornado.locale.Locale class method), 44  
 get\_list() (tornado.httputil.HTTPHeaders method), 46  
 get\_old\_value() (tornado.util.ArgReplacer method), 61

`get_ssl_certificate()` (tornado.httputil.HTTPServerRequest method), 48

`get_supported_locales()` (in module tornado.locale), 44

`GettextLocale` (class in tornado.locale), 45

`group_dict()` (tornado.options.OptionParser method), 59

`groups()` (tornado.options.OptionParser method), 59

`GzipDecompressor` (class in tornado.util), 59

## H

`headers` (tornado.httputil.HTTPServerRequest attribute), 47

`headers_received()` (tornado.httputil.HTTPMessageDelegate method), 49

`host` (tornado.httputil.HTTPServerRequest attribute), 47

`HTTPConnection` (class in tornado.httputil), 49

`HTTPFile` (class in tornado.httputil), 50

`HTTPHeaders` (class in tornado.httputil), 46

`HTTPInputError`, 48

`HTTPMessageDelegate` (class in tornado.httputil), 49

`HTTPOutputError`, 48

`HTTPServerConnectionDelegate` (class in tornado.httputil), 48

`HTTPServerRequest` (class in tornado.httputil), 47

## I

`import_object()` (in module tornado.util), 60

`initialize()` (tornado.util.Configurable method), 61

`items()` (tornado.options.OptionParser method), 58

## J

`json_decode()` (in module tornado.escape), 41

`json_encode()` (in module tornado.escape), 41

## L

`linkify()` (in module tornado.escape), 42

`list()` (tornado.locale.Locale method), 45

`load()` (tornado.template.BaseLoader method), 40

`load_gettext_translations()` (in module tornado.locale), 44

`load_translations()` (in module tornado.locale), 43

`Loader` (class in tornado.template), 40

`Locale` (class in tornado.locale), 44

`LogFormatter` (class in tornado.log), 54

## M

`method` (tornado.httputil.HTTPServerRequest attribute), 47

`method` (tornado.httputil.RequestStartLine attribute), 51

`mockable()` (tornado.options.OptionParser method), 58

## N

`native_str()` (in module tornado.escape), 42

## O

`ObjectDict` (class in tornado.util), 59

`on_close()` (tornado.httputil.HTTPServerConnectionDelegate method), 49

`on_connection_close()` (tornado.httputil.HTTPMessageDelegate method), 49

`OptionParser` (class in tornado.options), 57

`options` (in module tornado.options), 56

## P

`parse()` (tornado.httputil.HTTPHeaders class method), 46

`parse_body_arguments()` (in module tornado.httputil), 50

`parse_command_line()` (in module tornado.options), 57

`parse_command_line()` (tornado.options.OptionParser method), 57

`parse_config_file()` (in module tornado.options), 57

`parse_config_file()` (tornado.options.OptionParser method), 58

`parse_cookie()` (in module tornado.httputil), 52

`parse_line()` (tornado.httputil.HTTPHeaders method), 46

`parse_multipart_form_data()` (in module tornado.httputil), 50

`parse_request_start_line()` (in module tornado.httputil), 51

`parse_response_start_line()` (in module tornado.httputil), 51

`ParseError`, 40

`path` (tornado.httputil.HTTPServerRequest attribute), 47

`path` (tornado.httputil.RequestStartLine attribute), 51

`pgettext()` (tornado.locale.GettextLocale method), 45

`print_help()` (in module tornado.options), 57

`print_help()` (tornado.options.OptionParser method), 58

`protocol` (tornado.httputil.HTTPServerRequest attribute), 47

## Q

`qs_to_qs()` (in module tornado.httputil), 52

`query` (tornado.httputil.HTTPServerRequest attribute), 47

`query_arguments` (tornado.httputil.HTTPServerRequest attribute), 48

## R

`re_unescape()` (in module tornado.util), 60

`reason` (tornado.httputil.ResponseStartLine attribute), 51

`recursive_unicode()` (in module tornado.escape), 42

`remote_ip` (tornado.httputil.HTTPServerRequest attribute), 47

`replace()` (tornado.util.ArgReplacer method), 61

`request_time()` (tornado.httputil.HTTPServerRequest method), 48

`RequestStartLine` (class in tornado.httputil), 51

`reset()` (tornado.template.BaseLoader method), 40

resolve\_path() (tornado.template.BaseLoader method),  
40

ResponseStartLine (class in tornado.httputil), 51

## S

set\_default\_locale() (in module tornado.locale), 43

split\_host\_and\_port() (in module tornado.httputil), 51

squeeze() (in module tornado.escape), 43

start\_request() (tornado.httputil.HTTPServerConnectionDelegate  
method), 48

## T

Template (class in tornado.template), 39

timedelta\_to\_seconds() (in module tornado.util), 61

TimeoutError, 59

to\_basestring() (in module tornado.escape), 42

to\_unicode() (in module tornado.escape), 42

tornado.curl\_httpclient (module), 45

tornado.escape (module), 41

tornado.httputil (module), 46

tornado.locale (module), 43

tornado.log (module), 54

tornado.options (module), 55

tornado.platform.caresresolver (module), 54

tornado.template (module), 37

tornado.util (module), 59

tornado.wsgi (module), 53

translate() (tornado.locale.Locale method), 44

## U

unconsumed\_tail (tornado.util.GzipDecompressor at-  
tribute), 60

uri (tornado.httputil.HTTPServerRequest attribute), 47

url\_concat() (in module tornado.httputil), 50

url\_escape() (in module tornado.escape), 41

url\_unescape() (in module tornado.escape), 41

utf8() (in module tornado.escape), 42

## V

version (tornado.httputil.HTTPServerRequest attribute),  
47

version (tornado.httputil.RequestStartLine attribute), 51

version (tornado.httputil.ResponseStartLine attribute), 51

## W

write() (tornado.httputil.HTTPConnection method), 50

write\_headers() (tornado.httputil.HTTPConnection  
method), 49

WSGIContainer (class in tornado.wsgi), 53

## X

xhtml\_escape() (in module tornado.escape), 41

xhtml\_unescape() (in module tornado.escape), 41