Tornado Documentation

Release 3.1

Facebook

May 24, 2017

Contents

Overview

FriendFeed's web server is a relatively simple, non-blocking web server written in Python. The FriendFeed application is written using a web framework that looks a bit like web.py or Google's webapp, but with additional tools and optimizations to take advantage of the non-blocking web server and tools.

Tornado is an open source version of this web server and some of the tools we use most often at FriendFeed. The framework is distinct from most mainstream web server frameworks (and certainly most Python frameworks) because it is non-blocking and reasonably fast. Because it is non-blocking and uses epoll or kqueue, it can handle thousands of simultaneous standing connections, which means the framework is ideal for real-time web services. We built the web server specifically to handle FriendFeed's real-time features — every active user of FriendFeed maintains an open connection to the FriendFeed servers. (For more information on scaling servers to support thousands of clients, see The C10K problem.)

Here is the canonical "Hello, world" example app:

```python
import tornado.ioloop
import tornado.web

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

application = tornado.web.Application([
    (r"/", MainHandler),
])

if __name__ == "__main__":
    application.listen(8888)
    tornado.ioloop.IOLoop.instance().start()
```

We attempted to clean up the code base to reduce interdependencies between modules, so you should (theoretically) be able to use any of the modules independently in your project without using the whole package.

Request handlers and request arguments

A Tornado web application maps URLs or URL patterns to subclasses of `tornado.web.RequestHandler`. Those classes define `get()` or `post()` methods to handle HTTP `GET` or `POST` requests to that URL.

This code maps the root URL `/` to `MainHandler` and the URL pattern `/story/([0-9]+)` to `StoryHandler`. Regular expression groups are passed as arguments to the `RequestHandler` methods:

```
class MainHandler(tornado.web.RequestHandler):
    def get(self):
        self.write("You requested the main page")

class StoryHandler(tornado.web.RequestHandler):
    def get(self, story_id):
        self.write("You requested the story " + story_id)

application = tornado.web.Application([
    (r"/", MainHandler),
    (r"/story/([0-9]+)", StoryHandler),
])
```

You can get query string arguments and parse `POST` bodies with the `get_argument()` method:

```
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_argument("message"))
```

Uploaded files 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":...}`.

If you want to send an error response to the client, e.g., 403 Unauthorized, you can just raise a `tornado.web.HTTPError` exception:

```
if not self.user_is_logged_in():
    raise tornado.web.HTTPError(403)
```

The request handler can access the object representing the current request with `self.request`. The `HTTPRequest` object includes a number of useful attributes, including:

- `arguments` - all of the `GET` and `POST` arguments
- `files` - all of the uploaded files (via `multipart/form-data` POST requests)
- `path` - the request path (everything before the `?`)
- `headers` - the request headers

See the class definition for `tornado.httpserver.HTTPRequest` for a complete list of attributes.

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 keyword arguments from the `Application` configuration. (the `initialize` method is new in Tornado 1.1; in older versions subclasses would override __init__ instead). `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 `send_error`, etc), processing stops here.

4. One of the HTTP methods is called: `get()`, `post()`, `put()`, etc. If the URL regular expression contains capturing groups, they are passed as arguments to this method.

5. When the request is finished, `on_finish()` is called. For synchronous handlers this is immediately after `get()` (etc) return; for asynchronous handlers it is after the call to `finish()`.

Here is an example demonstrating the `initialize()` method:

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

    def get(self, username):
        ...

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

Other methods designed for overriding include:

- `write_error(self, status_code, exc_info=None, **kwargs)` - outputs HTML for use on error pages.

- `get_current_user(self)` - see *User Authentication* below

- `get_user_locale(self)` - returns `locale` object to use for the current user

- `get_login_url(self)` - returns login url to be used by the `@authenticated` decorator (default is in `Application` settings)

- `get_template_path(self)` - returns location of template files (default is in `Application` settings)

- `set_default_headers(self)` - may be used to set additional headers on the response (such as a custom `Server` header)

Error Handling

There are three ways to return an error from a `RequestHandler`:

1. Manually call `set_status` and output the response body normally.

2. Call `send_error`. This discards any pending unflushed output and calls `write_error` to generate an error page.

3. Raise an exception. *tornado.web.HTTPError* can be used to generate a specified status code; all other exceptions return a 500 status. The exception handler uses *send_error* and *write_error* to generate the error page.

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*. 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).

In Tornado 2.0 and earlier, custom error pages were implemented by overriding RequestHandler.get_error_html, which returned the error page as a string instead of calling the normal output methods (and had slightly different semantics for exceptions). This method is still supported, but it is deprecated and applications are encouraged to switch to *RequestHandler.write_error*.

Redirection

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

You can use `self.redirect` within a `RequestHandler` method (like `get`) to redirect users elsewhere. There is also an optional parameter `permanent` which you can use to indicate that the redirection is considered permanent.

This triggers a `301 Moved Permanently` HTTP status, which is useful for e.g. redirecting to a canonical URL for a page in an SEO-friendly manner.

The default value of `permanent` is `False`, which is apt for things like redirecting users on successful POST requests.

```
self.redirect('/some-canonical-page', permanent=True)
```

`RedirectHandler` is available for your use when you initialize `Application`.

For example, notice how we redirect to a longer download URL on this website:

```
application = tornado.wsgi.WSGIApplication([
    (r"/([a-z]*)", ContentHandler),
    (r"/static/tornado-0.2.tar.gz", tornado.web.RedirectHandler,
     dict(url="https://github.com/downloads/facebook/tornado/tornado-0.2.tar.gz")),
], **settings)
```

The default `RedirectHandler` status code is `301 Moved Permanently`, but to use `302 Found` instead, set `permanent` to `False`.

```
application = tornado.wsgi.WSGIApplication([
    (r"/foo", tornado.web.RedirectHandler, {"url":"/bar", "permanent":False}),
], **settings)
```

Note that the default value of `permanent` is different in `self.redirect` than in `RedirectHandler`. This should make some sense if you consider that `self.redirect` is used in your methods and is probably invoked by logic involving environment, authentication, or form submission, but `RedirectHandler` patterns are going to fire 100% of the time they match the request URL.

Templates

You can use any template language supported by Python, but Tornado ships with its own templating language that is a lot faster and more flexible than many of the most popular templating systems out there. See the `tornado.template` module documentation for complete documentation.

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 surronded 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 `template` module directly outside of a `RequestHandler` many of these entries are not present).

- escape: alias for `tornado.escape.xhtml_escape`
- xhtml_escape: alias for `tornado.escape.xhtml_escape`
- url_escape: alias for `tornado.escape.url_escape`
- json_encode: alias for `tornado.escape.json_encode`
- squeeze: alias for `tornado.escape.squeeze`
- linkify: alias for `tornado.escape.linkify`
- datetime: the Python `datetime` module
- handler: the current `RequestHandler` object
- request: alias for `handler.request`
- current_user: alias for `handler.current_user`

- `locale`: alias for `handler.locale`

- `_`: alias for `handler.locale.translate`

- `static_url`: alias for `handler.static_url`

- `xsrf_form_html`: alias for `handler.xsrf_form_html`

- `reverse_url`: alias for `Application.reverse_url`

- All entries from the `ui_methods` and `ui_modules` `Application` settings

- Any keyword arguments passed to `render` or `render_string`

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

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 easily forged by malicious clients. If you need to set cookies to, e.g., save the user ID of the currently logged in user, you need to sign your cookies to prevent forgery. Tornado supports this out of the box 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!")
```

User authentication

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

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

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

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

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

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

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

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

```
class MainHandler(BaseHandler):
    @tornado.web.authenticated
    def get(self):
```

```
        name = tornado.escape.xhtml_escape(self.current_user)
        self.write("Hello, " + name)

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

If you decorate `post()` methods with the `authenticated` decorator, and the user is not logged in, the server will send a `403` response.

Tornado comes with built-in support for third-party authentication schemes like Google OAuth. See the *tornado.* *auth* for more details. Check out the Tornado Blog example application for a complete example that uses authentication (and stores user data in a MySQL database).

Cross-site request forgery protection

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

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

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

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

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

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

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

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

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

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

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

Static files and aggressive file caching

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

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

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

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

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

---

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

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

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

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

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

In production, you probably want to serve static files from a more optimized static file server like nginx. You can configure most any web server to support these caching semantics. Here is the nginx configuration we use at FriendFeed:

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

Localization

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 localized 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 `get_user_locale` in your request handler:

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

    def get_user_locale(self):
        if "locale" not in self.current_user.prefs:
            # Use the Accept-Language header
            return None
        return self.current_user.prefs["locale"]
```

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

You can load all the translations for your application using the `tornado.locale.load_translations` method. It takes in the name of the directory which should contain CSV files named after the locales whose translations they contain, e.g., `es_GT.csv` or `fr_CA.csv`. The method loads all the translations from those CSV files and infers the list of supported locales based on the presence of each CSV file. You typically call this method once in the `main()` method of your server:

```
def main():
    tornado.locale.load_translations(
        os.path.join(os.path.dirname(__file__), "translations"))
    start_server()
```

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.

See the *`tornado.locale`* documentation for detailed information on the CSV format and other localization methods.

UI modules

Tornado supports *UI modules* to make it easy to support standard, reusable UI widgets across your application. UI modules are like special functional 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:

```
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 `home.html`, you reference the `Entry` module rather than printing the HTML directly:

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

Within `entry.html`, you reference the `Entry` module with the `show_comments` argument to show the expanded form of the entry:

```
{% 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.

Non-blocking, asynchronous requests

When a request handler is executed, the request is automatically finished. Since Tornado uses a non-blocking I/O style, you can override this default behavior if you want a request to remain open after the main request handler method returns using the `tornado.web.asynchronous` decorator.

When you use this decorator, it is your responsibility to call `self.finish()` to finish the HTTP request, or the user's browser will simply hang:

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

Here is a real example that makes a call to the FriendFeed API using Tornado's built-in asynchronous HTTP client:

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

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

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

For 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).

Asynchronous HTTP clients

Tornado includes two non-blocking HTTP client implementations: `SimpleAsyncHTTPClient` and `CurlAsyncHTTPClient`. The simple client has no external dependencies because it is implemented directly on

top of Tornado's `IOLoop`. The Curl client requires that `libcurl` and `pycurl` be installed (and a recent version of each is highly recommended to avoid bugs in older version's asynchronous interfaces), but is more likely to be compatible with sites that exercise little-used parts of the HTTP specification.

Each of these clients is available in its own module (`tornado.simple_httpclient` and `tornado.curl_httpclient`), as well as via a configurable alias in `tornado.httpclient`. `SimpleAsyncHTTPClient` is the default, but to use a different implementation call the `AsyncHTTPClient.configure` method at startup:

```
AsyncHTTPClient.configure('tornado.curl_httpclient.CurlAsyncHTTPClient')
```

Third party authentication

Tornado's `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 GoogleHandler(tornado.web.RequestHandler, tornado.auth.GoogleMixin):
    @tornado.web.asynchronous
    def get(self):
        if self.get_argument("openid.mode", None):
            self.get_authenticated_user(self._on_auth)
            return
        self.authenticate_redirect()

    def _on_auth(self, user):
        if not user:
            self.authenticate_redirect()
            return
        # Save the user with, e.g., set_secure_cookie()
```

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

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:

- 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.

- Templates will not be cached, nor will static file hashes (used by the `static_url` function)

- When an exception in a `RequestHandler` is not caught, an error page including a stack trace will be generated.

Debug mode is not compatible with `HTTPServer`'s multi-process mode. You must not give `HTTPServer.start` an argument other than 1 (or call *tornado.process.fork_processes*) if you are using debug 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 `debug=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.

Running Tornado in production

At FriendFeed, we use nginx as a load balancer and static file server. We run multiple instances of the Tornado web server on multiple frontend machines. We typically run one Tornado frontend per core on the machine (sometimes more depending on utilization).

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

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

```
user nginx;
worker_processes 1;

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

events {
    worker_connections 1024;
    use epoll;
}

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

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

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

    keepalive_timeout 65;
    proxy_read_timeout 200;
    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    gzip on;
    gzip_min_length 1000;
    gzip_proxied any;
```

```
    gzip_types text/plain text/html text/css text/xml
               application/x-javascript application/xml
               application/atom+xml text/javascript;

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

    server {
        listen 80;

        # Allow file uploads
        client_max_body_size 50M;

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

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

WSGI and Google AppEngine

Tornado comes with limited support for WSGI. However, since WSGI does not support non-blocking requests, you cannot use any of the asynchronous/non-blocking features of Tornado in your application if you choose to use WSGI instead of Tornado's HTTP server. Some of the features that are not available in WSGI applications: `@tornado.web.asynchronous`, the `httpclient` module, and the `auth` module.

You can create a valid WSGI application from your Tornado request handlers by using `WSGIApplication` in the `wsgi` module instead of using `tornado.web.Application`. Here is an example that uses the built-in WSGI `CGIHandler` to make a valid Google AppEngine application:

```
import tornado.web
import tornado.wsgi
import wsgiref.handlers


class MainHandler(tornado.web.RequestHandler):
```

```
    def get(self):
        self.write("Hello, world")

if __name__ == "__main__":
    application = tornado.wsgi.WSGIApplication([
        (r"/", MainHandler),
    ])
    wsgiref.handlers.CGIHandler().run(application)
```

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

Core web framework

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

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

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

```python
import tornado.ioloop
import tornado.web

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

if __name__ == "__main__":
    application = tornado.web.Application([
        (r"/", MainHandler),
    ])
    application.listen(8888)
    tornado.ioloop.IOLoop.instance().start()
```

See the *Tornado overview* for more details and a good getting started guide.

Thread-safety notes

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

Request handlers

class `tornado.web.`**`RequestHandler`**(*application*, *request*, ****kwargs***)
>   Subclass this class and define *get()* or *post()* to make a handler.
>
>   If you want to support more methods than the standard GET/HEAD/POST, you should override the class variable `SUPPORTED_METHODS` in your *RequestHandler* subclass.

Entry points

`RequestHandler.`**`initialize`**()
>   Hook for subclass initialization.
>
>   A dictionary passed as the third argument of a url spec will be supplied as keyword arguments to initialize().
>
>   Example:

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

    def get(self, username):
        ...

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

`RequestHandler.`**`prepare`**()
>   Called at the beginning of a request before *get*/*post*/etc.
>
>   Override this method to perform common initialization regardless of the request method.
>
>   Asynchronous support: Decorate this method with *gen.coroutine* or *return_future* to make it asynchronous (the *asynchronous* decorator cannot be used on *prepare*). If this method returns a *Future* execution will not proceed until the *Future* is done.
>
>   New in version 3.1: Asynchronous support.

`RequestHandler.`**`on_finish`**()
>   Called after the end of a request.
>
>   Override this method to perform cleanup, logging, etc. This method is a counterpart to *prepare*. on_finish may not produce any output, as it is called after the response has been sent to the client.

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

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

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

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

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

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

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

Input

`RequestHandler.`**`get_argument`** (*name*, *default=[]*, *strip=True*)
> Returns the value of the argument with the given name.
>
> If default is not provided, the argument is considered to be required, and we raise a *MissingArgumentError* if it is missing.
>
> If the argument appears in the url more than once, we return the last value.
>
> The returned value is always unicode.

`RequestHandler.`**`get_arguments`** (*name*, *strip=True*)
> Returns a list of the arguments with the given name.
>
> If the argument is not present, returns an empty list.
>
> The returned values are always unicode.

`RequestHandler.`**`decode_argument`** (*value*, *name=None*)
> Decodes an argument from the request.
>
> The argument has been percent-decoded and is now a byte string. By default, this method decodes the argument as utf-8 and returns a unicode string, but this may be overridden in subclasses.
>
> This method is used as a filter for both *get_argument()* and for values extracted from the url and passed to *get()*/*post()*/etc.
>
> The name of the argument is provided if known, but may be None (e.g. for unnamed groups in the url regex).

`RequestHandler.`**`request`**
> The *tornado.httpserver.HTTPRequest* object containing additional request parameters including e.g. headers and body data.

`RequestHandler.`**`path_args`**

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

Output

`RequestHandler.`**`set_status`** (*status_code*, *reason=None*)
> Sets the status code for our response.
>
> > **Parameters**
> >
> > * **status_code** (*int*) – Response status code. If `reason` is `None`, it must be present in `httplib.responses`.
> >
> > * **reason** (*string*) – Human-readable reason phrase describing the status code. If `None`, it will be filled in from `httplib.responses`.

`RequestHandler.`**`set_header`** (*name*, *value*)
> Sets the given response header name and value.
>
> If a datetime is given, we automatically format it according to the HTTP specification. If the value is not a string, we convert it to a string. All header values are then encoded as UTF-8.

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

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

RequestHandler.**clear_header**(*name*)
> Clears an outgoing header, undoing a previous *set_header* call.
>
> Note that this method does not apply to multi-valued headers set by *add_header*.

RequestHandler.**set_default_headers**()
> Override this to set HTTP headers at the beginning of the request.
>
> For example, this is the place to set a custom Server header. Note that setting such headers in the normal flow of request processing may not do what you want, since headers may be reset during error handling.

RequestHandler.**write**(*chunk*)
> Writes the given chunk to the output buffer.
>
> To write the output to the network, use the flush() method below.
>
> If the given chunk is a dictionary, we write it as JSON and set the Content-Type of the response to be application/json. (if you want to send JSON as a different Content-Type, call set_header *after* calling write()).
>
> Note that lists are not converted to JSON because of a potential cross-site security vulnerability. All JSON output should be wrapped in a dictionary. More details at http://haacked.com/archive/2008/11/20/anatomy-of-a-subtle-json-vulnerability.aspx

RequestHandler.**flush**(*include_footers=False*, *callback=None*)
> Flushes the current output buffer to the network.
>
> The callback argument, if given, can be used for flow control: it will be run when all flushed data has been written to the socket. Note that only one flush callback can be outstanding at a time; if another flush occurs before the previous flush's callback has been run, the previous callback will be discarded.

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

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

RequestHandler.**render_string**(*template_name*, ***kwargs*)
> Generate the given template with the given arguments.
>
> We return the generated byte string (in utf8). To generate and write a template as a response, use render() above.

RequestHandler.**get_template_namespace**()
> Returns a dictionary to be used as the default template namespace.
>
> May be overridden by subclasses to add or modify values.
>
> The results of this method will be combined with additional defaults in the *tornado.template* module and keyword arguments to *render* or *render_string*.

RequestHandler.**redirect**(*url*, *permanent=False*, *status=None*)
> Sends a redirect to the given (optionally relative) URL.
>
> If the status argument is specified, that value is used as the HTTP status code; otherwise either 301 (permanent) or 302 (temporary) is chosen based on the permanent argument. The default is 302 (temporary).

RequestHandler.**send_error**(*status_code=500*, ***kwargs*)
> Sends the given HTTP error code to the browser.
>
> If *flush()* has already been called, it is not possible to send an error, so this method will simply terminate the response. If output has been written but not yet flushed, it will be discarded and replaced with the error page.

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

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

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

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

> For historical reasons, if a method get_error_html exists, it will be used instead of the default write_error implementation. get_error_html returned a string instead of producing output normally, and had different semantics for exception handling. Users of get_error_html are encouraged to convert their code to override write_error instead.

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

Cookies

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

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

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

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

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

`RequestHandler.`**`clear_all_cookies`**()
> Deletes all the cookies the user sent with this request.

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

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

`RequestHandler.`**`set_secure_cookie`**(*name*, *value*, *expires_days=30*, ***kwargs*)
> Signs and timestamps a cookie so it cannot be forged.

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

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

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

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

`RequestHandler.`**`create_signed_value`**(*name*, *value*)
> Signs and timestamps a string so it cannot be forged.

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

Other

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

`RequestHandler.`**`async_callback`**(*callback*, **args*, ***kwargs*)
  Obsolete - catches exceptions from the wrapped function.

  This function is unnecessary since Tornado 1.1.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    New in version 3.1.

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

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

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

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

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

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

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

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

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

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

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

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

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

    See *check_xsrf_cookie()* above for more information.

Application configuration

class `tornado.web.`**`Application`**(*handlers=None*, *default_host=''*, *transforms=None*, *wsgi=False*,
                                        ***settings*)

A collection of request handlers that make up a web application.

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

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

The constructor for this class takes in a list of *URLSpec* objects or (regexp, request_class) tuples. When we receive requests, we iterate over the list in order and instantiate an instance of the first request class whose regexp matches the request path.

Each tuple can contain an optional third element, which should be a dictionary if it is present. That dictionary is passed as keyword arguments to the contructor of the handler. This pattern is used for the *StaticFileHandler* in this example (note that a *StaticFileHandler* can be installed automatically with the static_path setting described below):

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

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

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

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

`settings`

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

General settings:

- •`debug`: If `True` the application runs in debug mode, described in *Debug mode and automatic reloading*.

- •`gzip`: If `True`, responses in textual formats will be gzipped automatically.

- •`log_function`: This function will be called at the end of every request to log the result (with one argument, the *RequestHandler* object). The default implementation writes to the `logging` module's root logger. May also be customized by overriding *Application.log_request*.

- ui_modules and ui_methods: May be set to a mapping of *UIModule* or UI methods to be made available to templates. May be set to a module, dictionary, or a list of modules and/or dicts. See *UI modules* for more details.

Authentication and security settings:

- cookie_secret: Used by *RequestHandler.get_secure_cookie* and *set_secure_cookie* to sign cookies.

- login_url: The *authenticated* decorator will redirect to this url if the user is not logged in. Can be further customized by overriding *RequestHandler.get_login_url*

- xsrf_cookies: If true, *Cross-site request forgery protection* will be enabled.

- twitter_consumer_key, twitter_consumer_secret, friendfeed_consumer_key, friendfeed_consumer_secret, google_consumer_key, google_consumer_secret, facebook_api_key, facebook_secret: Used in the *tornado.auth* module to authenticate to various APIs.

Template settings:

- autoescape: Controls automatic escaping for templates. May be set to None to disable escaping, or to the *name* of a function that all output should be passed through. Defaults to "xhtml_escape". Can be changed on a per-template basis with the {% autoescape %} directive.

- template_path: Directory containing template files. Can be further customized by overriding *RequestHandler.get_template_path*

- template_loader: Assign to an instance of *tornado.template.BaseLoader* to customize template loading. If this setting is used the template_path and autoescape settings are ignored. Can be further customized by overriding *RequestHandler.create_template_loader*.

Static file settings:

- static_path: Directory from which static files will be served.

- static_url_prefix: Url prefix for static files, defaults to "/static/".

- static_handler_class, static_handler_args: May be set to use a different handler for static files instead of the default *tornado.web.StaticFileHandler*. static_handler_args, if set, should be a dictionary of keyword arguments to be passed to the handler's initialize method.

listen (*port*, *address=''*, ***kwargs*)
Starts an HTTP server for this application on the given port.

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

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

add_handlers (*host_pattern*, *host_handlers*)
Appends the given handlers to our handler list.

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

reverse_url (*name*, **args*)
Returns a URL path for handler named name

> The handler must be added to the application as a named *URLSpec*.
>
> Args will be substituted for capturing groups in the *URLSpec* regex. They will be converted to strings if necessary, encoded as utf8, and url-escaped.

> **log_request** (*handler*)
> > Writes a completed HTTP request to the logs.
> >
> > By default writes to the python root logger. To change this behavior either subclass Application and override this method, or pass a function in the application settings dictionary as log_function.

class tornado.web.**URLSpec** (*pattern*, *handler_class*, *kwargs=None*, *name=None*)
> Specifies mappings between URLs and handlers.
>
> Parameters:
>
> > • pattern: Regular expression to be matched. Any groups in the regex will be passed in to the handler's get/post/etc methods as arguments.
> >
> > • handler_class: *RequestHandler* subclass to be invoked.
> >
> > • kwargs (optional): A dictionary of additional arguments to be passed to the handler's constructor.
> >
> > • name (optional): A name for this handler. Used by *Application.reverse_url*.
>
> The URLSpec class is also available under the name tornado.web.url.

Decorators

tornado.web.**asynchronous** (*method*)
> Wrap request handler methods with this if they are asynchronous.
>
> This decorator is unnecessary if the method is also decorated with @gen.coroutine (it is legal but unnecessary to use the two decorators together, in which case @asynchronous must be first).
>
> This decorator should only be applied to the *HTTP verb methods*; its behavior is undefined for any other method. This decorator does not *make* a method asynchronous; it tells the framework that the method *is* asynchronous. For this decorator to be useful the method must (at least sometimes) do something asynchronous.
>
> If this decorator is given, the response is not finished when the method returns. It is up to the request handler to call *self.finish()* to finish the HTTP request. Without this decorator, the request is automatically finished when the get() or post() method returns. Example:

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

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

> New in version 3.1: The ability to use @gen.coroutine without @asynchronous.

tornado.web.**authenticated** (*method*)
> Decorate methods with this to require that the user be logged in.
>
> If the user is not logged in, they will be redirected to the configured *login url*.

`tornado.web.`**`addslash`**(*method*)
> Use this decorator to add a missing trailing slash to the request path.

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

`tornado.web.`**`removeslash`**(*method*)
> Use this decorator to remove trailing slashes from the request path.

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

Everything else

exception `tornado.web.`**`HTTPError`**(*status_code*, *log_message=None*, **args*, ***kwargs*)
> An exception that will turn into an HTTP error response.

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

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

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

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

> New in version 3.1.

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

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

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

> **`embedded_javascript`**()
>> Returns a JavaScript string that will be embedded in the page.

> **`javascript_files`**()
>> Returns a list of JavaScript files required by this module.

> **`embedded_css`**()
>> Returns a CSS string that will be embedded in the page.

> **`css_files`**()
>> Returns a list of CSS files required by this module.

> **html_head** ()
> > Returns a CSS string that will be put in the <head/> element
>
> **html_body** ()
> > Returns an HTML string that will be put in the <body/> element
>
> **render_string** (*path*, ***kwargs*)
> > Renders a template and returns it as a string.

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

class tornado.web.**FallbackHandler** (*application*, *request*, ***kwargs*)
> A *RequestHandler* that wraps another HTTP server callback.
>
> The fallback is a callable object that accepts an *HTTPRequest*, such as an *Application* or *tornado.wsgi.WSGIContainer*. This is most useful to use both Tornado RequestHandlers and WSGI in the same server. Typical usage:

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

class tornado.web.**RedirectHandler** (*application*, *request*, ***kwargs*)
> Redirects the client to the given URL for all GET requests.
>
> You should provide the keyword argument url to the handler, e.g.:

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

class tornado.web.**StaticFileHandler** (*application*, *request*, ***kwargs*)
> A simple handler that can serve static content from a directory.
>
> A *StaticFileHandler* is configured automatically if you pass the static_path keyword argument to *Application*. This handler can be customized with the static_url_prefix, static_handler_class, and static_handler_args settings.
>
> To map an additional path to this handler for a static data directory you would add a line to your application like:

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

> The handler constructor requires a path argument, which specifies the local root directory of the content to be served.
>
> Note that a capture group in the regex is required to parse the value for the path argument to the get() method (different than the constructor argument above); see *URLSpec* for details.
>
> To maximize the effectiveness of browser caching, this class supports versioned urls (by default using the argument ?v=). If a version is given, we instruct the browser to cache this file indefinitely. *make_static_url* (also available as *RequestHandler.static_url*) can be used to construct a versioned url.
>
> This handler is intended primarily for use in development and light-duty file serving; for heavy traffic it will be more efficient to use a dedicated static file server (such as nginx or Apache). We support the HTTP Accept-Ranges mechanism to return partial content (because some browsers require this functionality to

be present to seek in HTML5 audio or video), but this handler should not be used with files that are too large to fit comfortably in memory.

Subclassing notes

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

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

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

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

compute_etag()
> Sets the `Etag` header based on static url version.
>
> This allows efficient `If-None-Match` checks against cached versions, and sends the correct `Etag` for a partial response (i.e. the same `Etag` as the full file).
>
> New in version 3.1.

set_headers()
> Sets the content and caching headers on the response.
>
> New in version 3.1.

should_return_304()
> Returns True if the headers indicate that we should return 304.
>
> New in version 3.1.

classmethod get_absolute_path(*root*, *path*)
> Returns the absolute location of `path` relative to `root`.
>
> `root` is the path configured for this *StaticFileHandler* (in most cases the `static_path` *Application* setting).
>
> This class method may be overridden in subclasses. By default it returns a filesystem path, but other strings may be used as long as they are unique and understood by the subclass's overridden *get_content*.
>
> New in version 3.1.

validate_absolute_path(*root*, *absolute_path*)
> Validate and return the absolute path.
>
> `root` is the configured path for the *StaticFileHandler*, and `path` is the result of *get_absolute_path*
>
> This is an instance method called during request processing, so it may raise *HTTPError* or use methods like *RequestHandler.redirect* (return None after redirecting to halt further processing). This is where 404 errors for missing files are generated.
>
> This method may modify the path before returning it, but note that any such modifications will not be understood by *make_static_url*.
>
> In instance methods, this method's result is available as `self.absolute_path`.
>
> New in version 3.1.

---

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

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

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

    New in version 3.1.

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

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

    New in version 3.1.

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

    This method may be overridden by subclasses. It will only be called if a partial result is requested from *get_content*

    New in version 3.1.

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

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

    New in version 3.1.

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

    New in version 3.1.

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

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

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

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

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

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

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

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

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

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

> This is the inverse of *make_static_url*.

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

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

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

tornado.httpserver — Non-blocking HTTP server

A non-blocking, single-threaded HTTP server.

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

This module also defines the *HTTPRequest* class which is exposed via *tornado.web.RequestHandler.request*.

HTTPRequest objects

class `tornado.httpserver.`**HTTPRequest**(*method*, *uri*, *version='HTTP/1.0'*, *headers=None*, *body=None*, *remote_ip=None*, *protocol=None*, *host=None*, *files=None*, *connection=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.

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 "connec-
>   tion" attribute. Since connections are typically kept open in HTTP/1.1, multiple requests can be handled
>   sequentially on a single connection.

supports_http_1_1()
>   Returns True if this request supports HTTP/1.1 semantics

cookies
>   A dictionary of Cookie.Morsel objects.

write(*chunk*, *callback=None*)
>   Writes the given chunk to the response stream.

finish()
>   Finishes this HTTP request on the open connection.

full_url()
>   Reconstructs the full URL for this request.

request_time()
>   Returns the amount of time it took for this request to execute.

get_ssl_certificate(*binary_form=False*)
>   Returns the client's SSL certificate, if any.
>
>   To use client certificates, the HTTPServer must have been constructed with cert_reqs set in ssl_options,
>   e.g.:

```
server = HTTPServer(app,
    ssl_options=dict(
        certfile="foo.crt",
        keyfile="foo.key",
        cert_reqs=ssl.CERT_REQUIRED,
        ca_certs="cacert.crt"))
```

>   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

---

HTTP Server

class `tornado.httpserver.`**`HTTPServer`**(*request_callback*,   *no_keep_alive=False*,   *io_loop=None*,
xheaders=False,   *ssl_options=None*,   *protocol=None*,
***kwargs*)

A non-blocking, single-threaded HTTP server.

A server is defined by a request callback that takes an HTTPRequest instance as an argument and writes a valid HTTP response with *HTTPRequest.write*. *HTTPRequest.finish* finishes the request (but does not necessarily close the connection in the case of HTTP/1.1 keep-alive requests). A simple example server that echoes back the URI you requested:

```
import tornado.httpserver
import tornado.ioloop

def handle_request(request):
   message = "You requested %s\n" % request.uri
   request.write("HTTP/1.1 200 OK\r\nContent-Length: %d\r\n\r\n%s" % (
                 len(message), message))
   request.finish()

http_server = tornado.httpserver.HTTPServer(handle_request)
http_server.listen(8888)
tornado.ioloop.IOLoop.instance().start()
```

HTTPServer is a very basic connection handler. It parses the request headers and body, but the request callback is responsible for producing the response exactly as it will appear on the wire. This affords maximum flexibility for applications to implement whatever parts of HTTP responses are required.

HTTPServer supports keep-alive connections by default (automatically for HTTP/1.1, or for HTTP/1.0 when the client requests `Connection:  keep-alive`). This means that the request callback must generate a properly-framed response, using either the `Content-Length` header or `Transfer-Encoding: chunked`. Applications that are unable to frame their responses properly should instead return a `Connection:  close` header in each response and pass `no_keep_alive=True` to the *HTTPServer* constructor.

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

To make this server serve SSL traffic, send the `ssl_options` dictionary argument with the arguments required for the `ssl.wrap_socket` method, including `certfile` and `keyfile`. (In Python 3.2+ you can pass an `ssl.SSLContext` object instead of a dict):

```
HTTPServer(applicaton, ssl_options={
    "certfile": os.path.join(data_dir, "mydomain.crt"),
    "keyfile": os.path.join(data_dir, "mydomain.key"),
})
```

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

1. *listen*: simple single-process:

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

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

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

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

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

3. `add_sockets`: advanced multi-process:

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

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

class tornado.httpserver.**HTTPConnection**(*stream*, *address*, *request_callback*, *no_keep_alive=False*, *xheaders=False*, *protocol=None*)

Handles a connection to an HTTP client, executing HTTP requests.

We parse HTTP headers and bodies, and execute the request callback until the HTTP conection is closed.

set_close_callback(*callback*)
Sets a callback that will be run when the connection is closed.

Use this instead of accessing `HTTPConnection.stream.set_close_callback` directly (which was the recommended approach prior to Tornado 3.0).

write(*chunk*, *callback=None*)
Writes a chunk of output to the stream.

finish()
Finishes the request.

`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, 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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

{% try %}...{% except %}...{% finally %}...{% else %}...{% 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.

Class reference

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

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

`generate`(***kwargs*)
Generate this template with the given arguments.

class `tornado.template.`**`BaseLoader`**(*autoescape='xhtml_escape'*, *namespace=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.

`autoescape` must be either None or a string naming a function in the template namespace, such as "xhtml_escape".

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

`reset`()
Resets the cache of compiled templates.

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

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

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

exception `tornado.template.`**`ParseError`**
Raised for template syntax errors.

`tornado.escape` — Escaping and string manipulation

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

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

Escaping functions

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

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

`tornado.escape.`**`url_escape`**(*value*, *plus=True*)
> Returns a URL-encoded version of the given value.
>
> If `plus` is true (the default), spaces will be represented as "+" instead of "%20". This is appropriate for query strings but not for the path component of a URL. Note that this default is the reverse of Python's urllib module.
>
> New in version 3.1: The `plus` argument

`tornado.escape.`**`url_unescape`**(*value*, *encoding='utf-8'*, *plus=True*)
> Decodes the given value from a URL.
>
> The argument may be either a byte or unicode string.
>
> If encoding is None, the result will be a byte string. Otherwise, the result is a unicode string in the specified encoding.
>
> If `plus` is true (the default), plus signs will be interpreted as spaces (literal plus signs must be represented as "%2B"). This is appropriate for query strings and form-encoded values but not for the path component of a URL. Note that this default is the reverse of Python's urllib module.
>
> New in version 3.1: The `plus` argument

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

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

Byte/unicode conversions

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

`tornado.escape.`**`utf8`**(*value*)
> Converts a string argument to a byte string.
>
> If the argument is already a byte string or None, it is returned unchanged. Otherwise it must be a unicode string and is encoded as utf8.

`tornado.escape.`**`to_unicode`**(*value*)
> Converts a string argument to a unicode string.
>
> If the argument is already a unicode string or None, it is returned unchanged. Otherwise it must be a byte string and is decoded as utf8.

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

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

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

tornado.escape.**recursive_unicode**(*obj*)

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

   Supports lists, tuples, and dictionaries.

Miscellaneous functions

tornado.escape.**linkify**(*text, shorten=False, extra_params='', require_protocol=False, permitted_protocols=['http', 'https']*)

   Converts plain text into HTML with links.

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

   Parameters:

   • `shorten`: Long urls will be shortened for display.

   • **`extra_params`: Extra text to include in the link tag, or a callable** taking the link as an argument and returning the extra text e.g. `linkify(text, extra_params='rel="nofollow" class="external"')`, or:

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

   • **`require_protocol`: Only linkify urls which include a protocol. If** this is False, urls such as www.facebook.com will also be linkified.

   • **`permitted_protocols`: List (or set) of protocols which should be** linkified, e.g. `linkify(text, permitted_protocols=["http", "ftp", "mailto"])`. It is very unsafe to include protocols such as `javascript`.

tornado.escape.**squeeze**(*value*)

   Replace all sequences of whitespace chars with a single space.

tornado.locale — Internationalization support

Translation methods for generating localized strings.

To load a locale and generate a translated string:

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

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

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

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

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

`tornado.locale.`**`get`**(**locale_codes*)
> Returns the closest match for the given locale codes.
>
> We iterate over all given locale codes in order. If we have a tight or a loose match for the code (e.g., "en" for "en_US"), we return the locale. Otherwise we move to the next code in the list.
>
> By default we return `en_US` if no translations are found for any of the specified locales. You can change the default locale with *set_default_locale()*.

`tornado.locale.`**`set_default_locale`**(*code*)
> Sets the default locale.
>
> The default locale is assumed to be the language used for all strings in the system. The translations loaded from disk are mappings from the default locale to the destination locale. Consequently, you don't need to create a translation file for the default locale.

`tornado.locale.`**`load_translations`**(*directory*)
> 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.
>
> 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"
```

`tornado.locale.`**`load_gettext_translations`**(*directory*, *domain*)
> Loads translations from `gettext`'s locale tree
>
> Locale tree is similar to system's `/usr/share/locale`, like:

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

> Three steps are required to have you app translated:
>
> > 1.Generate POT translation file:

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

2.Merge against existing POT file:

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

3.Compile:

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

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

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

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

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

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

        If it is not supported, we raise an exception.

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

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

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

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

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

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

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

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

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

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

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

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

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

Asynchronous networking

`tornado.gen` — Simplify asynchronous code

`tornado.gen` is a generator-based interface to make it easier to work in an asynchronous environment. Code using the `gen` module is technically asynchronous, but it is written as a single generator instead of a collection of separate functions.

For example, the following asynchronous handler:

```
class AsyncHandler(RequestHandler):
    @asynchronous
    def get(self):
        http_client = AsyncHTTPClient()
        http_client.fetch("http://example.com",
                          callback=self.on_fetch)

    def on_fetch(self, response):
        do_something_with_response(response)
        self.render("template.html")
```

could be written with `gen` as:

```
class GenAsyncHandler(RequestHandler):
    @gen.coroutine
    def get(self):
        http_client = AsyncHTTPClient()
        response = yield http_client.fetch("http://example.com")
        do_something_with_response(response)
        self.render("template.html")
```

Most asynchronous functions in Tornado return a `Future`; yielding this object returns its `result`.

For functions that do not return `Futures`, `Task` works with any function that takes a `callback` keyword argument (most Tornado functions can be used in either style, although the `Future` style is preferred since it is both shorter and provides better exception handling):

```
@gen.coroutine
def get(self):
    yield gen.Task(AsyncHTTPClient().fetch, "http://example.com")
```

You can also yield a list of `Futures` and/or `Tasks`, which will be started at the same time and run in parallel; a list of results will be returned when they are all finished:

```
@gen.coroutine
def get(self):
    http_client = AsyncHTTPClient()
    response1, response2 = yield [http_client.fetch(url1),
                                  http_client.fetch(url2)]
```

For more complicated interfaces, *Task* can be split into two parts: *Callback* and *Wait*:

```
class GenAsyncHandler2(RequestHandler):
    @asynchronous
    @gen.coroutine
    def get(self):
        http_client = AsyncHTTPClient()
        http_client.fetch("http://example.com",
                          callback=(yield gen.Callback("key")))
        response = yield gen.Wait("key")
        do_something_with_response(response)
        self.render("template.html")
```

The `key` argument to *Callback* and *Wait* allows for multiple asynchronous operations to be started at different times and proceed in parallel: yield several callbacks with different keys, then wait for them once all the async operations have started.

The result of a *Wait* or *Task* yield expression depends on how the callback was run. If it was called with no arguments, the result is `None`. If it was called with one argument, the result is that argument. If it was called with more than one argument or any keyword arguments, the result is an *Arguments* object, which is a named tuple `(args, kwargs)`.

Decorators

tornado.gen.**coroutine**(*func*)
   Decorator for asynchronous generators.

   Any generator that yields objects from this module must be wrapped in either this decorator or *engine*.

   Coroutines may "return" by raising the special exception *Return(value)*. In Python 3.3+, it is also possible for the function to simply use the `return value` statement (prior to Python 3.3 generators were not allowed to also return values). In all versions of Python a coroutine that simply wishes to exit early may use the `return` statement without a value.

   Functions with this decorator return a *Future*. Additionally, they may be called with a `callback` keyword argument, which will be invoked with the future's result when it resolves. If the coroutine fails, the callback will not be run and an exception will be raised into the surrounding *StackContext*. The `callback` argument is not visible inside the decorated function; it is handled by the decorator itself.

   From the caller's perspective, `@gen.coroutine` is similar to the combination of `@return_future` and `@gen.engine`.

tornado.gen.**engine**(*func*)
   Callback-oriented decorator for asynchronous generators.

This is an older interface; for new code that does not need to be compatible with versions of Tornado older than 3.0 the *coroutine* decorator is recommended instead.

This decorator is similar to *coroutine*, except it does not return a *Future* and the `callback` argument is not treated specially.

In most cases, functions decorated with *engine* should take a `callback` argument and invoke it with their result when they are finished. One notable exception is the *RequestHandler HTTP verb methods*, which use `self.finish()` in place of a callback argument.

Yield points

Instances of the following classes may be used in yield expressions in the generator. *Futures* may be yielded as well; their result method will be called automatically when they are ready. Additionally, lists of any combination of these objects may be yielded; the result is a list of the results of each yield point in the same order.

class `tornado.gen.`**`Task`**(*func*, **args*, ***kwargs*)
    Runs a single asynchronous operation.

    Takes a function (and optional additional arguments) and runs it with those arguments plus a `callback` keyword argument. The argument passed to the callback is returned as the result of the yield expression.

    A *Task* is equivalent to a *Callback*/*Wait* pair (with a unique key generated automatically):

```
result = yield gen.Task(func, args)

func(args, callback=(yield gen.Callback(key)))
result = yield gen.Wait(key)
```

class `tornado.gen.`**`Callback`**(*key*)
    Returns a callable object that will allow a matching *Wait* to proceed.

    The key may be any value suitable for use as a dictionary key, and is used to match Callbacks to their corresponding `Wait`s. The key must be unique among outstanding callbacks within a single run of the generator function, but may be reused across different runs of the same function (so constants generally work fine).

    The callback may be called with zero or one arguments; if an argument is given it will be returned by *Wait*.

class `tornado.gen.`**`Wait`**(*key*)
    Returns the argument passed to the result of a previous *Callback*.

class `tornado.gen.`**`WaitAll`**(*keys*)
    Returns the results of multiple previous *Callbacks*.

    The argument is a sequence of *Callback* keys, and the result is a list of results in the same order.

    WaitAll is equivalent to yielding a list of *Wait* objects.

class `tornado.gen.`**`YieldPoint`**
    Base class for objects that may be yielded from the generator.

    Applications do not normally need to use this class, but it may be subclassed to provide additional yielding behavior.

    `start`(*runner*)
        Called by the runner after the generator has yielded.

        No other methods will be called on this object before `start`.

    `is_ready`()
        Called by the runner to determine whether to resume the generator.

Returns a boolean; may be called more than once.

get_result()
Returns the value to use as the result of the yield expression.

This method will only be called once, and only after *is_ready* has returned true.

Other classes

exception `tornado.gen.`**`Return`**`(value=None)`
Special exception to return a value from a *coroutine*.

If this exception is raised, its value argument is used as the result of the coroutine:

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

In Python 3.3, this exception is no longer necessary: the `return` statement can be used directly to return a value (previously `yield` and `return` with a value could not be combined in the same function).

By analogy with the return statement, the value argument is optional, but it is never necessary to `raise gen.Return()`. The `return` statement can be used with no arguments instead.

class `tornado.gen.`**`Arguments`**
The result of a yield expression whose callback had more than one argument (or keyword arguments).

The *Arguments* object is a `collections.namedtuple` and can be used either as a tuple `(args, kwargs)` or an object with attributes `args` and `kwargs`.

`tornado.ioloop` — Main event loop

An I/O event loop for non-blocking sockets.

Typical applications will use a single *IOLoop* object, in the *IOLoop.instance* singleton. The *IOLoop.start* method should usually be called at the end of the `main()` function. Atypical applications may use more than one *IOLoop*, such as one *IOLoop* per thread, or per `unittest` case.

In addition to I/O events, the *IOLoop* can also schedule time-based events. *IOLoop.add_timeout* is a non-blocking alternative to `time.sleep`.

IOLoop objects

class `tornado.ioloop.`**`IOLoop`**
A level-triggered I/O loop.

We use `epoll` (Linux) or `kqueue` (BSD and Mac OS X) if they are available, or else we fall back on select(). If you are implementing a system that needs to handle thousands of simultaneous connections, you should use a system that supports either `epoll` or `kqueue`.

Example usage for a simple TCP server:

```
import errno
import functools
import ioloop
```

```
import socket

def connection_ready(sock, fd, events):
    while True:
        try:
            connection, address = sock.accept()
        except socket.error, e:
            if e.args[0] not in (errno.EWOULDBLOCK, errno.EAGAIN):
                raise
            return
        connection.setblocking(0)
        handle_connection(connection, address)

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.setblocking(0)
sock.bind(("", port))
sock.listen(128)

io_loop = ioloop.IOLoop.instance()
callback = functools.partial(connection_ready, sock)
io_loop.add_handler(sock.fileno(), callback, io_loop.READ)
io_loop.start()
```

Running an IOLoop

static IOLoop.**current**()
> Returns the current thread's *IOLoop*.

> If an *IOLoop* is currently running or has been marked as current by *make_current*, returns that instance.
> Otherwise returns *IOLoop.instance()*, i.e. the main thread's *IOLoop*.

> A common pattern for classes that depend on IOLoops is to use a default argument to enable programs with
> multiple IOLoops but not require the argument for simpler applications:

```
class MyClass(object):
    def __init__(self, io_loop=None):
        self.io_loop = io_loop or IOLoop.current()
```

> In general you should use *IOLoop.current* as the default when constructing an asynchronous object, and
> use *IOLoop.instance* when you mean to communicate to the main thread from a different one.

IOLoop.**make_current**()
> Makes this the *IOLoop* for the current thread.

> An *IOLoop* automatically becomes current for its thread when it is started, but it is sometimes useful to call
> *make_current* explictly before starting the *IOLoop*, so that code run at startup time can find the right
> instance.

static IOLoop.**instance**()
> Returns a global *IOLoop* instance.

> Most applications have a single, global *IOLoop* running on the main thread. Use this method to get this instance
> from another thread. To get the current thread's *IOLoop*, use *current()*.

static IOLoop.**initialized**()
> Returns true if the singleton instance has been created.

IOLoop.**install**()
> Installs this *IOLoop* object as the singleton instance.
>
> This is normally not necessary as *instance()* will create an *IOLoop* on demand, but you may want to call *install* to use a custom subclass of *IOLoop*.

IOLoop.**start**()
> Starts the I/O loop.
>
> The loop will run until one of the callbacks calls *stop()*, which will make the loop stop after the current event iteration completes.

IOLoop.**stop**()
> Stop the I/O loop.
>
> If the event loop is not currently running, the next call to *start()* will return immediately.
>
> To use asynchronous methods from otherwise-synchronous code (such as unit tests), you can start and stop the event loop like this:

```
ioloop = IOLoop()
async_method(ioloop=ioloop, callback=ioloop.stop)
ioloop.start()
```

> ioloop.start() will return after async_method has run its callback, whether that callback was invoked before or after ioloop.start.
>
> Note that even after *stop* has been called, the *IOLoop* is not completely stopped until *IOLoop.start* has also returned. Some work that was scheduled before the call to *stop* may still be run before the *IOLoop* shuts down.

IOLoop.**run_sync**(*func*, *timeout=None*)
> Starts the *IOLoop*, runs the given function, and stops the loop.
>
> If the function returns a *Future*, the *IOLoop* will run until the future is resolved. If it raises an exception, the *IOLoop* will stop and the exception will be re-raised to the caller.
>
> The keyword-only argument timeout may be used to set a maximum duration for the function. If the timeout expires, a TimeoutError is raised.
>
> This method is useful in conjunction with *tornado.gen.coroutine* to allow asynchronous calls in a main() function:

```
@gen.coroutine
def main():
    # do stuff...

if __name__ == '__main__':
    IOLoop.instance().run_sync(main)
```

IOLoop.**close**(*all_fds=False*)
> Closes the *IOLoop*, freeing any resources used.
>
> If all_fds is true, all file descriptors registered on the IOLoop will be closed (not just the ones created by the *IOLoop* itself).
>
> Many applications will only use a single *IOLoop* that runs for the entire lifetime of the process. In that case closing the *IOLoop* is not necessary since everything will be cleaned up when the process exits. *IOLoop.close* is provided mainly for scenarios such as unit tests, which create and destroy a large number of IOLoops.

An *IOLoop* must be completely stopped before it can be closed. This means that *IOLoop.stop()* must be called *and IOLoop.start()* must be allowed to return before attempting to call *IOLoop.close()*. Therefore the call to *close* will usually appear just after the call to *start* rather than near the call to *stop*.

Changed in version 3.1: If the *IOLoop* implementation supports non-integer objects for "file descriptors", those objects will have their close method when all_fds is true.

I/O events

IOLoop.**add_handler**(*fd*, *handler*, *events*)
> Registers the given handler to receive the given events for fd.
>
> The events argument is a bitwise or of the constants IOLoop.READ, IOLoop.WRITE, and IOLoop.ERROR.
>
> When an event occurs, handler(fd, events) will be run.

IOLoop.**update_handler**(*fd*, *events*)
> Changes the events we listen for fd.

IOLoop.**remove_handler**(*fd*)
> Stop listening for events on fd.

Callbacks and timeouts

IOLoop.**add_callback**(*callback*, **args*, ***kwargs*)
> Calls the given callback on the next I/O loop iteration.
>
> It is safe to call this method from any thread at any time, except from a signal handler. Note that this is the **only** method in *IOLoop* that makes this thread-safety guarantee; all other interaction with the *IOLoop* must be done from that *IOLoop*'s thread. *add_callback()* may be used to transfer control from other threads to the *IOLoop*'s thread.
>
> To add a callback from a signal handler, see *add_callback_from_signal*.

IOLoop.**add_callback_from_signal**(*callback*, **args*, ***kwargs*)
> Calls the given callback on the next I/O loop iteration.
>
> Safe for use from a Python signal handler; should not be used otherwise.
>
> Callbacks added with this method will be run without any *stack_context*, to avoid picking up the context of the function that was interrupted by the signal.

IOLoop.**add_future**(*future*, *callback*)
> Schedules a callback on the IOLoop when the given *Future* is finished.
>
> The callback is invoked with one argument, the *Future*.

IOLoop.**add_timeout**(*deadline*, *callback*)
> Runs the callback at the time deadline from the I/O loop.
>
> Returns an opaque handle that may be passed to *remove_timeout* to cancel.
>
> deadline may be a number denoting a time (on the same scale as *IOLoop.time*, normally time.time), or a datetime.timedelta object for a deadline relative to the current time.
>
> Note that it is not safe to call *add_timeout* from other threads. Instead, you must use *add_callback* to transfer control to the *IOLoop*'s thread, and then call *add_timeout* from there.

IOLoop.**remove_timeout**(*timeout*)
> Cancels a pending timeout.

> The argument is a handle as returned by *add_timeout*. It is safe to call *remove_timeout* even if the callback has already been run.

IOLoop.**time**()
> Returns the current time according to the *IOLoop*'s clock.

> The return value is a floating-point number relative to an unspecified time in the past.

> By default, the *IOLoop*'s time function is time.time. However, it may be configured to use e.g. time.monotonic instead. Calls to *add_timeout* that pass a number instead of a datetime.timedelta should use this function to compute the appropriate time, so they can work no matter what time function is chosen.

class tornado.ioloop.**PeriodicCallback**(*callback*, *callback_time*, *io_loop=None*)
> Schedules the given callback to be called periodically.

> The callback is called every callback_time milliseconds.

> *start* must be called after the *PeriodicCallback* is created.

> **start**()
> > Starts the timer.

> **stop**()
> > Stops the timer.

Debugging and error handling

IOLoop.**handle_callback_exception**(*callback*)
> This method is called whenever a callback run by the *IOLoop* throws an exception.

> By default simply logs the exception as an error. Subclasses may override this method to customize reporting of exceptions.

> The exception itself is not passed explicitly, but is available in sys.exc_info.

IOLoop.**set_blocking_signal_threshold**(*seconds*, *action*)
> Sends a signal if the *IOLoop* is blocked for more than s seconds.

> Pass seconds=None to disable. Requires Python 2.6 on a unixy platform.

> The action parameter is a Python signal handler. Read the documentation for the signal module for more information. If action is None, the process will be killed if it is blocked for too long.

IOLoop.**set_blocking_log_threshold**(*seconds*)
> Logs a stack trace if the *IOLoop* is blocked for more than s seconds.

> Equivalent to set_blocking_signal_threshold(seconds, self.log_stack)

IOLoop.**log_stack**(*signal*, *frame*)
> Signal handler to log the stack trace of the current thread.

> For use with *set_blocking_signal_threshold*.

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

Utility classes to write to and read from non-blocking files and sockets.

Contents:

- *BaseIOStream*: Generic interface for reading and writing.
- *IOStream*: Implementation of BaseIOStream using non-blocking sockets.
- *SSLIOStream*: SSL-aware version of IOStream.
- *PipeIOStream*: Pipe-based IOStream implementation.

Base class

class `tornado.iostream.`**`BaseIOStream`**(*io_loop=None*,   *max_buffer_size=None*,
                                          read_chunk_size=4096)
>   A utility class to write to and read from a non-blocking file or socket.

>   We support a non-blocking `write()` and a family of `read_*()` methods. All of the methods take callbacks (since writing and reading are non-blocking and asynchronous).

>   When a stream is closed due to an error, the IOStream's `error` attribute contains the exception object.

>   Subclasses must implement *fileno*, *close_fd*, *write_to_fd*, *read_from_fd*, and optionally *get_fd_error*.

Main interface

`BaseIOStream.`**`write`**(*data*, *callback=None*)
>   Write the given data to this stream.

>   If `callback` is given, we call it when all of the buffered write data has been successfully written to the stream. If there was previously buffered write data and an old write callback, that callback is simply overwritten with this new callback.

`BaseIOStream.`**`read_bytes`**(*num_bytes*, *callback*, *streaming_callback=None*)
>   Run callback when we read the given number of bytes.

>   If a `streaming_callback` is given, it will be called with chunks of data as they become available, and the argument to the final `callback` will be empty. Otherwise, the `callback` gets the data as an argument.

`BaseIOStream.`**`read_until`**(*delimiter*, *callback*)
>   Run `callback` when we read the given delimiter.

>   The callback will get the data read (including the delimiter) as an argument.

`BaseIOStream.`**`read_until_regex`**(*regex*, *callback*)
>   Run `callback` when we read the given regex pattern.

>   The callback will get the data read (including the data that matched the regex and anything that came before it) as an argument.

`BaseIOStream.`**`read_until_close`**(*callback*, *streaming_callback=None*)
>   Reads all data from the socket until it is closed.

>   If a `streaming_callback` is given, it will be called with chunks of data as they become available, and the argument to the final `callback` will be empty. Otherwise, the `callback` gets the data as an argument.

Subject to `max_buffer_size` limit from *IOStream* constructor if a `streaming_callback` is not used.

BaseIOStream.**close**(*exc_info=False*)
  Close this stream.

  If `exc_info` is true, set the `error` attribute to the current exception from `sys.exc_info` (or if `exc_info` is a tuple, use that instead of `sys.exc_info`).

BaseIOStream.**set_close_callback**(*callback*)
  Call the given callback when the stream is closed.

BaseIOStream.**closed**()
  Returns true if the stream has been closed.

BaseIOStream.**reading**()
  Returns true if we are currently reading from the stream.

BaseIOStream.**writing**()
  Returns true if we are currently writing to the stream.

BaseIOStream.**set_nodelay**(*value*)
  Sets the no-delay flag for this stream.

  By default, data written to TCP streams may be held for a time to make the most efficient use of bandwidth (according to Nagle's algorithm). The no-delay flag requests that data be written as soon as possible, even if doing so would consume additional bandwidth.

  This flag is currently defined only for TCP-based `IOStreams`.

  New in version 3.1.

Methods for subclasses

BaseIOStream.**fileno**()
  Returns the file descriptor for this stream.

BaseIOStream.**close_fd**()
  Closes the file underlying this stream.

  `close_fd` is called by *BaseIOStream* and should not be called elsewhere; other users should call *close* instead.

BaseIOStream.**write_to_fd**(*data*)
  Attempts to write `data` to the underlying file.

  Returns the number of bytes written.

BaseIOStream.**read_from_fd**()
  Attempts to read from the underlying file.

  Returns `None` if there was nothing to read (the socket returned `EWOULDBLOCK` or equivalent), otherwise returns the data. When possible, should return no more than `self.read_chunk_size` bytes at a time.

BaseIOStream.**get_fd_error**()
  Returns information about any error on the underlying file.

  This method is called after the *IOLoop* has signaled an error on the file descriptor, and should return an Exception (such as `socket.error` with additional information, or None if no such information is available.

Implementations

class `tornado.iostream.`**`IOStream`**(*socket*, **args*, ***kwargs*)

Socket-based `IOStream` implementation.

This class supports the read and write methods from `BaseIOStream` plus a `connect` method.

The `socket` parameter may either be connected or unconnected. For server operations the socket is the result of calling `socket.accept`. For client operations the socket is created with `socket.socket`, and may either be connected before passing it to the `IOStream` or connected with `IOStream.connect`.

A very simple (and broken) HTTP client using this class:

```
import tornado.ioloop
import tornado.iostream
import socket

def send_request():
    stream.write(b"GET / HTTP/1.0\r\nHost: friendfeed.com\r\n\r\n")
    stream.read_until(b"\r\n\r\n", on_headers)

def on_headers(data):
    headers = {}
    for line in data.split(b"\r\n"):
        parts = line.split(b":")
        if len(parts) == 2:
            headers[parts[0].strip()] = parts[1].strip()
    stream.read_bytes(int(headers[b"Content-Length"]), on_body)

def on_body(data):
    print data
    stream.close()
    tornado.ioloop.IOLoop.instance().stop()

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
stream = tornado.iostream.IOStream(s)
stream.connect(("friendfeed.com", 80), send_request)
tornado.ioloop.IOLoop.instance().start()
```

`connect`(*address*, *callback=None*, *server_hostname=None*)

Connects the socket to a remote address without blocking.

May only be called if the socket passed to the constructor was not previously connected. The address parameter is in the same format as for `socket.connect`, i.e. a `(host, port)` tuple. If `callback` is specified, it will be called when the connection is completed.

If specified, the `server_hostname` parameter will be used in SSL connections for certificate validation (if requested in the `ssl_options`) and SNI (if supported; requires Python 3.2+).

Note that it is safe to call `IOStream.write` while the connection is pending, in which case the data will be written as soon as the connection is ready. Calling `IOStream` read methods before the socket is connected works on some platforms but is non-portable.

class `tornado.iostream.`**`SSLIOStream`**(**args*, ***kwargs*)

A utility class to write to and read from a non-blocking SSL socket.

If the socket passed to the constructor is already connected, it should be wrapped with:

```
ssl.wrap_socket(sock, do_handshake_on_connect=False, **kwargs)
```

before constructing the *SSLIOStream*. Unconnected sockets will be wrapped when *IOStream.connect* is finished.

The `ssl_options` keyword argument may either be a dictionary of keywords arguments for `ssl.wrap_socket`, or an `ssl.SSLContext` object.

class `tornado.iostream.`**`PipeIOStream`**(*fd*, **args*, ***kwargs*)
  Pipe-based *IOStream* implementation.

  The constructor takes an integer file descriptor (such as one returned by `os.pipe`) rather than an open file object. Pipes are generally one-way, so a *PipeIOStream* can be used for reading or writing but not both.

Exceptions

exception `tornado.iostream.`**`StreamClosedError`**
  Exception raised by *IOStream* methods when the stream is closed.

  Note that the close callback is scheduled to run *after* other callbacks on the stream (to allow for buffered data to be processed), so you may see this error before you see the close callback.

`tornado.httpclient` — Asynchronous HTTP client

Blocking and non-blocking HTTP client interfaces.

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

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

- `curl_httpclient` has some features not found in `simple_httpclient`, including support for HTTP proxies and the ability to use a specified network interface.

- `curl_httpclient` is more likely to be compatible with sites that are not-quite-compliant with the HTTP spec, or sites that use little-exercised features of HTTP.

- `curl_httpclient` is faster.

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

Note that if you are using `curl_httpclient`, it is highly recommended that you use a recent version of `libcurl` and `pycurl`. Currently the minimum supported version is 7.18.2, and the recommended version is 7.21.1 or newer.

HTTP client interfaces

class `tornado.httpclient.`**`HTTPClient`**(*async_client_class=None*, ***kwargs*)
  A blocking HTTP client.

  This interface is provided for convenience and testing; most applications that are running an IOLoop will want to use *AsyncHTTPClient* instead. Typical usage looks like this:

```
http_client = httpclient.HTTPClient()
try:
    response = http_client.fetch("http://www.google.com/")
    print response.body
```

```
except httpclient.HTTPError as e:
    print "Error:", e
http_client.close()
```

close()
>   Closes the HTTPClient, freeing any resources used.

fetch(*request*, ***kwargs*)
>   Executes a request, returning an *HTTPResponse*.
>
>   The request may be either a string URL or an *HTTPRequest* object. If it is a string, we construct an *HTTPRequest* using any additional kwargs: HTTPRequest(request, **kwargs)
>
>   If an error occurs during the fetch, we raise an *HTTPError*.

class tornado.httpclient.**AsyncHTTPClient**
>   An non-blocking HTTP client.
>
>   Example usage:

```
def handle_request(response):
    if response.error:
        print "Error:", response.error
    else:
        print response.body

http_client = AsyncHTTPClient()
http_client.fetch("http://www.google.com/", handle_request)
```

>   The constructor for this class is magic in several respects: It actually creates an instance of an implementation-specific subclass, and instances are reused as a kind of pseudo-singleton (one per *IOLoop*). The keyword argument force_instance=True can be used to suppress this singleton behavior. Constructor arguments other than io_loop and force_instance are deprecated. The implementation subclass as well as arguments to its constructor can be set with the static method *configure()*

close()
>   Destroys this HTTP client, freeing any file descriptors used. Not needed in normal use, but may be helpful in unittests that create and destroy http clients. No other methods may be called on the *AsyncHTTPClient* after close().

fetch(*request*, *callback=None*, ***kwargs*)
>   Executes a request, asynchronously returning an *HTTPResponse*.
>
>   The request may be either a string URL or an *HTTPRequest* object. If it is a string, we construct an *HTTPRequest* using any additional kwargs: HTTPRequest(request, **kwargs)
>
>   This method returns a *Future* whose result is an *HTTPResponse*. The Future wil raise an *HTTPError* if the request returned a non-200 response code.
>
>   If a callback is given, it will be invoked with the *HTTPResponse*. In the callback interface, *HTTPError* is not automatically raised. Instead, you must check the response's error attribute or call its *rethrow* method.

classmethod configure(*impl*, ***kwargs*)
>   Configures the *AsyncHTTPClient* subclass to use.
>
>   AsyncHTTPClient() actually creates an instance of a subclass. This method may be called with either a class object or the fully-qualified name of such a class (or None to use the default, SimpleAsyncHTTPClient)

If additional keyword arguments are given, they will be passed to the constructor of each subclass instance created. The keyword argument `max_clients` determines the maximum number of simultaneous *`fetch()`* operations that can execute in parallel on each *`IOLoop`*. Additional arguments may be supported depending on the implementation class in use.

Example:

```
AsyncHTTPClient.configure("tornado.curl_httpclient.CurlAsyncHTTPClient")
```

Request objects

class `tornado.httpclient.`**`HTTPRequest`**(*url*, *method='GET'*, *headers=None*, *body=None*, *auth_username=None*, *auth_password=None*, *auth_mode=None*, *connect_timeout=None*, *request_timeout=None*, *if_modified_since=None*, *follow_redirects=None*, *max_redirects=None*, *user_agent=None*, *use_gzip=None*, *network_interface=None*, *streaming_callback=None*, *header_callback=None*, *prepare_curl_callback=None*, *proxy_host=None*, *proxy_port=None*, *proxy_username=None*, *proxy_password=None*, *allow_nonstandard_methods=None*, *validate_cert=None*, *ca_certs=None*, *allow_ipv6=None*, *client_key=None*, *client_cert=None*)

HTTP client request object.

All parameters except `url` are optional.

> **Parameters**
>
> - **url** (*string*) – URL to fetch
>
> - **method** (*string*) – HTTP method, e.g. "GET" or "POST"
>
> - **headers** (*HTTPHeaders* or *dict*) – Additional HTTP headers to pass on the request
>
> - **body** – HTTP body to pass on the request
>
> - **auth_username** (*string*) – Username for HTTP authentication
>
> - **auth_password** (*string*) – Password for HTTP authentication
>
> - **auth_mode** (*string*) – Authentication mode; default is "basic". Allowed values are implementation-defined; `curl_httpclient` supports "basic" and "digest"; `simple_httpclient` only supports "basic"
>
> - **connect_timeout** (*float*) – Timeout for initial connection in seconds
>
> - **request_timeout** (*float*) – Timeout for entire request in seconds
>
> - **if_modified_since** (*datetime* or *float*) – Timestamp for If-Modified-Since header
>
> - **follow_redirects** (*bool*) – Should redirects be followed automatically or return the 3xx response?
>
> - **max_redirects** (*int*) – Limit for `follow_redirects`
>
> - **user_agent** (*string*) – String to send as User-Agent header
>
> - **use_gzip** (*bool*) – Request gzip encoding from the server

- **network_interface** (*string*) – Network interface to use for request

- **streaming_callback** (*callable*) – If set, `streaming_callback` will be run with each chunk of data as it is received, and `HTTPResponse.body` and `HTTPResponse.buffer` will be empty in the final response.

- **header_callback** (*callable*) – If set, `header_callback` will be run with each header line as it is received (including the first line, e.g. `HTTP/1.0 200 OK\r\n`, and a final line containing only `\r\n`. All lines include the trailing newline characters). `HTTPResponse.headers` will be empty in the final response. This is most useful in conjunction with `streaming_callback`, because it's the only way to get access to header data while the request is in progress.

- **prepare_curl_callback** (*callable*) – If set, will be called with a `pycurl.Curl` object to allow the application to make additional `setopt` calls.

- **proxy_host** (*string*) – HTTP proxy hostname. To use proxies, `proxy_host` and `proxy_port` must be set; `proxy_username` and `proxy_pass` are optional. Proxies are currently only supported with `curl_httpclient`.

- **proxy_port** (*int*) – HTTP proxy port

- **proxy_username** (*string*) – HTTP proxy username

- **proxy_password** (*string*) – HTTP proxy password

- **allow_nonstandard_methods** (*bool*) – Allow unknown values for `method` argument?

- **validate_cert** (*bool*) – For HTTPS requests, validate the server's certificate?

- **ca_certs** (*string*) – filename of CA certificates in PEM format, or None to use defaults. Note that in `curl_httpclient`, if any request uses a custom `ca_certs` file, they all must (they don't have to all use the same `ca_certs`, but it's not possible to mix requests with `ca_certs` and requests that use the defaults.

- **allow_ipv6** (*bool*) – Use IPv6 when available? Default is false in `simple_httpclient` and true in `curl_httpclient`

- **client_key** (*string*) – Filename for client SSL key, if any

- **client_cert** (*string*) – Filename for client SSL certificate, if any

New in version 3.1: The `auth_mode` argument.

Response objects

class tornado.httpclient.**HTTPResponse**(*request*, *code*, *headers=None*, *buffer=None*, *effective_url=None*, *error=None*, *request_time=None*, *time_info=None*, *reason=None*)

HTTP Response object.

Attributes:

- request: HTTPRequest object

- code: numeric HTTP status code, e.g. 200 or 404

- reason: human-readable reason phrase describing the status code (with curl_httpclient, this is a default value rather than the server's actual response)

- headers: *tornado.httputil.HTTPHeaders* object

- •buffer: `cStringIO` object for response body

- •body: response body as string (created on demand from `self.buffer`)

- •error: Exception object, if any

- •request_time: seconds from request start to finish

- •time_info: dictionary of diagnostic timing information from the request. Available data are subject to change, but currently uses timings available from http://curl.haxx.se/libcurl/c/curl_easy_getinfo.html, plus `queue`, which is the delay (if any) introduced by waiting for a slot under *AsyncHTTPClient*'s `max_clients` setting.

rethrow()
> If there was an error on the request, raise an *HTTPError*.

Exceptions

exception `tornado.httpclient.`**HTTPError**(*code*, *message=None*, *response=None*)
> Exception thrown for an unsuccessful HTTP request.

> Attributes:

- •`code` - HTTP error integer error code, e.g. 404. Error code 599 is used when no HTTP response was received, e.g. for a timeout.

- •`response` - *HTTPResponse* object, if any.

> Note that if `follow_redirects` is False, redirects become HTTPErrors, and you can look at `error.response.headers['Location']` to see the destination of the redirect.

Command-line interface

This module provides a simple command-line interface to fetch a url using Tornado's HTTP client. Example usage:

```
# Fetch the url and print its body
python -m tornado.httpclient http://www.google.com

# Just print the headers
python -m tornado.httpclient --print_headers --print_body=false http://www.google.com
```

`tornado.netutil` — Miscellaneous network utilities

Miscellaneous network utility code.

`tornado.netutil.`**bind_sockets**(*port*, *address=None*, *family=0*, *backlog=128*, *flags=None*)
> Creates listening sockets bound to the given port and address.

> Returns a list of socket objects (multiple sockets are returned if the given address maps to multiple IP addresses, which is most common for mixed IPv4 and IPv6 use).

> Address may be either an IP address or hostname. If it's a hostname, the server will listen on all IP addresses associated with the name. Address may be an empty string or None to listen on all available interfaces. Family may be set to either `socket.AF_INET` or `socket.AF_INET6` to restrict to IPv4 or IPv6 addresses, otherwise both will be used if available.

> The `backlog` argument has the same meaning as for `socket.listen()`.

flags is a bitmask of AI_* flags to `getaddrinfo`, like socket.AI_PASSIVE | socket.AI_NUMERICHOST.

tornado.netutil.**bind_unix_socket**(*file*, *mode=384*, *backlog=128*)
   Creates a listening unix socket.

   If a socket with the given name already exists, it will be deleted. If any other file with that name exists, an exception will be raised.

   Returns a socket object (not a list of socket objects like `bind_sockets`)

tornado.netutil.**add_accept_handler**(*sock*, *callback*, *io_loop=None*)
   Adds an `IOLoop` event handler to accept new connections on sock.

   When a connection is accepted, callback(connection, address) will be run (connection is a socket object, and address is the address of the other end of the connection). Note that this signature is different from the callback(fd, events) signature used for `IOLoop` handlers.

tornado.netutil.**is_valid_ip**(*ip*)
   Returns true if the given string is a well-formed IP address.

   Supports IPv4 and IPv6.

class tornado.netutil.**Resolver**
   Configurable asynchronous DNS resolver interface.

   By default, a blocking implementation is used (which simply calls `socket.getaddrinfo`). An alternative implementation can be chosen with the `Resolver.configure` class method:

   ```
   Resolver.configure('tornado.netutil.ThreadedResolver')
   ```

   The implementations of this interface included with Tornado are

   - *tornado.netutil.BlockingResolver*
   - *tornado.netutil.ThreadedResolver*
   - *tornado.netutil.OverrideResolver*
   - *tornado.platform.twisted.TwistedResolver*
   - *tornado.platform.caresresolver.CaresResolver*

   resolve(*host*, *port*, *family=0*, *callback=None*)
      Resolves an address.

      The host argument is a string which may be a hostname or a literal IP address.

      Returns a *Future* whose result is a list of (family, address) pairs, where address is a tuple suitable to pass to `socket.connect` (i.e. a (host, port) pair for IPv4; additional fields may be present for IPv6). If a callback is passed, it will be run with the result as an argument when it is complete.

   close()
      Closes the *Resolver*, freeing any resources used.

      New in version 3.1.

class tornado.netutil.**ExecutorResolver**
   Resolver implementation using a `concurrent.futures.Executor`.

   Use this instead of *ThreadedResolver* when you require additional control over the executor being used.

   The executor will be shut down when the resolver is closed unless close_resolver=False; use this if you want to reuse the same executor elsewhere.

class `tornado.netutil.`**`BlockingResolver`**
>   Default *`Resolver`* implementation, using `socket.getaddrinfo`.
>
>   The *`IOLoop`* will be blocked during the resolution, although the callback will not be run until the next *`IOLoop`* iteration.

class `tornado.netutil.`**`ThreadedResolver`**
>   Multithreaded non-blocking *`Resolver`* implementation.
>
>   Requires the `concurrent.futures` package to be installed (available in the standard library since Python 3.2, installable with `pip install futures` in older versions).
>
>   The thread pool size can be configured with:

```
Resolver.configure('tornado.netutil.ThreadedResolver',
                   num_threads=10)
```

>   Changed in version 3.1: All `ThreadedResolvers` share a single thread pool, whose size is set by the first one to be created.

class `tornado.netutil.`**`OverrideResolver`**
>   Wraps a resolver with a mapping of overrides.
>
>   This can be used to make local DNS changes (e.g. for testing) without modifying system-wide settings.
>
>   The mapping can contain either host strings or host-port pairs.

`tornado.netutil.`**`ssl_options_to_context`**(*ssl_options*)
>   Try to convert an `ssl_options` dictionary to an `SSLContext` object.
>
>   The `ssl_options` dictionary contains keywords to be passed to `ssl.wrap_socket`. In Python 3.2+, `ssl.SSLContext` objects can be used instead. This function converts the dict form to its `SSLContext` equivalent, and may be used when a component which accepts both forms needs to upgrade to the `SSLContext` version to use features like SNI or NPN.

`tornado.netutil.`**`ssl_wrap_socket`**(*socket*, *ssl_options*, *server_hostname=None*, ***kwargs*)
>   Returns an `ssl.SSLSocket` wrapping the given socket.
>
>   `ssl_options` may be either a dictionary (as accepted by *`ssl_options_to_context`*) or an `ssl.SSLContext` object. Additional keyword arguments are passed to `wrap_socket` (either the `SSLContext` method or the `ssl` module function as appropriate).

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

A non-blocking, single-threaded TCP server.

class `tornado.tcpserver.`**`TCPServer`**(*io_loop=None*, *ssl_options=None*, *max_buffer_size=None*)
>   A non-blocking, single-threaded TCP server.
>
>   To use *`TCPServer`*, define a subclass which overrides the *`handle_stream`* method.
>
>   To make this server serve SSL traffic, send the ssl_options dictionary argument with the arguments required for the `ssl.wrap_socket` method, including "certfile" and "keyfile":

```
TCPServer(ssl_options={
    "certfile": os.path.join(data_dir, "mydomain.crt"),
    "keyfile": os.path.join(data_dir, "mydomain.key"),
})
```

>   *`TCPServer`* initialization follows one of three patterns:

1. *listen*: simple single-process:

```
server = TCPServer()
server.listen(8888)
IOLoop.instance().start()
```

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

```
server = TCPServer()
server.bind(8888)
server.start(0)  # Forks multiple sub-processes
IOLoop.instance().start()
```

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

3. *add_sockets*: advanced multi-process:

```
sockets = bind_sockets(8888)
tornado.process.fork_processes(0)
server = TCPServer()
server.add_sockets(sockets)
IOLoop.instance().start()
```

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

New in version 3.1: The max_buffer_size argument.

listen (*port*, *address=''*)
Starts accepting connections on the given port.

This method may be called more than once to listen on multiple ports. *listen* takes effect immediately; it is not necessary to call *TCPServer.start* afterwards. It is, however, necessary to start the *IOLoop*.

add_sockets (*sockets*)
Makes this server start accepting connections on the given sockets.

The sockets parameter is a list of socket objects such as those returned by *bind_sockets*. *add_sockets* is typically used in combination with that method and *tornado.process.fork_processes* to provide greater control over the initialization of a multi-process server.

add_socket (*socket*)
Singular version of *add_sockets*. Takes a single socket object.

bind (*port*, *address=None*, *family=0*, *backlog=128*)
Binds this server to the given port on the given address.

To start the server, call *start*. If you want to run this server in a single process, you can call *listen* as a shortcut to the sequence of *bind* and *start* calls.

Address may be either an IP address or hostname. If it's a hostname, the server will listen on all IP addresses associated with the name. Address may be an empty string or None to listen on all available interfaces. Family may be set to either socket.AF_INET or socket.AF_INET6 to restrict to IPv4 or IPv6 addresses, otherwise both will be used if available.

The backlog argument has the same meaning as for socket.listen.

This method may be called multiple times prior to *start* to listen on multiple ports or interfaces.

start (*num_processes=1*)
:   Starts this server in the *IOLoop*.

    By default, we run the server in this process and do not fork any additional child process.

    If num_processes is `None` or `<= 0`, we detect the number of cores available on this machine and fork that number of child processes. If num_processes is given and > 1, we fork that specific number of sub-processes.

    Since we use processes and not threads, there is no shared memory between any server code.

    Note that multiple processes are not compatible with the autoreload module (or the `debug=True` option to *tornado.web.Application*). When using multiple processes, no IOLoops can be created or referenced until after the call to `TCPServer.start(n)`.

stop ()
:   Stops listening for new connections.

    Requests currently in progress may still continue after the server is stopped.

handle_stream (*stream*, *address*)
:   Override to handle a new *IOStream* from an incoming connection.

Integration with other services

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

This module contains implementations of various third-party authentication schemes.

All the classes in this file are class mixins designed to be used with the *tornado.web.RequestHandler* class. They are used in two ways:

- On a login handler, use methods such as authenticate_redirect(), authorize_redirect(), and get_authenticated_user() to establish the user's identity and store authentication tokens to your database and/or cookies.

- In non-login handlers, use methods such as facebook_request() or twitter_request() to use the authentication tokens to make requests to the respective services.

They all take slightly different arguments due to the fact all these services implement authentication and authorization slightly differently. See the individual service classes below for complete documentation.

Example usage for Google OpenID:

```
class GoogleLoginHandler(tornado.web.RequestHandler,
                         tornado.auth.GoogleMixin):
    @tornado.web.asynchronous
    @tornado.gen.coroutine
    def get(self):
        if self.get_argument("openid.mode", None):
            user = yield self.get_authenticated_user()
            # Save the user with e.g. set_secure_cookie()
        else:
            yield self.authenticate_redirect()
```

Common protocols

These classes implement the OpenID and OAuth standards. They will generally need to be subclassed to use them with any particular site. The degree of customization required will vary, but in most cases overridding the class attributes (which are named beginning with underscores for historical reasons) should be sufficient.

class `tornado.auth.`**`OpenIdMixin`**
> Abstract implementation of OpenID and Attribute Exchange.
>
> See *GoogleMixin* below for a customized example (which also includes OAuth support).
>
> Class attributes:
>
> > • `_OPENID_ENDPOINT`: the identity provider's URI.
>
> **`authenticate_redirect`**(**args*, ***kwargs*)
> > Redirects to the authentication URL for this service.
> >
> > After authentication, the service will redirect back to the given callback URI with additional parameters including `openid.mode`.
> >
> > We request the given attributes for the authenticated user by default (name, email, language, and username). If you don't need all those attributes for your app, you can request fewer with the ax_attrs keyword argument.
> >
> > Changed in version 3.1: Returns a *Future* and takes an optional callback. These are not strictly necessary as this method is synchronous, but they are supplied for consistency with *OAuthMixin.authorize_redirect*.
>
> **`get_authenticated_user`**(**args*, ***kwargs*)
> > Fetches the authenticated user data upon redirect.
> >
> > This method should be called by the handler that receives the redirect from the *authenticate_redirect()* method (which is often the same as the one that calls it; in that case you would call *get_authenticated_user* if the `openid.mode` parameter is present and *authenticate_redirect* if it is not).
> >
> > The result of this method will generally be used to set a cookie.
>
> **`get_auth_http_client`**()
> > Returns the *AsyncHTTPClient* instance to be used for auth requests.
> >
> > May be overridden by subclasses to use an HTTP client other than the default.

class `tornado.auth.`**`OAuthMixin`**
> Abstract implementation of OAuth 1.0 and 1.0a.
>
> See *TwitterMixin* and *FriendFeedMixin* below for example implementations, or *GoogleMixin* for an OAuth/OpenID hybrid.
>
> Class attributes:
>
> > • `_OAUTH_AUTHORIZE_URL`: The service's OAuth authorization url.
> >
> > • `_OAUTH_ACCESS_TOKEN_URL`: The service's OAuth access token url.
> >
> > • `_OAUTH_VERSION`: May be either "1.0" or "1.0a".
> >
> > • `_OAUTH_NO_CALLBACKS`: Set this to True if the service requires advance registration of callbacks.
>
> Subclasses must also override the *_oauth_get_user_future* and *_oauth_consumer_token* methods.

authorize_redirect(**args*, ***kwargs*)
> Redirects the user to obtain OAuth authorization for this service.
>
> The `callback_uri` may be omitted if you have previously registered a callback URI with the third-party service. For some sevices (including Friendfeed), you must use a previously-registered callback URI and cannot specify a callback via this method.
>
> This method sets a cookie called `_oauth_request_token` which is subsequently used (and cleared) in *get_authenticated_user* for security purposes.
>
> Note that this method is asynchronous, although it calls *RequestHandler.finish* for you so it may not be necessary to pass a callback or use the *Future* it returns. However, if this method is called from a function decorated with *gen.coroutine*, you must call it with `yield` to keep the response from being closed prematurely.
>
> Changed in version 3.1: Now returns a *Future* and takes an optional callback, for compatibility with *gen.coroutine*.

get_authenticated_user(**args*, ***kwargs*)
> Gets the OAuth authorized user and access token.
>
> This method should be called from the handler for your OAuth callback URL to complete the registration process. We run the callback with the authenticated user dictionary. This dictionary will contain an `access_key` which can be used to make authorized requests to this service on behalf of the user. The dictionary will also contain other fields such as `name`, depending on the service used.

_oauth_consumer_token()
> Subclasses must override this to return their OAuth consumer keys.
>
> The return value should be a *dict* with keys `key` and `secret`.

_oauth_get_user_future(**args*, ***kwargs*)
> Subclasses must override this to get basic information about the user.
>
> Should return a *Future* whose result is a dictionary containing information about the user, which may have been retrieved by using `access_token` to make a request to the service.
>
> The access token will be added to the returned dictionary to make the result of *get_authenticated_user*.
>
> For backwards compatibility, the callback-based `_oauth_get_user` method is also supported.

get_auth_http_client()
> Returns the *AsyncHTTPClient* instance to be used for auth requests.
>
> May be overridden by subclasses to use an HTTP client other than the default.

class tornado.auth.**OAuth2Mixin**
> Abstract implementation of OAuth 2.0.
>
> See *FacebookGraphMixin* below for an example implementation.
>
> Class attributes:
>
> - `_OAUTH_AUTHORIZE_URL`: The service's authorization url.
>
> - `_OAUTH_ACCESS_TOKEN_URL`: The service's access token url.

authorize_redirect(**args*, ***kwargs*)
> Redirects the user to obtain OAuth authorization for this service.
>
> Some providers require that you register a redirect URL with your application instead of passing one via this method. You should call this method to log the user in, and then call `get_authenticated_user` in the handler for your redirect URL to complete the authorization process.

Changed in version 3.1: Returns a *Future* and takes an optional callback. These are not strictly necessary as this method is synchronous, but they are supplied for consistency with *OAuthMixin.authorize_redirect*.

Google

class `tornado.auth.`**`GoogleMixin`**
Google Open ID / OAuth authentication.

No application registration is necessary to use Google for authentication or to access Google resources on behalf of a user.

Google implements both OpenID and OAuth in a hybrid mode. If you just need the user's identity, use *authenticate_redirect*. If you need to make requests to Google on behalf of the user, use *authorize_redirect*. On return, parse the response with *get_authenticated_user*. We send a dict containing the values for the user, including `email`, `name`, and `locale`.

Example usage:

```
class GoogleLoginHandler(tornado.web.RequestHandler,
                         tornado.auth.GoogleMixin):
    @tornado.web.asynchronous
    @tornado.gen.coroutine
    def get(self):
        if self.get_argument("openid.mode", None):
            user = yield self.get_authenticated_user()
            # Save the user with e.g. set_secure_cookie()
        else:
            yield self.authenticate_redirect()
```

`authorize_redirect`(**args*, ***kwargs*)
Authenticates and authorizes for the given Google resource.

Some of the available resources which can be used in the `oauth_scope` argument are:

  •Gmail Contacts - http://www.google.com/m8/feeds/

  •Calendar - http://www.google.com/calendar/feeds/

  •Finance - http://finance.google.com/finance/feeds/

You can authorize multiple resources by separating the resource URLs with a space.

Changed in version 3.1: Returns a *Future* and takes an optional callback. These are not strictly necessary as this method is synchronous, but they are supplied for consistency with *OAuthMixin.authorize_redirect*.

`get_authenticated_user`(**args*, ***kwargs*)
Fetches the authenticated user data upon redirect.

Facebook

class `tornado.auth.`**`FacebookGraphMixin`**
Facebook authentication using the new Graph API and OAuth2.

`get_authenticated_user`(**args*, ***kwargs*)
Handles the login for the Facebook user, returning a user object.

Example usage:

```
class FacebookGraphLoginHandler(LoginHandler, tornado.auth.
↪FacebookGraphMixin):
  @tornado.web.asynchronous
  @tornado.gen.coroutine
  def get(self):
      if self.get_argument("code", False):
          user = yield self.get_authenticated_user(
              redirect_uri='/auth/facebookgraph/',
              client_id=self.settings["facebook_api_key"],
              client_secret=self.settings["facebook_secret"],
              code=self.get_argument("code"))
          # Save the user with e.g. set_secure_cookie
      else:
          yield self.authorize_redirect(
              redirect_uri='/auth/facebookgraph/',
              client_id=self.settings["facebook_api_key"],
              extra_params={"scope": "read_stream,offline_access"})
```

facebook_request(**args*, ***kwargs*)

Fetches the given relative API path, e.g., "/btaylor/picture"

If the request is a POST, `post_args` should be provided. Query string arguments should be given as keyword arguments.

An introduction to the Facebook Graph API can be found at http://developers.facebook.com/docs/api

Many methods require an OAuth access token which you can obtain through *authorize_redirect* and *get_authenticated_user*. The user returned through that process includes an `access_token` attribute that can be used to make authenticated requests via this method.

Example usage:

```
class MainHandler(tornado.web.RequestHandler,
                  tornado.auth.FacebookGraphMixin):
    @tornado.web.authenticated
    @tornado.web.asynchronous
    @tornado.gen.coroutine
    def get(self):
        new_entry = yield self.facebook_request(
            "/me/feed",
            post_args={"message": "I am posting from my Tornado application!"}
↪,
            access_token=self.current_user["access_token"])

        if not new_entry:
            # Call failed; perhaps missing permission?
            yield self.authorize_redirect()
            return
        self.finish("Posted a message!")
```

The given path is relative to `self._FACEBOOK_BASE_URL`, by default "https://graph.facebook.com".

Changed in version 3.1: Added the ability to override `self._FACEBOOK_BASE_URL`.

get_auth_http_client()

Returns the *AsyncHTTPClient* instance to be used for auth requests.

May be overridden by subclasses to use an HTTP client other than the default.

class `tornado.auth.`**`FacebookMixin`**
> Facebook Connect authentication.
>
> *Deprecated:* New applications should use *`FacebookGraphMixin`* below instead of this class. This class does not support the Future-based interface seen on other classes in this module.
>
> To authenticate with Facebook, register your application with Facebook at http://www.facebook.com/developers/apps.php. Then copy your API Key and Application Secret to the application settings `facebook_api_key` and `facebook_secret`.
>
> When your application is set up, you can use this mixin like this to authenticate the user with Facebook:

```
class FacebookHandler(tornado.web.RequestHandler,
                      tornado.auth.FacebookMixin):
    @tornado.web.asynchronous
    def get(self):
        if self.get_argument("session", None):
            self.get_authenticated_user(self.async_callback(self._on_auth))
            return
        yield self.authenticate_redirect()

    def _on_auth(self, user):
        if not user:
            raise tornado.web.HTTPError(500, "Facebook auth failed")
        # Save the user using, e.g., set_secure_cookie()
```

> The user object returned by *`get_authenticated_user`* includes the attributes `facebook_uid` and `name` in addition to session attributes like `session_key`. You should save the session key with the user; it is required to make requests on behalf of the user later with *`facebook_request`*.

> **`authenticate_redirect`**(**args*, ***kwargs*)
> > Authenticates/installs this app for the current user.
> >
> > Changed in version 3.1: Returns a *`Future`* and takes an optional callback. These are not strictly necessary as this method is synchronous, but they are supplied for consistency with *`OAuthMixin.`* *`authorize_redirect`*.

> **`authorize_redirect`**(*extended_permissions*,   *callback_uri=None*,   *cancel_uri=None*,   *callback=None*)
> > Redirects to an authorization request for the given FB resource.
> >
> > The available resource names are listed at http://wiki.developers.facebook.com/index.php/Extended_permission. The most common resource types include:
> >
> > > •publish_stream
> > >
> > > •read_stream
> > >
> > > •email
> > >
> > > •sms
> >
> > extended_permissions can be a single permission name or a list of names. To get the session secret and session key, call get_authenticated_user() just as you would with authenticate_redirect().
> >
> > Changed in version 3.1: Returns a *`Future`* and takes an optional callback. These are not strictly necessary as this method is synchronous, but they are supplied for consistency with *`OAuthMixin.`* *`authorize_redirect`*.

> **`get_authenticated_user`**(*callback*)
> > Fetches the authenticated Facebook user.

The authenticated user includes the special Facebook attributes 'session_key' and 'facebook_uid' in addition to the standard user attributes like 'name'.

facebook_request (*method*, *callback*, ***args*)
Makes a Facebook API REST request.

We automatically include the Facebook API key and signature, but it is the callers responsibility to include 'session_key' and any other required arguments to the method.

The available Facebook methods are documented here: http://wiki.developers.facebook.com/index.php/API

Here is an example for the stream.get() method:

```
class MainHandler(tornado.web.RequestHandler,
                  tornado.auth.FacebookMixin):
    @tornado.web.authenticated
    @tornado.web.asynchronous
    def get(self):
        self.facebook_request(
            method="stream.get",
            callback=self.async_callback(self._on_stream),
            session_key=self.current_user["session_key"])

    def _on_stream(self, stream):
        if stream is None:
            # Not authorized to read the stream yet?
            self.redirect(self.authorize_redirect("read_stream"))
            return
        self.render("stream.html", stream=stream)
```

get_auth_http_client ()
Returns the *AsyncHTTPClient* instance to be used for auth requests.

May be overridden by subclasses to use an HTTP client other than the default.

Twitter

class tornado.auth.**TwitterMixin**
Twitter OAuth authentication.

To authenticate with Twitter, register your application with Twitter at http://twitter.com/apps. Then copy your Consumer Key and Consumer Secret to the application *settings* twitter_consumer_key and twitter_consumer_secret. Use this mixin on the handler for the URL you registered as your application's callback URL.

When your application is set up, you can use this mixin like this to authenticate the user with Twitter and get access to their stream:

```
class TwitterLoginHandler(tornado.web.RequestHandler,
                          tornado.auth.TwitterMixin):
    @tornado.web.asynchronous
    @tornado.gen.coroutine
    def get(self):
        if self.get_argument("oauth_token", None):
            user = yield self.get_authenticated_user()
            # Save the user using e.g. set_secure_cookie()
        else:
            yield self.authorize_redirect()
```

The user object returned by *get_authenticated_user* includes the attributes username, name, access_token, and all of the custom Twitter user attributes described at https://dev.twitter.com/docs/api/1.1/get/users/show

authenticate_redirect(**args*, ***kwargs*)

Just like *authorize_redirect*, but auto-redirects if authorized.

This is generally the right interface to use if you are using Twitter for single-sign on.

Changed in version 3.1: Now returns a *Future* and takes an optional callback, for compatibility with *gen.coroutine*.

twitter_request(**args*, ***kwargs*)

Fetches the given API path, e.g., statuses/user_timeline/btaylor

The path should not include the format or API version number. (we automatically use JSON format and API version 1).

If the request is a POST, post_args should be provided. Query string arguments should be given as keyword arguments.

All the Twitter methods are documented at http://dev.twitter.com/

Many methods require an OAuth access token which you can obtain through *authorize_redirect* and *get_authenticated_user*. The user returned through that process includes an 'access_token' attribute that can be used to make authenticated requests via this method. Example usage:

```
class MainHandler(tornado.web.RequestHandler,
                  tornado.auth.TwitterMixin):
    @tornado.web.authenticated
    @tornado.web.asynchronous
    @tornado.gen.coroutine
    def get(self):
        new_entry = yield self.twitter_request(
            "/statuses/update",
            post_args={"status": "Testing Tornado Web Server"},
            access_token=self.current_user["access_token"])
        if not new_entry:
            # Call failed; perhaps missing permission?
            yield self.authorize_redirect()
            return
        self.finish("Posted a message!")
```

FriendFeed

class tornado.auth.**FriendFeedMixin**

FriendFeed OAuth authentication.

To authenticate with FriendFeed, register your application with FriendFeed at http://friendfeed.com/api/applications. Then copy your Consumer Key and Consumer Secret to the application *settings* friendfeed_consumer_key and friendfeed_consumer_secret. Use this mixin on the handler for the URL you registered as your application's Callback URL.

When your application is set up, you can use this mixin like this to authenticate the user with FriendFeed and get access to their feed:

```
class FriendFeedLoginHandler(tornado.web.RequestHandler,
                             tornado.auth.FriendFeedMixin):
    @tornado.web.asynchronous
```

```
        @tornado.gen.coroutine
        def get(self):
            if self.get_argument("oauth_token", None):
                user = yield self.get_authenticated_user()
                # Save the user using e.g. set_secure_cookie()
            else:
                yield self.authorize_redirect()
```

The user object returned by *get_authenticated_user()* includes the attributes `username`, `name`, and `description` in addition to `access_token`. You should save the access token with the user; it is required to make requests on behalf of the user later with *friendfeed_request()*.

friendfeed_request (**args*, ***kwargs*)
    Fetches the given relative API path, e.g., "/bret/friends"

    If the request is a POST, `post_args` should be provided. Query string arguments should be given as keyword arguments.

    All the FriendFeed methods are documented at http://friendfeed.com/api/documentation.

    Many methods require an OAuth access token which you can obtain through *authorize_redirect* and *get_authenticated_user*. The user returned through that process includes an `access_token` attribute that can be used to make authenticated requests via this method.

    Example usage:

```
class MainHandler(tornado.web.RequestHandler,
                  tornado.auth.FriendFeedMixin):
    @tornado.web.authenticated
    @tornado.web.asynchronous
    @tornado.gen.coroutine
    def get(self):
        new_entry = yield self.friendfeed_request(
            "/entry",
            post_args={"body": "Testing Tornado Web Server"},
            access_token=self.current_user["access_token"])

        if not new_entry:
            # Call failed; perhaps missing permission?
            yield self.authorize_redirect()
            return
        self.finish("Posted a message!")
```

`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`.

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

This module lets you run applications and libraries written for Twisted in a Tornado application. It can be used in two modes, depending on which library's underlying event loop you want to use.

This module has been tested with Twisted versions 11.0.0 and newer.

Twisted on Tornado

class `tornado.platform.twisted.`**`TornadoReactor`**

> `TornadoReactor` implements the Twisted reactor interface on top of the Tornado IOLoop. To use it, simply call `install` at the beginning of the application:

```
import tornado.platform.twisted
tornado.platform.twisted.install()
from twisted.internet import reactor
```

> When the app is ready to start, call `IOLoop.instance().start()` instead of `reactor.run()`.

> It is also possible to create a non-global reactor by calling `tornado.platform.twisted.TornadoReactor(io_loop)`. However, if the *IOLoop* and reactor are to be short-lived (such as those used in unit tests), additional cleanup may be required. Specifically, it is recommended to call:

```
reactor.fireSystemEvent('shutdown')
reactor.disconnectAll()
```

> before closing the *IOLoop*.

Tornado on Twisted

class `tornado.platform.twisted.`**`TwistedIOLoop`**

> `TwistedIOLoop` implements the Tornado IOLoop interface on top of the Twisted reactor. Recommended usage:

```
from tornado.platform.twisted import TwistedIOLoop
from twisted.internet import reactor
TwistedIOLoop().install()
# Set up your tornado application as usual using `IOLoop.instance`
reactor.run()
```

> `TwistedIOLoop` always uses the global Twisted reactor.

Twisted DNS resolver

class `tornado.platform.twisted.`**`TwistedResolver`**

> This is a non-blocking and non-threaded resolver. It is recommended only when threads cannot be used, since it has limitations compared to the standard `getaddrinfo`-based *Resolver* and *ThreadedResolver*. Specifically, it returns at most one result, and arguments other than `host` and `family` are ignored. It may fail to resolve when `family` is not `socket.AF_UNSPEC`.

> Requires Twisted 12.1 or newer.

`tornado.websocket` — Bidirectional communication to the browser

Implementation of the WebSocket protocol.

WebSockets allow for bidirectional communication between the browser and server.

> **Warning:** The WebSocket protocol was recently finalized as RFC 6455 and is not yet supported in all browsers. Refer to http://caniuse.com/websockets for details on compatibility. In addition, during development the protocol went through several incompatible versions, and some browsers only support older versions. By default this module only supports the latest version of the protocol, but optional support for an older version (known as "draft 76" or "hixie-76") can be enabled by overriding *WebSocketHandler.allow_draft76* (see that method's documentation for caveats).

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

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

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

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

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

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

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

WebSockets are not standard HTTP connections. The "handshake" is HTTP, but after the handshake, the protocol is message-based. Consequently, most of the Tornado HTTP facilities are not available in handlers of this type. The only communication methods available to you are *write_message()*, *ping()*, and *close()*. Likewise, your request handler class should implement *open()* method rather than `get()` or `post()`.

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

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

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

Event handlers

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

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

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

This method must be overridden.

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

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

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

Output

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

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

WebSocketHandler.**close**()
Closes this Web Socket.

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

Configuration

WebSocketHandler.**allow_draft76**()
Override to enable support for the older "draft76" protocol.

The draft76 version of the websocket protocol is disabled by default due to security concerns, but it can be enabled by overriding this method to return True.

Connections using the draft76 protocol do not support the binary=True flag to *write_message*.

Support for the draft76 protocol is deprecated and will be removed in a future version of Tornado.

WebSocketHandler.**get_websocket_scheme**()
Return the url scheme used for this request, either "ws" or "wss".

This is normally decided by HTTPServer, but applications may wish to override this if they are using an SSL proxy that does not provide the X-Scheme header as understood by HTTPServer.

Note that this is only used by the draft76 protocol.

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

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

See *BaseIOStream.set_nodelay* for additional details.

New in version 3.1.

Other

WebSocketHandler.**async_callback**(*callback*, **args*, ***kwargs*)
    Obsolete - catches exceptions from the wrapped function.

    This function is normally unncecessary thanks to *tornado.stack_context*.

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

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

Client-side support

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

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

class tornado.websocket.**WebSocketClientConnection**(*io_loop*, *request*)
    WebSocket client connection.

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

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

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

`tornado.wsgi` — Interoperability with other Python frameworks and servers

WSGI support for the Tornado web framework.

WSGI is the Python standard for web servers, and allows for interoperability between Tornado and other Python web
frameworks and servers. This module provides WSGI support in two ways:

- *WSGIApplication* is a version of *tornado.web.Application* that can run inside a WSGI server.
  This is useful for running a Tornado app on another HTTP server, such as Google App Engine. See the
  WSGIApplication class documentation for limitations that apply.

- *WSGIContainer* lets you run other WSGI applications and frameworks on the Tornado HTTP server. For
  example, with this class you can mix Django and Tornado handlers in a single server.

WSGIApplication

class tornado.wsgi.**WSGIApplication**(*handlers=None*, *default_host=''*, ***settings*)
    A WSGI equivalent of *tornado.web.Application*.

WSGIApplication is very similar to *tornado.web.Application*, except no asynchronous methods are supported (since WSGI does not support non-blocking requests properly). If you call self.flush() or other asynchronous methods in your request handlers running in a *WSGIApplication*, we throw an exception.

Example usage:

```python
import tornado.web
import tornado.wsgi
import wsgiref.simple_server

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

if __name__ == "__main__":
    application = tornado.wsgi.WSGIApplication([
        (r"/", MainHandler),
    ])
    server = wsgiref.simple_server.make_server('', 8888, application)
    server.serve_forever()
```

See the appengine demo for an example of using this module to run a Tornado app on Google App Engine.

WSGI applications use the same *RequestHandler* class, but not @asynchronous methods or flush(). This means that it is not possible to use *AsyncHTTPClient*, or the *tornado.auth* or *tornado.websocket* modules.

class tornado.wsgi.**HTTPRequest**(*environ*)
   Mimics *tornado.httpserver.HTTPRequest* for WSGI applications.

   Parses the given WSGI environment to construct the request.

   supports_http_1_1()
      Returns True if this request supports HTTP/1.1 semantics

   cookies
      A dictionary of Cookie.Morsel objects.

   full_url()
      Reconstructs the full URL for this request.

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

WSGIContainer

class tornado.wsgi.**WSGIContainer**(*wsgi_application*)
   Makes a WSGI-compatible function runnable on Tornado's HTTP server.

   Wrap a WSGI function in a *WSGIContainer* and pass it to *HTTPServer* to run it. For example:

```python
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.instance().start()
```

This class is intended to let other frameworks (Django, web.py, etc) run on the Tornado HTTP server and I/O loop.

The `tornado.web.FallbackHandler` class is often useful for mixing Tornado and WSGI apps in the same server. See https://github.com/bdarnell/django-tornado-demo for a complete example.

static **environ** (*request*)
> Converts a `tornado.httpserver.HTTPRequest` to a WSGI environment.

Utilities

`tornado.autoreload` — **Automatically detect code changes in development**

xAutomatically restart the server when a source file is modified.

Most applications should not access this module directly. Instead, pass the keyword argument debug=True to the *tornado.web.Application* constructor. This will enable autoreload mode as well as checking for changes to templates and static resources. Note that restarting is a destructive operation and any requests in progress will be aborted when the process restarts.

This module can also be used as a command-line wrapper around scripts such as unit test runners. See the *main* method for details.

The command-line wrapper and Application debug modes can be used together. This combination is encouraged as the wrapper catches syntax errors and other import-time failures, while debug mode catches changes once the server has started.

This module depends on *IOLoop*, so it will not work in WSGI applications and Google App Engine. It also will not work correctly when *HTTPServer*'s multi-process mode is used.

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

tornado.autoreload.**add_reload_hook**(*fn*)
    Add a function to be called before reloading the process.

    Note that for open file and socket handles it is generally preferable to set the FD_CLOEXEC flag (using *fcntl* or tornado.platform.auto.set_close_exec) instead of using a reload hook to close them.

tornado.autoreload.**main**()
    Command-line wrapper to re-run a script whenever its source changes.

    Scripts may be specified by filename or module name:

```
python -m tornado.autoreload -m tornado.test.runtests
python -m tornado.autoreload tornado/test/runtests.py
```

Running a script with this wrapper is similar to calling `tornado.autoreload.wait` at the end of the script, but this wrapper can catch import-time problems like syntax errors that would otherwise prevent the script from reaching its call to `wait`.

tornado.autoreload.**start**(*io_loop=None*, *check_time=500*)
   Begins watching source files for changes using the given `IOLoop`.

tornado.autoreload.**wait**()
   Wait for a watched file to change, then restart the process.

   Intended to be used at the end of scripts like unit test runners, to run the tests again after any source file changes (but see also the command-line interface in `main`)

tornado.autoreload.**watch**(*filename*)
   Add a file to the watch list.

   All imported modules are watched by default.

`tornado.concurrent` — Work with threads and futures

Utilities for working with threads and `Futures`.

`Futures` are a pattern for concurrent programming introduced in Python 3.2 in the `concurrent.futures` package (this package has also been backported to older versions of Python and can be installed with `pip install futures`). Tornado will use `concurrent.futures.Future` if it is available; otherwise it will use a compatible class defined in this module.

class tornado.concurrent.**Future**
   A `Future` encapsulates the result of an asynchronous operation. In synchronous applications `Futures` are used to wait for the result from a thread or process pool; in Tornado they are normally used with `IOLoop.add_future` or by yielding them in a `gen.coroutine`.

   If the `concurrent.futures` package is available, `tornado.concurrent.Future` is simply an alias for `concurrent.futures.Future`. Otherwise, we support the same interface with a few limitations:

   • It is an error to call `result` or `exception` before the `Future` has completed.

   • Cancellation is not supported.

   result()
      If the operation succeeded, return its result. If it failed, re-raise its exception.

   exception()
      If the operation raised an exception, return the `Exception` object. Otherwise returns None.

   add_done_callback(*fn*)
      Attaches the given callback to the `Future`. It will be invoked with the `Future` as its argument when it has finished running and its result is available. In Tornado consider using `IOLoop.add_future` instead of calling `add_done_callback` directly.

   done()
      Returns True if the future has finished running and its `result` and `exception` methods are available.

class tornado.concurrent.**TracebackFuture**
   Subclass of `Future` which can store a traceback with exceptions.

The traceback is automatically available in Python 3, but in the Python 2 futures backport this information is discarded.

set_exc_info(*exc_info*)
> Traceback-aware replacement for `set_exception`.

tornado.concurrent.**run_on_executor**(*fn*)
> Decorator to run a synchronous method asynchronously on an executor.

> The decorated method may be called with a `callback` keyword argument and returns a future.

tornado.concurrent.**return_future**(*f*)
> Decorator to make a function that returns via callback return a *Future*.

> The wrapped function should take a `callback` keyword argument and invoke it with one argument when it has finished. To signal failure, the function can simply raise an exception (which will be captured by the *StackContext* and passed along to the `Future`).

> From the caller's perspective, the callback argument is optional. If one is given, it will be invoked when the function is complete with *Future.result()* as an argument. If the function fails, the callback will not be run and an exception will be raised into the surrounding *StackContext*.

> If no callback is given, the caller should use the `Future` to wait for the function to complete (perhaps by yielding it in a *gen.engine* function, or passing it to *IOLoop.add_future*).

> Usage:

```
@return_future
def future_func(arg1, arg2, callback):
    # Do stuff (possibly asynchronous)
    callback(result)

@gen.engine
def caller(callback):
    yield future_func(arg1, arg2)
    callback()
```

> Note that `@return_future` and `@gen.engine` can be applied to the same function, provided `@return_future` appears first. However, consider using `@gen.coroutine` instead of this combination.

tornado.concurrent.**chain_future**(*a*, *b*)
> Chain two futures together so that when one completes, so does the other.

> The result (success or failure) of `a` will be copied to `b`.

`tornado.httputil` — Manipulate HTTP headers and URLs

HTTP utility code shared by clients and servers.

class tornado.httputil.**HTTPHeaders**(**args*, ***kwargs*)
> A dictionary that maintains `Http-Header-Case` for all keys.

> Supports multiple values per key via a pair of new methods, *add()* and *get_list()*. The regular dictionary interface returns a single value per key, with multiple values joined by a comma.

```
>>> h = HTTPHeaders({"content-type": "text/html"})
>>> list(h.keys())
['Content-Type']
>>> h["Content-Type"]
'text/html'
```

```
>>> h.add("Set-Cookie", "A=B")
>>> h.add("Set-Cookie", "C=D")
>>> h["set-cookie"]
'A=B,C=D'
>>> h.get_list("set-cookie")
['A=B', 'C=D']
```

```
>>> for (k,v) in sorted(h.get_all()):
...     print('%s: %s' % (k,v))
...
Content-Type: text/html
Set-Cookie: A=B
Set-Cookie: C=D
```

add(*name*, *value*)
> Adds a new value for the given key.

get_list(*name*)
> Returns all values for the given header as a list.

get_all()
> Returns an iterable of all (name, value) pairs.
>
> If a header has multiple values, multiple pairs will be returned with the same name.

parse_line(*line*)
> Updates the dictionary with a single header line.

```
>>> h = HTTPHeaders()
>>> h.parse_line("Content-Type: text/html")
>>> h.get('content-type')
'text/html'
```

classmethod parse(*headers*)
> Returns a dictionary from HTTP header text.

```
>>> h = HTTPHeaders.parse("Content-Type: text/html\r\nContent-Length: 42\r\n")
>>> sorted(h.items())
[('Content-Length', '42'), ('Content-Type', 'text/html')]
```

tornado.httputil.**url_concat**(*url*, *args*)
> Concatenate url and argument dictionary regardless of whether url has existing query parameters.

```
>>> url_concat("http://example.com/foo?a=b", dict(c="d"))
'http://example.com/foo?a=b&c=d'
```

class tornado.httputil.**HTTPFile**
> Represents a file uploaded via a form.
>
> For backwards compatibility, its instance attributes are also accessible as dictionary keys.
>
> > • filename
> >
> > • body
> >
> > • content_type

tornado.httputil.**parse_body_arguments**(*content_type*, *body*, *arguments*, *files*)
> Parses a form request body.

Supports `application/x-www-form-urlencoded` and `multipart/form-data`. The `content_type` parameter should be a string and `body` should be a byte string. The `arguments` and `files` parameters are dictionaries that will be updated with the parsed contents.

tornado.httputil.**parse_multipart_form_data**(*boundary*, *data*, *arguments*, *files*)
> Parses a `multipart/form-data` body.
>
> The `boundary` and `data` parameters are both byte strings. The dictionaries given in the arguments and files parameters will be updated with the contents of the body.

tornado.httputil.**format_timestamp**(*ts*)
> Formats a timestamp in the format used by HTTP.
>
> The argument may be a numeric timestamp as returned by `time.time`, a time tuple as returned by `time.gmtime`, or a `datetime.datetime` object.

```
>>> format_timestamp(1359312200)
'Sun, 27 Jan 2013 18:43:20 GMT'
```

`tornado.log` — Logging support

Logging support for Tornado.

Tornado uses three logger streams:

- `tornado.access`: Per-request logging for Tornado's HTTP servers (and potentially other servers in the future)

- `tornado.application`: Logging of errors from application code (i.e. uncaught exceptions from callbacks)

- `tornado.general`: General-purpose logging, including any errors or warnings from Tornado itself.

These streams may be configured independently using the standard library's `logging` module. For example, you may wish to send `tornado.access` logs to a separate file for analysis.

class tornado.log.**LogFormatter**(*color=True*, **args*, ***kwargs*)
> 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* (unless `--logging=none` is used).

tornado.log.**enable_pretty_logging**(*options=None*, *logger=None*)
> Turns on formatted logging output as configured.
>
> This is called automaticaly by *tornado.options.parse_command_line* and *tornado.options.parse_config_file*.

`tornado.options` — Command-line parsing

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

Each module defines its own options which are added to the global option namespace, e.g.:

```
from tornado.options import define, options

define("mysql_host", default="127.0.0.1:3306", help="Main user DB")
define("memcache_hosts", default="127.0.0.1:11011", multiple=True,
       help="Main user memcache servers")

def connect():
    db = database.Connection(options.mysql_host)
    ...
```

The `main()` method of your application does not need to be aware of all of the options used throughout your program; they are all automatically loaded when the modules are loaded. However, all modules that define options must have been imported before the command line is parsed.

Your `main()` method can parse the command line or parse a config file with either:

```
tornado.options.parse_command_line()
# or
tornado.options.parse_config_file("/etc/server.conf")
```

Command line formats are what you would expect (`--myoption=myvalue`). Config files are just Python files. Global names become options, e.g.:

```
myoption = "myvalue"
myotheroption = "myothervalue"
```

We support `datetimes`, `timedeltas`, ints, and floats (just pass a `type` kwarg to `define`). We also accept multi-value options. See the documentation for `define()` below.

`tornado.options.options` is a singleton instance of `OptionParser`, and the top-level functions in this module (`define`, `parse_command_line`, etc) simply call methods on it. You may create additional `OptionParser` instances to define isolated sets of options, such as for subcommands.

Global functions

tornado.options.**define**(*name*, *default=None*, *type=None*, *help=None*, *metavar=None*, *multiple=False*, *group=None*, *callback=None*)
    Defines an option in the global namespace.

    See `OptionParser.define`.

tornado.options.**options**
    Global options object. All defined options are available as attributes on this object.

tornado.options.**parse_command_line**(*args=None*, *final=True*)
    Parses global options from the command line.

    See `OptionParser.parse_command_line`.

tornado.options.**parse_config_file**(*path*, *final=True*)
    Parses global options from a config file.

    See `OptionParser.parse_config_file`.

tornado.options.**print_help**(*file=sys.stderr*)
    Prints all the command line options to stderr (or another file).

    See `OptionParser.print_help`.

`tornado.options.`**`add_parse_callback`**(*callback*)
> Adds a parse callback, to be invoked when option parsing is done.
>
> See *OptionParser.add_parse_callback*

exception `tornado.options.`**`Error`**
> Exception raised by errors in the options module.

OptionParser class

class `tornado.options.`**`OptionParser`**
> A collection of options, a dictionary with object-like access.
>
> Normally accessed via static functions in the *tornado.options* module, which reference a global instance.

> **`add_parse_callback`**(*callback*)
> > Adds a parse callback, to be invoked when option parsing is done.

> **`as_dict`**()
> > The names and values of all options.
> >
> > New in version 3.1.

> **`define`**(*name*, *default=None*, *type=None*, *help=None*, *metavar=None*, *multiple=False*, *group=None*, *callback=None*)
> > Defines a new command line option.
> >
> > If `type` is given (one of str, float, int, datetime, or timedelta) or can be inferred from the `default`, we parse the command line arguments based on the given type. If `multiple` is True, we accept comma-separated values, and the option value is always a list.
> >
> > For multi-value integers, we also accept the syntax `x:y`, which turns into `range(x, y)` - very useful for long integer ranges.
> >
> > `help` and `metavar` are used to construct the automatically generated command line help string. The help message is formatted like:
> >
> > ```
> > --name=METAVAR        help string
> > ```
> >
> > `group` is used to group the defined options in logical groups. By default, command line options are grouped by the file in which they are defined.
> >
> > Command line option names must be unique globally. They can be parsed from the command line with *parse_command_line* or parsed from a config file with *parse_config_file*.
> >
> > If a `callback` is given, it will be run with the new value whenever the option is changed. This can be used to combine command-line and file-based options:
> >
> > ```
> > define("config", type=str, help="path to config file",
> >        callback=lambda path: parse_config_file(path, final=False))
> > ```
> >
> > With this definition, options in the file specified by `--config` will override options set earlier on the command line, but can be overridden by later flags.

> **`group_dict`**(*group*)
> > The names and values of options in a group.
> >
> > Useful for copying options into Application settings:

```
from tornado.options import define, parse_command_line, options

define('template_path', group='application')
define('static_path', group='application')

parse_command_line()

application = Application(
    handlers, **options.group_dict('application'))
```

New in version 3.1.

groups()
    The set of option-groups created by `define`.

    New in version 3.1.

items()
    A sequence of (name, value) pairs.

    New in version 3.1.

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
```

parse_command_line(*args=None, final=True*)
    Parses all options given on the command line (defaults to `sys.argv`).

    Note that `args[0]` is ignored since it is the program name in `sys.argv`.

    We return a list of all arguments that are not parsed as options.

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

parse_config_file(*path, final=True*)
    Parses and loads the Python config file at the given path.

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

print_help(*file=None*)
    Prints all the command line options to stderr (or another file).

`tornado.process` — Utilities for multiple processes

Utilities for working with multiple processes, including both forking the server into multiple processes and managing subprocesses.

tornado.process.**cpu_count**()
    Returns the number of processors on this machine.

tornado.process.**fork_processes**(*num_processes*, *max_restarts=100*)

    Starts multiple worker processes.

    If `num_processes` is None or <= 0, we detect the number of cores available on this machine and fork that number of child processes. If `num_processes` is given and > 0, we fork that specific number of sub-processes.

    Since we use processes and not threads, there is no shared memory between any server code.

    Note that multiple processes are not compatible with the autoreload module (or the debug=True option to *tornado.web.Application*). When using multiple processes, no IOLoops can be created or referenced until after the call to `fork_processes`.

    In each child process, `fork_processes` returns its *task id*, a number between 0 and `num_processes`. Processes that exit abnormally (due to a signal or non-zero exit status) are restarted with the same id (up to `max_restarts` times). In the parent process, `fork_processes` returns None if all child processes have exited normally, but will otherwise only exit by throwing an exception.

tornado.process.**task_id**()

    Returns the current task id, if any.

    Returns None if this process was not created by *fork_processes*.

class tornado.process.**Subprocess**(**args*, ***kwargs*)

    Wraps `subprocess.Popen` with IOStream support.

    The constructor is the same as `subprocess.Popen` with the following additions:

        • `stdin`, `stdout`, and `stderr` may have the value `tornado.process.Subprocess.STREAM`, which will make the corresponding attribute of the resulting Subprocess a *PipeIOStream*.

        • A new keyword argument `io_loop` may be used to pass in an IOLoop.

    set_exit_callback(*callback*)

        Runs `callback` when this process exits.

        The callback takes one argument, the return code of the process.

        This method uses a SIGCHILD handler, which is a global setting and may conflict if you have other libraries trying to handle the same signal. If you are using more than one IOLoop it may be necessary to call *Subprocess.initialize* first to designate one IOLoop to run the signal handlers.

        In many cases a close callback on the stdout or stderr streams can be used as an alternative to an exit callback if the signal handler is causing a problem.

    classmethod initialize(*io_loop=None*)

        Initializes the SIGCHILD handler.

        The signal handler is run on an *IOLoop* to avoid locking issues. Note that the *IOLoop* used for signal handling need not be the same one used by individual Subprocess objects (as long as the IOLoops are each running in separate threads).

    classmethod uninitialize()

        Removes the SIGCHILD handler.

tornado.stack_context — Exception handling across asynchronous callbacks

StackContext allows applications to maintain threadlocal-like state that follows execution as it moves to other execution contexts.

The motivating examples are to eliminate the need for explicit `async_callback` wrappers (as in *`tornado.web.RequestHandler`*), and to allow some additional context to be kept for logging.

This is slightly magic, but it's an extension of the idea that an exception handler is a kind of stack-local state and when that stack is suspended and resumed in a new context that state needs to be preserved. *`StackContext`* shifts the burden of restoring that state from each call site (e.g. wrapping each *`AsyncHTTPClient`* callback in `async_callback`) to the mechanisms that transfer control from one context to another (e.g. *`AsyncHTTPClient`* itself, *`IOLoop`*, thread pools, etc).

Example usage:

```
@contextlib.contextmanager
def die_on_error():
    try:
        yield
    except Exception:
        logging.error("exception in asynchronous operation",exc_info=True)
        sys.exit(1)

with StackContext(die_on_error):
    # Any exception thrown here *or in callback and its desendents*
    # will cause the process to exit instead of spinning endlessly
    # in the ioloop.
    http_client.fetch(url, callback)
ioloop.start()
```

Most applications shouln't have to work with *`StackContext`* directly. Here are a few rules of thumb for when it's necessary:

- If you're writing an asynchronous library that doesn't rely on a stack_context-aware library like *`tornado.ioloop`* or *`tornado.iostream`* (for example, if you're writing a thread pool), use *`stack_context.wrap()`* before any asynchronous operations to capture the stack context from where the operation was started.

- If you're writing an asynchronous library that has some shared resources (such as a connection pool), create those shared resources within a `with stack_context.NullContext():` block. This will prevent `StackContexts` from leaking from one request to another.

- If you want to write something like an exception handler that will persist across asynchronous calls, create a new *`StackContext`* (or *`ExceptionStackContext`*), and make your asynchronous calls in a `with` block that references your *`StackContext`*.

class `tornado.stack_context.`**`StackContext`** (*context_factory*)

    Establishes the given context as a StackContext that will be transferred.

    Note that the parameter is a callable that returns a context manager, not the context itself. That is, where for a non-transferable context manager you would say:

```
with my_context():
```

    StackContext takes the function itself rather than its result:

```
with StackContext(my_context):
```

    The result of `with StackContext() as cb:` is a deactivation callback. Run this callback when the StackContext is no longer needed to ensure that it is not propagated any further (note that deactivating a context does not affect any instances of that context that are currently pending). This is an advanced feature and not necessary in most applications.

class `tornado.stack_context.`**`ExceptionStackContext`** (*exception_handler*)

    Specialization of StackContext for exception handling.

The supplied `exception_handler` function will be called in the event of an uncaught exception in this context. The semantics are similar to a try/finally clause, and intended use cases are to log an error, close a socket, or similar cleanup actions. The `exc_info` triple (`type`, `value`, `traceback`) will be passed to the exception_handler function.

If the exception handler returns true, the exception will be consumed and will not be propagated to other exception handlers.

class `tornado.stack_context.`**`NullContext`**

Resets the *StackContext*.

Useful when creating a shared resource on demand (e.g. an *AsyncHTTPClient*) where the stack that caused the creating is not relevant to future operations.

`tornado.stack_context.`**`wrap`**(*fn*)

Returns a callable object that will restore the current *StackContext* when executed.

Use this whenever saving a callback to be executed later in a different execution context (either in a different thread or asynchronously in the same thread).

`tornado.stack_context.`**`run_with_stack_context`**(*context*, *func*)

Run a coroutine `func` in the given *StackContext*.

It is not safe to have a `yield` statement within a `with StackContext` block, so it is difficult to use stack context with *gen.coroutine*. This helper function runs the function in the correct context while keeping the `yield` and `with` statements syntactically separate.

Example:

```
@gen.coroutine
def incorrect():
    with StackContext(ctx):
        # ERROR: this will raise StackContextInconsistentError
        yield other_coroutine()

@gen.coroutine
def correct():
    yield run_with_stack_context(StackContext(ctx), other_coroutine)
```

New in version 3.1.

`tornado.testing` — Unit testing support for asynchronous code

Support classes for automated testing.

- *AsyncTestCase* and *AsyncHTTPTestCase*: Subclasses of unittest.TestCase with additional support for testing asynchronous (*IOLoop* based) code.

- *ExpectLog* and *LogTrapTestCase*: Make test logs less spammy.

- *main()*: A simple test runner (wrapper around unittest.main()) with support for the tornado.autoreload module to rerun the tests when code changes.

Asynchronous test cases

class `tornado.testing.`**`AsyncTestCase`**(**args*, ***kwargs*)

TestCase subclass for testing *IOLoop*-based asynchronous code.

The unittest framework is synchronous, so the test must be complete by the time the test method returns. This means that asynchronous code cannot be used in quite the same way as usual. To write test functions that use the same `yield`-based patterns used with the *tornado.gen* module, decorate your test methods with *tornado.testing.gen_test* instead of *tornado.gen.coroutine*. This class also provides the *stop()* and *wait()* methods for a more manual style of testing. The test method itself must call `self.wait()`, and asynchronous callbacks should call `self.stop()` to signal completion.

By default, a new *IOLoop* is constructed for each test and is available as `self.io_loop`. This *IOLoop* should be used in the construction of HTTP clients/servers, etc. If the code being tested requires a global *IOLoop*, subclasses should override *get_new_ioloop* to return it.

The *IOLoop*'s `start` and `stop` methods should not be called directly. Instead, use *self.stop* and *self.wait*. Arguments passed to `self.stop` are returned from `self.wait`. It is possible to have multiple `wait`/`stop` cycles in the same test.

Example:

```
# This test uses coroutine style.
class MyTestCase(AsyncTestCase):
    @tornado.testing.gen_test
    def test_http_fetch(self):
        client = AsyncHTTPClient(self.io_loop)
        response = yield client.fetch("http://www.tornadoweb.org")
        # Test contents of response
        self.assertIn("FriendFeed", response.body)

# This test uses argument passing between self.stop and self.wait.
class MyTestCase2(AsyncTestCase):
    def test_http_fetch(self):
        client = AsyncHTTPClient(self.io_loop)
        client.fetch("http://www.tornadoweb.org/", self.stop)
        response = self.wait()
        # Test contents of response
        self.assertIn("FriendFeed", response.body)

# This test uses an explicit callback-based style.
class MyTestCase3(AsyncTestCase):
    def test_http_fetch(self):
        client = AsyncHTTPClient(self.io_loop)
        client.fetch("http://www.tornadoweb.org/", self.handle_fetch)
        self.wait()

    def handle_fetch(self, response):
        # Test contents of response (failures and exceptions here
        # will cause self.wait() to throw an exception and end the
        # test).
        # Exceptions thrown here are magically propagated to
        # self.wait() in test_http_fetch() via stack_context.
        self.assertIn("FriendFeed", response.body)
        self.stop()
```

get_new_ioloop()
> Creates a new *IOLoop* for this test. May be overridden in subclasses for tests that require a specific *IOLoop* (usually the singleton *IOLoop.instance()*).

stop(*_arg=None*, ***kwargs*)
> Stops the *IOLoop*, causing one pending (or future) call to *wait()* to return.

> Keyword arguments or a single positional argument passed to *stop()* are saved and will be returned by

> *wait()*.

wait(*condition=None*, *timeout=None*)
> Runs the *IOLoop* until stop is called or timeout has passed.
>
> In the event of a timeout, an exception will be thrown. The default timeout is 5 seconds; it may be overridden with a `timeout` keyword argument or globally with the `ASYNC_TEST_TIMEOUT` environment variable.
>
> If `condition` is not None, the *IOLoop* will be restarted after *stop()* until `condition()` returns true.
>
> Changed in version 3.1: Added the `ASYNC_TEST_TIMEOUT` environment variable.

class `tornado.testing.`**`AsyncHTTPTestCase`**(**args*, ***kwargs*)
> A test case that starts up an HTTP server.
>
> Subclasses must override *get_app()*, which returns the *tornado.web.Application* (or other *HTTPServer* callback) to be tested. Tests will typically use the provided `self.http_client` to fetch URLs from this server.
>
> Example:

```
class MyHTTPTest(AsyncHTTPTestCase):
    def get_app(self):
        return Application([('/', MyHandler)...])

    def test_homepage(self):
        # The following two lines are equivalent to
        #   response = self.fetch('/')
        # but are shown in full here to demonstrate explicit use
        # of self.stop and self.wait.
        self.http_client.fetch(self.get_url('/'), self.stop)
        response = self.wait()
        # test contents of response
```

> **fetch**(*path*, ***kwargs*)
> > Convenience method to synchronously fetch a url.
> >
> > The given path will be appended to the local server's host and port. Any additional kwargs will be passed directly to *AsyncHTTPClient.fetch* (and so could be used to pass `method="POST"`, `body="...", etc).
>
> **get_app**()
> > Should be overridden by subclasses to return a *tornado.web.Application* or other *HTTPServer* callback.
>
> **get_http_port**()
> > Returns the port used by the server.
> >
> > A new port is chosen for each test.
>
> **get_httpserver_options**()
> > May be overridden by subclasses to return additional keyword arguments for the server.
>
> **get_url**(*path*)
> > Returns an absolute url for the given path on the test server.

class `tornado.testing.`**`AsyncHTTPSTestCase`**(**args*, ***kwargs*)
> A test case that starts an HTTPS server.
>
> Interface is generally the same as *AsyncHTTPTestCase*.

get_ssl_options()
> May be overridden by subclasses to select SSL options.

> By default includes a self-signed testing certificate.

tornado.testing.**gen_test**(*func=None*, *timeout=None*)
> Testing equivalent of @gen.coroutine, to be applied to test methods.

> @gen.coroutine cannot be used on tests because the *IOLoop* is not already running. @gen_test should be applied to test methods on subclasses of *AsyncTestCase*.

> Example:

```
class MyTest(AsyncHTTPTestCase):
    @gen_test
    def test_something(self):
        response = yield gen.Task(self.fetch('/'))
```

> By default, @gen_test times out after 5 seconds. The timeout may be overridden globally with the ASYNC_TEST_TIMEOUT environment variable, or for each test with the timeout keyword argument:

```
class MyTest(AsyncHTTPTestCase):
    @gen_test(timeout=10)
    def test_something_slow(self):
        response = yield gen.Task(self.fetch('/'))
```

> New in version 3.1: The timeout argument and ASYNC_TEST_TIMEOUT environment variable.

Controlling log output

class tornado.testing.**ExpectLog**(*logger*, *regex*, *required=True*)
> Context manager to capture and suppress expected log output.

> Useful to make tests of error conditions less noisy, while still leaving unexpected log entries visible. *Not thread safe.*

> Usage:

```
with ExpectLog('tornado.application', "Uncaught exception"):
    error_response = self.fetch("/some_page")
```

> Constructs an ExpectLog context manager.

> **Parameters**

> - **logger** – Logger object (or name of logger) to watch. Pass an empty string to watch the root logger.

> - **regex** – Regular expression to match. Any log entries on the specified logger that match this regex will be suppressed.

> - **required** – If true, an exeption will be raised if the end of the with statement is reached without matching any log entries.

class tornado.testing.**LogTrapTestCase**(*methodName='runTest'*)
> A test case that captures and discards all logging output if the test passes.

> Some libraries can produce a lot of logging output even when the test succeeds, so this class can be useful to minimize the noise. Simply use it as a base class for your test case. It is safe to combine with AsyncTestCase via multiple inheritance (class MyTestCase(AsyncHTTPTestCase, LogTrapTestCase):)

This class assumes that only one log handler is configured and that it is a `StreamHandler`. This is true for both `logging.basicConfig` and the "pretty logging" configured by *tornado.options*. It is not compatible with other log buffering mechanisms, such as those provided by some test runners.

Create an instance of the class that will use the named test method when executed. Raises a ValueError if the instance does not have a method with the specified name.

Test runner

`tornado.testing.`**`main`**`(**kwargs)`
>   A simple test runner.

>   This test runner is essentially equivalent to `unittest.main` from the standard library, but adds support for tornado-style option parsing and log formatting.

>   The easiest way to run a test is via the command line:

```
python -m tornado.testing tornado.test.stack_context_test
```

>   See the standard library unittest module for ways in which tests can be specified.

>   Projects with many tests may wish to define a test script like `tornado/test/runtests.py`. This script should define a method `all()` which returns a test suite and then call *tornado.testing.main()*. Note that even when a test script is used, the `all()` test suite may be overridden by naming a single test on the command line:

```
# Runs all tests
python -m tornado.test.runtests
# Runs one test
python -m tornado.test.runtests tornado.test.stack_context_test
```

>   Additional keyword arguments passed through to `unittest.main()`. For example, use `tornado.testing.main(verbosity=2)` to show many test details as they are run. See http://docs.python.org/library/unittest.html#unittest.main for full argument list.

Helper functions

`tornado.testing.`**`bind_unused_port`**`()`
>   Binds a server socket to an available port on localhost.

>   Returns a tuple (socket, port).

`tornado.testing.`**`get_unused_port`**`()`
>   Returns a (hopefully) unused port number.

>   This function does not guarantee that the port it returns is available, only that a series of get_unused_port calls in a single process return distinct ports.

>   **Deprecated**. Use bind_unused_port instead, which is guaranteed to find an unused port.

`tornado.testing.`**`get_async_test_timeout`**`()`
>   Get the global timeout setting for async tests.

>   Returns a float, the timeout in seconds.

>   New in version 3.1.

`tornado.util` — General-purpose utilities

Miscellaneous utility functions and classes.

This module is used internally by Tornado. It is not necessarily expected that the functions and classes defined here will be useful to other applications, but they are documented here in case they are.

The one public-facing part of this module is the *Configurable* class and its *configure* method, which becomes a part of the interface of its subclasses, including *AsyncHTTPClient*, *IOLoop*, and *Resolver*.

class tornado.util.**ObjectDict**
> Makes a dictionary behave like an object, with attribute-style access.

class tornado.util.**GzipDecompressor**
> Streaming gzip decompressor.
>
> The interface is like that of `zlib.decompressobj` (without the optional arguments, but it understands gzip headers and checksums.
>
> **decompress**(*value*)
> > 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.
>
> **flush**()
> > Return any remaining buffered data not yet returned by decompress.
> >
> > Also checks for errors such as truncated input. No other methods may be called on this object after *flush*.

tornado.util.**import_object**(*name*)
> Imports an object by name.
>
> import_object('x') is equivalent to 'import x'. import_object('x.y.z') is equivalent to 'from x.y import z'.

```
>>> import tornado.escape
>>> import_object('tornado.escape') is tornado.escape
True
>>> import_object('tornado.escape.utf8') is tornado.escape.utf8
True
>>> import_object('tornado') is tornado
True
>>> import_object('tornado.missing_module')
Traceback (most recent call last):
    ...
ImportError: No module named missing_module
```

class tornado.util.**Configurable**
> Base class for configurable interfaces.
>
> A configurable interface is an (abstract) class whose constructor acts as a factory function for one of its implementation subclasses. The implementation subclass as well as optional keyword arguments to its initializer can be set globally at runtime with *configure*.
>
> By using the constructor as the factory method, the interface looks like a normal class, `isinstance` works as usual, etc. This pattern is most useful when the choice of implementation is likely to be a global decision (e.g. when `epoll` is available, always use it instead of `select`), or when a previously-monolithic class has been split into specialized subclasses.
>
> Configurable subclasses must define the class methods *configurable_base* and *configurable_default*, and use the instance method *initialize* instead of __init__.

classmethod configurable_base()

> Returns the base class of a configurable hierarchy.
>
> This will normally return the class in which it is defined. (which is *not* necessarily the same as the cls classmethod parameter).

classmethod configurable_default()

> Returns the implementation class to be used if none is configured.

initialize()

> Initialize a *Configurable* subclass instance.
>
> Configurable classes should use *initialize* instead of __init__.

classmethod configure(*impl*, ***kwargs*)

> Sets the class to use when the base class is instantiated.
>
> Keyword arguments will be saved and added to the arguments passed to the constructor. This can be used to set global defaults for some parameters.

classmethod configured_class()

> Returns the currently configured class.

class tornado.util.**ArgReplacer**(*func*, *name*)

> Replaces one value in an `args, kwargs` pair.

Inspects the function signature to find an argument by name whether it is passed by position or keyword. For use in decorators and similar wrappers.

replace(*new_value*, *args*, *kwargs*)

> Replace the named argument in `args, kwargs` with `new_value`.
>
> Returns (`old_value, args, kwargs`). The returned `args` and `kwargs` objects may not be the same as the input objects, or the input objects may be mutated.
>
> If the named argument was not found, `new_value` will be added to `kwargs` and None will be returned as `old_value`.

Release notes

What's new in Tornado 3.1

Jun 15, 2013

Multiple modules

- Many reference cycles have been broken up throughout the package, allowing for more efficient garbage collection on CPython.

- Silenced some log messages when connections are opened and immediately closed (i.e. port scans), or other situations related to closed connections.

- Various small speedups: *HTTPHeaders* case normalization, *UIModule* proxy objects, precompile some regexes.

`tornado.auth`

- *OAuthMixin* always sends oauth_version=1.0 in its request as required by the spec.

- *FacebookGraphMixin* now uses self._FACEBOOK_BASE_URL in *facebook_request* to allow the base url to be overridden.

- The authenticate_redirect and authorize_redirect methods in the *tornado.auth* mixin classes all now return Futures. These methods are asynchronous in *OAuthMixin* and derived classes, although they do not take a callback. The *Future* these methods return must be yielded if they are called from a function decorated with *gen.coroutine* (but not *gen.engine*).

- *TwitterMixin* now uses /account/verify_credentials to get information about the logged-in user, which is more robust against changing screen names.

- The demos directory (in the source distribution) has a new twitter demo using *TwitterMixin*.

tornado.escape

- *url_escape* and *url_unescape* have a new `plus` argument (defaulting to True for consistency with the previous behavior) which specifies whether they work like `urllib.parse.unquote` or `urllib.parse. unquote_plus`.

tornado.gen

- Fixed a potential memory leak with long chains of *tornado.gen* coroutines.

tornado.httpclient

- *tornado.httpclient.HTTPRequest* takes a new argument `auth_mode`, which can be either `basic` or `digest`. Digest authentication is only supported with `tornado.curl_httpclient`.
- `tornado.curl_httpclient` no longer goes into an infinite loop when pycurl returns a negative timeout.
- `curl_httpclient` now supports the `PATCH` and `OPTIONS` methods without the use of `allow_nonstandard_methods=True`.
- Worked around a class of bugs in libcurl that would result in errors from *IOLoop.update_handler* in various scenarios including digest authentication and socks proxies.
- The `TCP_NODELAY` flag is now set when appropriate in `simple_httpclient`.
- `simple_httpclient` no longer logs exceptions, since those exceptions are made available to the caller as `HTTPResponse.error`.

tornado.httpserver

- *tornado.httpserver.HTTPServer* handles malformed HTTP headers more gracefully.
- *HTTPServer* now supports lists of IPs in `X-Forwarded-For` (it chooses the last, i.e. nearest one).
- Memory is now reclaimed promptly on CPython when an HTTP request fails because it exceeded the maximum upload size.
- The `TCP_NODELAY` flag is now set when appropriate in *HTTPServer*.
- The *HTTPServer* `no_keep_alive` option is now respected with HTTP 1.0 connections that explicitly pass `Connection: keep-alive`.
- The `Connection: keep-alive` check for HTTP 1.0 connections is now case-insensitive.
- The `str` and `repr` of *tornado.httpserver.HTTPRequest* no longer include the request body, reducing log spam on errors (and potential exposure/retention of private data).

tornado.httputil

- The cache used in *HTTPHeaders* will no longer grow without bound.

tornado.ioloop

- Some *IOLoop* implementations (such as pyzmq) accept objects other than integer file descriptors; these objects will now have their .close() method called when the IOLoop` is closed with ``all_fds=True.
- The stub handles left behind by *IOLoop.remove_timeout* will now get cleaned up instead of waiting to expire.

tornado.iostream

- Fixed a bug in *BaseIOStream.read_until_close* that would sometimes cause data to be passed to the final callback instead of the streaming callback.
- The *IOStream* close callback is now run more reliably if there is an exception in _try_inline_read.
- New method *BaseIOStream.set_nodelay* can be used to set the TCP_NODELAY flag.
- Fixed a case where errors in SSLIOStream.connect (and SimpleAsyncHTTPClient) were not being reported correctly.

tornado.locale

- *Locale.format_date* now works on Python 3.

tornado.netutil

- The default *Resolver* implementation now works on Solaris.
- *Resolver* now has a *close* method.
- Fixed a potential CPU DoS when tornado.netutil.ssl_match_hostname is used on certificates with an abusive wildcard pattern.
- All instances of *ThreadedResolver* now share a single thread pool, whose size is set by the first one to be created (or the static Resolver.configure method).
- *ExecutorResolver* is now documented for public use.
- *bind_sockets* now works in configurations with incomplete IPv6 support.

tornado.options

- *tornado.options.define* with multiple=True now works on Python 3.
- *tornado.options.options* and other *OptionParser* instances support some new dict-like methods: *items()*, iteration over keys, and (read-only) access to options with square braket syntax. *OptionParser.group_dict* returns all options with a given group name, and *OptionParser.as_dict* returns all options.

tornado.process

- *tornado.process.Subprocess* no longer leaks file descriptors into the child process, which fixes a problem in which the child could not detect that the parent process had closed its stdin pipe.

- *Subprocess.set_exit_callback* now works for subprocesses created without an explicit io_loop parameter.

tornado.stack_context

- *tornado.stack_context* has been rewritten and is now much faster.

- New function *run_with_stack_context* facilitates the use of stack contexts with coroutines.

tornado.tcpserver

- The constructors of *TCPServer* and *HTTPServer* now take a max_buffer_size keyword argument.

tornado.template

- Some internal names used by the template system have been changed; now all "reserved" names in templates start with _tt_.

tornado.testing

- *tornado.testing.AsyncTestCase.wait* now raises the correct exception when it has been modified by *tornado.stack_context*.

- *tornado.testing.gen_test* can now be called as @gen_test(timeout=60) to give some tests a longer timeout than others.

- The environment variable ASYNC_TEST_TIMEOUT can now be set to override the default timeout for *AsyncTestCase.wait* and *gen_test*.

- *bind_unused_port* now passes None instead of 0 as the port to getaddrinfo, which works better with some unusual network configurations.

tornado.util

- *tornado.util.import_object* now works with top-level module names that do not contain a dot.

- *tornado.util.import_object* now consistently raises *ImportError* instead of *AttributeError* when it fails.

tornado.web

- The handlers list passed to the *tornado.web.Application* constructor and *add_handlers* methods can now contain lists in addition to tuples and *URLSpec* objects.

- *tornado.web.StaticFileHandler* now works on Windows when the client passes an If-Modified-Since timestamp before 1970.

- New method *RequestHandler.log_exception* can be overridden to customize the logging behavior when an exception is uncaught. Most apps that currently override _handle_request_exception can now use a combination of *RequestHandler.log_exception* and *write_error*.

- *RequestHandler.get_argument* now raises *MissingArgumentError* (a subclass of *tornado.web.HTTPError*, which is what it raised previously) if the argument cannot be found.

- *Application.reverse_url* now uses *url_escape* with plus=False, i.e. spaces are encoded as %20 instead of +.

- Arguments extracted from the url path are now decoded with *url_unescape* with plus=False, so plus signs are left as-is instead of being turned into spaces.

- *RequestHandler.send_error* will now only be called once per request, even if multiple exceptions are caught by the stack context.

- The *tornado.web.asynchronous* decorator is no longer necessary for methods that return a *Future* (i.e. those that use the *gen.coroutine* or *return_future* decorators)

- *RequestHandler.prepare* may now be asynchronous if it returns a *Future*. The *asynchronous* decorator is not used with prepare; one of the *Future*-related decorators should be used instead.

- RequestHandler.current_user may now be assigned to normally.

- *RequestHandler.redirect* no longer silently strips control characters and whitespace. It is now an error to pass control characters, newlines or tabs.

- *StaticFileHandler* has been reorganized internally and now has additional extension points that can be overridden in subclasses.

- *StaticFileHandler* now supports HTTP Range requests. *StaticFileHandler* is still not suitable for files too large to comfortably fit in memory, but Range support is necessary in some browsers to enable seeking of HTML5 audio and video.

- *StaticFileHandler* now uses longer hashes by default, and uses the same hashes for Etag as it does for versioned urls.

- *StaticFileHandler.make_static_url* and *RequestHandler.static_url* now have an additional keyword argument include_version to suppress the url versioning.

- *StaticFileHandler* now reads its file in chunks, which will reduce memory fragmentation.

- Fixed a problem with the Date header and cookie expiration dates when the system locale is set to a non-english configuration.

tornado.websocket

- *WebSocketHandler* now catches *StreamClosedError* and runs *on_close* immediately instead of logging a stack trace.

- New method *WebSocketHandler.set_nodelay* can be used to set the TCP_NODELAY flag.

tornado.wsgi

- Fixed an exception in *WSGIContainer* when the connection is closed while output is being written.

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.

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.

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 com-
  mand line or config file has been parsed. The keyword argument `final=False` can be used on either parsing
  function to supress these callbacks.

- *tornado.options.define* now takes a `callback` argument. This callback will be run with the new
  value whenever the option is changed. This is especially useful for options that set other options, such as by
  reading from a config file.

- *tornado.options.parse_command_line* `--help` output now goes to `stderr` rather than `stdout`.

- *tornado.options.options* is no longer a subclass of `dict`; attribute-style access is now required.

- *tornado.options.options* (and *OptionParser* instances generally) now have a *mockable()*
  method that returns a wrapper object compatible with `mock.patch`.

- Function `tornado.options.enable_pretty_logging` has been moved to the *tornado.log* mod-
  ule.

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)

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.

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.

What's new in Tornado 2.3

May 31, 2012

HTTP clients

- *`tornado.httpclient.HTTPClient`* now supports the same constructor keyword arguments as *`AsyncHTTPClient`*.

- The max_clients keyword argument to *`AsyncHTTPClient.configure`* now works.

- tornado.simple_httpclient now supports the OPTIONS and PATCH HTTP methods.

- tornado.simple_httpclient is better about closing its sockets instead of leaving them for garbage collection.

- tornado.simple_httpclient correctly verifies SSL certificates for URLs containing IPv6 literals (This bug affected Python 2.5 and 2.6).

- tornado.simple_httpclient no longer includes basic auth credentials in the Host header when those credentials are extracted from the URL.

- tornado.simple_httpclient no longer modifies the caller-supplied header dictionary, which caused problems when following redirects.

- tornado.curl_httpclient now supports client SSL certificates (using the same client_cert and client_key arguments as tornado.simple_httpclient)

HTTP Server

- *`HTTPServer`* now works correctly with paths starting with //

- HTTPHeaders.copy (inherited from *`dict.copy`*) now works correctly.

- `HTTPConnection.address` is now always the socket address, even for non-IP sockets. *`HTTPRequest.remote_ip`* is still always an IP-style address (fake data is used for non-IP sockets)

- Extra data at the end of multipart form bodies is now ignored, which fixes a compatibility problem with an iOS HTTP client library.

`IOLoop` **and** `IOStream`

- *`IOStream`* now has an `error` attribute that can be used to determine why a socket was closed.

- `tornado.iostream.IOStream.read_until` and `read_until_regex` are much faster with large input.

- `IOStream.write` performs better when given very large strings.

- *`IOLoop.instance()`* is now thread-safe.

`tornado.options`

- *`tornado.options`* options with `multiple=True` that are set more than once now overwrite rather than append. This makes it possible to override values set in `parse_config_file` with `parse_command_line`.

- *`tornado.options`* `--help` output is now prettier.

- *`tornado.options.options`* now supports attribute assignment.

`tornado.template`

- Template files containing non-ASCII (utf8) characters now work on Python 3 regardless of the locale environment variables.

- Templates now support `else` clauses in `try`/`except`/`finally`/`else` blocks.

`tornado.web`

- *`tornado.web.RequestHandler`* now supports the `PATCH` HTTP method. Note that this means any existing methods named `patch` in `RequestHandler` subclasses will need to be renamed.

- *`tornado.web.addslash`* and `removeslash` decorators now send permanent redirects (301) instead of temporary (302).

- *`RequestHandler.flush`* now invokes its callback whether there was any data to flush or not.

- Repeated calls to *`RequestHandler.set_cookie`* with the same name now overwrite the previous cookie instead of producing additional copies.

- `tornado.web.OutputTransform.transform_first_chunk` now takes and returns a status code in addition to the headers and chunk. This is a backwards-incompatible change to an interface that was never technically private, but was not included in the documentation and does not appear to have been used outside Tornado itself.

- Fixed a bug on python versions before 2.6.5 when *`URLSpec`* regexes are constructed from unicode strings and keyword arguments are extracted.

- The `reverse_url` function in the template namespace now comes from the *`RequestHandler`* rather than the *`Application`*. (Unless overridden, *`RequestHandler.reverse_url`* is just an alias for the *`Application`* method).

- The `Etag` header is now returned on 304 responses to an `If-None-Match` request, improving compatibility with some caches.
- *tornado.web* will no longer produce responses with status code 304 that also have entity headers such as `Content-Length`.

Other modules

- *tornado.auth.FacebookGraphMixin* no longer sends `post_args` redundantly in the url.
- The `extra_params` argument to *tornado.escape.linkify* may now be a callable, to allow parameters to be chosen separately for each link.
- *tornado.gen* no longer leaks `StackContexts` when a `@gen.engine` wrapped function is called repeatedly.
- *tornado.locale.get_supported_locales* no longer takes a meaningless `cls` argument.
- *StackContext* instances now have a deactivation callback that can be used to prevent further propagation.
- *tornado.testing.AsyncTestCase.wait* now resets its timeout on each call.
- *tornado.wsgi.WSGIApplication* now parses arguments correctly on Python 3.
- Exception handling on Python 3 has been improved; previously some exceptions such as `UnicodeDecodeError` would generate `TypeErrors`

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).

What's new in Tornado 2.2

Jan 30, 2012

Highlights

- Updated and expanded WebSocket support.
- Improved compatibility in the Twisted/Tornado bridge.
- Template errors now generate better stack traces.

- Better exception handling in `tornado.gen`.

Security fixes

- `tornado.simple_httpclient` now disables SSLv2 in all cases. Previously SSLv2 would be allowed if the Python interpreter was linked against a pre-1.0 version of OpenSSL.

Backwards-incompatible changes

- `tornado.process.fork_processes` now raises `SystemExit` if all child processes exit cleanly rather than returning `None`. The old behavior was surprising and inconsistent with most of the documented examples of this function (which did not check the return value).

- On Python 2.6, `tornado.simple_httpclient` only supports SSLv3. This is because Python 2.6 does not expose a way to support both SSLv3 and TLSv1 without also supporting the insecure SSLv2.

- `tornado.websocket` no longer supports the older "draft 76" version of the websocket protocol by default, although this version can be enabled by overriding `tornado.websocket.WebSocketHandler.allow_draft76`.

`tornado.httpclient`

- `SimpleAsyncHTTPClient` no longer hangs on `HEAD` requests, responses with no content, or empty `POST`/`PUT` response bodies.

- `SimpleAsyncHTTPClient` now supports 303 and 307 redirect codes.

- `tornado.curl_httpclient` now accepts non-integer timeouts.

- `tornado.curl_httpclient` now supports basic authentication with an empty password.

`tornado.httpserver`

- `HTTPServer` with `xheaders=True` will no longer accept `X-Real-IP` headers that don't look like valid IP addresses.

- `HTTPServer` now treats the `Connection` request header as case-insensitive.

`tornado.ioloop` and `tornado.iostream`

- `IOStream.write` now works correctly when given an empty string.

- `IOStream.read_until` (and `read_until_regex`) now perform better when there is a lot of buffered data, which improves peformance of `SimpleAsyncHTTPClient` when downloading files with lots of chunks.

- `SSLIOStream` now works correctly when `ssl_version` is set to a value other than `SSLv23`.

- Idle `IOLoops` no longer wake up several times a second.

- `tornado.ioloop.PeriodicCallback` no longer triggers duplicate callbacks when stopped and started repeatedly.

tornado.template

- Exceptions in template code will now show better stack traces that reference lines from the original template file.

- `{#` and `#}` can now be used for comments (and unlike the old `{% comment %}` directive, these can wrap other template directives).

- Template directives may now span multiple lines.

tornado.web

- Now behaves better when given malformed `Cookie` headers

- *RequestHandler.redirect* now has a `status` argument to send status codes other than 301 and 302.

- New method *RequestHandler.on_finish* may be overridden for post-request processing (as a counterpart to *RequestHandler.prepare*)

- *StaticFileHandler* now outputs `Content-Length` and `Etag` headers on `HEAD` requests.

- *StaticFileHandler* now has overridable `get_version` and `parse_url_path` methods for use in subclasses.

- *RequestHandler.static_url* now takes an `include_host` parameter (in addition to the old support for the `RequestHandler.include_host` attribute).

tornado.websocket

- Updated to support the latest version of the protocol, as finalized in RFC 6455.

- Many bugs were fixed in all supported protocol versions.

- *tornado.websocket* no longer supports the older "draft 76" version of the websocket protocol by default, although this version can be enabled by overriding *tornado.websocket.WebSocketHandler. allow_draft76*.

- *WebSocketHandler.write_message* now accepts a `binary` argument to send binary messages.

- Subprotocols (i.e. the `Sec-WebSocket-Protocol` header) are now supported; see the *WebSocketHandler.select_subprotocol* method for details.

- *WebSocketHandler.get_websocket_scheme* can be used to select the appropriate url scheme (`ws:/ /` or `wss://`) in cases where `HTTPRequest.protocol` is not set correctly.

Other modules

- *tornado.auth.TwitterMixin.authenticate_redirect* now takes a `callback_uri` parameter.

- *tornado.auth.TwitterMixin.twitter_request* now accepts both URLs and partial paths (complete URLs are useful for the search API which follows different patterns).

- Exception handling in *tornado.gen* has been improved. It is now possible to catch exceptions thrown by a `Task`.

- *tornado.netutil.bind_sockets* now works when `getaddrinfo` returns duplicate addresses.

- *tornado.platform.twisted* compatibility has been significantly improved. Twisted version 11.1.0 is now supported in addition to 11.0.0.

- *tornado.process.fork_processes* correctly reseeds the `random` module even when `os.urandom` is not implemented.

- *tornado.testing.main* supports a new flag `--exception_on_interrupt`, which can be set to false to make `Ctrl-C` kill the process more reliably (at the expense of stack traces when it does so).

- `tornado.version_info` is now a four-tuple so official releases can be distinguished from development branches.

What's new in Tornado 2.1.1

Oct 4, 2011

Bug fixes

- Fixed handling of closed connections with the `epoll` (i.e. Linux) `IOLoop`. Previously, closed connections could be shut down too early, which most often manifested as "Stream is closed" exceptions in `SimpleAsyncHTTPClient`.

- Fixed a case in which chunked responses could be closed prematurely, leading to truncated output.

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

- `IOStream.read_bytes` again accepts both `int` and `long` arguments.

- `PeriodicCallback` no longer runs repeatedly when `IOLoop` iterations complete faster than the resolution of `time.time()` (mainly a problem on Windows).

Backwards-compatibility note

- Listening for `IOLoop.ERROR` alone is no longer sufficient for detecting closed connections on an otherwise unused socket. `IOLoop.ERROR` must always be used in combination with `READ` or `WRITE`.

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

What's new in Tornado 2.0

Jun 21, 2011

```
Major changes:
* Template output is automatically escaped by default; see backwards
  compatibility note below.
* The default AsyncHTTPClient implementation is now simple_httpclient.
* Python 3.2 is now supported.

Backwards compatibility:
* Template autoescaping is enabled by default.  Applications upgrading from
  a previous release of Tornado must either disable autoescaping or adapt
  their templates to work with it.  For most applications, the simplest
  way to do this is to pass autoescape=None to the Application constructor.
  Note that this affects certain built-in methods, e.g. xsrf_form_html
  and linkify, which must now be called with {% raw %} instead of {}
* Applications that wish to continue using curl_httpclient instead of
  simple_httpclient may do so by calling
    AsyncHTTPClient.configure("tornado.curl_httpclient.CurlAsyncHTTPClient")
  at the beginning of the process.  Users of Python 2.5 will probably want
  to use curl_httpclient as simple_httpclient only supports ssl on Python 2.6+.
* Python 3 compatibility involved many changes throughout the codebase,
  so users are encouraged to test their applications more thoroughly than
  usual when upgrading to this release.

Other changes in this release:
* Templates support several new directives:
```

```
   - {% autoescape ...%} to control escaping behavior
   - {% raw ... %} for unescaped output
   - {% module ... %} for calling UIModules
* {% module Template(path, **kwargs) %} may now be used to call another
  template with an independent namespace
* All IOStream callbacks are now run directly on the IOLoop via add_callback.
* HTTPServer now supports IPv6 where available.  To disable, pass
  family=socket.AF_INET to HTTPServer.bind().
* HTTPClient now supports IPv6, configurable via allow_ipv6=bool on the
  HTTPRequest.  allow_ipv6 defaults to false on simple_httpclient and true
  on curl_httpclient.
* RequestHandlers can use an encoding other than utf-8 for query parameters
  by overriding decode_argument()
* Performance improvements, especially for applications that use a lot of
  IOLoop timeouts
* HTTP OPTIONS method no longer requires an XSRF token.
* JSON output (RequestHandler.write(dict)) now sets Content-Type to
  application/json
* Etag computation can now be customized or disabled by overriding
  RequestHandler.compute_etag
* USE_SIMPLE_HTTPCLIENT environment variable is no longer supported.
  Use AsyncHTTPClient.configure instead.
```

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.
```

What's new in Tornado 1.2

Feb 20, 2011

```
We are pleased to announce the release of Tornado 1.2, available from
https://github.com/downloads/facebook/tornado/tornado-1.2.tar.gz

Backwards compatibility notes:
* This release includes the backwards-incompatible security change from
  version 1.1.1.  Users upgrading from 1.1 or earlier should read the
  release notes from that release:
```

```
   http://groups.google.com/group/python-tornado/browse_thread/thread/b36191c781580cde
 * StackContexts that do something other than catch exceptions may need to
   be modified to be reentrant.
   https://github.com/facebook/tornado/commit/7a7e24143e77481d140fb5579bc67e4c45cbcfad
 * When XSRF tokens are used, the token must also be present on PUT and
   DELETE requests (anything but GET and HEAD)

New features:
 * A new HTTP client implementation is available in the module
   tornado.simple_httpclient.  This HTTP client does not depend on pycurl.
   It has not yet been tested extensively in production, but is intended
   to eventually replace the pycurl-based HTTP client in a future release of
   Tornado.  To transparently replace tornado.httpclient.AsyncHTTPClient with
   this new implementation, you can set the environment variable
   USE_SIMPLE_HTTPCLIENT=1 (note that the next release of Tornado will
   likely include a different way to select HTTP client implementations)
 * Request logging is now done by the Application rather than the
   RequestHandler.  Logging behavior may be customized by either overriding
   Application.log_request in a subclass or by passing log_function
   as an Application setting
 * Application.listen(port): Convenience method as an alternative to
   explicitly creating an HTTPServer
 * tornado.escape.linkify(): Wrap urls in <a> tags
 * RequestHandler.create_signed_value(): Create signatures like the
   secure_cookie methods without setting cookies.
 * tornado.testing.get_unused_port(): Returns a port selected in the same
   way as inAsyncHTTPTestCase
 * AsyncHTTPTestCase.fetch(): Convenience method for synchronous fetches
 * IOLoop.set_blocking_signal_threshold(): Set a callback to be run when
   the IOLoop is blocked.
 * IOStream.connect(): Asynchronously connect a client socket
 * AsyncHTTPClient.handle_callback_exception(): May be overridden
   in subclass for custom error handling
 * httpclient.HTTPRequest has two new keyword arguments, validate_cert and
   ca_certs. Setting validate_cert=False will disable all certificate checks
   when fetching https urls.  ca_certs may be set to a filename containing
   trusted certificate authorities (defaults will be used if this is
   unspecified)
 * HTTPRequest.get_ssl_certificate(): Returns the client's SSL certificate
   (if client certificates were requested in the server's ssl_options
 * StaticFileHandler can be configured to return a default file (e.g.
   index.html) when a directory is requested
 * Template directives of the form "{% from x import y %}" are now
   supported (in addition to the existing support for "{% import x
   %}"
 * FacebookGraphMixin.get_authenticated_user now accepts a new
   parameter 'extra_fields' which may be used to request additional
   information about the user

Bug fixes:
 * auth: Fixed KeyError with Facebook offline_access
 * auth: Uses request.uri instead of request.path as the default redirect
   so that parameters are preserved.
 * escape: xhtml_escape() now returns a unicode string, not
   utf8-encoded bytes
 * ioloop: Callbacks added with add_callback are now run in the order they
   were added
 * ioloop: PeriodicCallback.stop can now be called from inside the callback.
```

```
* iostream: Fixed several bugs in SSLIOStream
* iostream: Detect when the other side has closed the connection even with
  the select()-based IOLoop
* iostream: read_bytes(0) now works as expected
* iostream: Fixed bug when writing large amounts of data on windows
* iostream: Fixed infinite loop that could occur with unhandled exceptions
* httpclient: Fix bugs when some requests use proxies and others don't
* httpserver: HTTPRequest.protocol is now set correctly when using the
  built-in SSL support
* httpserver: When using multiple processes, the standard library's
  random number generator is re-seeded in each child process
* httpserver: With xheaders enabled, X-Forwarded-Proto is supported as an
  alternative to X-Scheme
* httpserver: Fixed bugs in multipart/form-data parsing
* locale: format_date() now behaves sanely with dates in the future
* locale: Updates to the language list
* stack_context: Fixed bug with contexts leaking through reused IOStreams
* stack_context: Simplified semantics and improved performance
* web: The order of css_files from UIModules is now preserved
* web: Fixed error with default_host redirect
* web: StaticFileHandler works when os.path.sep != '/' (i.e. on Windows)
* web: Fixed a caching-related bug in StaticFileHandler when a file's
  timestamp has changed but its contents have not.
* web: Fixed bugs with HEAD requests and e.g. Etag headers
* web: Fix bugs when different handlers have different static_paths
* web: @removeslash will no longer cause a redirect loop when applied to the
  root path
* websocket: Now works over SSL
* websocket: Improved compatibility with proxies

Many thanks to everyone who contributed patches, bug reports, and feedback
that went into this release!

-Ben
```

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
```

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
```

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.
```

What's new in Tornado 1.0

July 22, 2010

```
We are pleased to announce the release of Tornado 1.0, available
from
https://github.com/downloads/facebook/tornado/tornado-1.0.tar.gz.
There have been many changes since version 0.2; here are some of
the highlights:

New features:
* Improved support for running other WSGI applications in a
  Tornado server (tested with Django and CherryPy)
* Improved performance on Mac OS X and BSD (kqueue-based IOLoop),
  and experimental support for win32
* Rewritten AsyncHTTPClient available as
  tornado.httpclient.AsyncHTTPClient2 (this will become the
  default in a future release)
* Support for standard .mo files in addition to .csv in the locale
  module
* Pre-forking support for running multiple Tornado processes at
  once (see HTTPServer.start())
* SSL and gzip support in HTTPServer
* reverse_url() function refers to urls from the Application
  config by name from templates and RequestHandlers
* RequestHandler.on_connection_close() callback is called when the
  client has closed the connection (subject to limitations of the
  underlying network stack, any proxies, etc)
* Static files can now be served somewhere other than /static/ via
  the static_url_prefix application setting
* URL regexes can now use named groups ("(?P<name>)") to pass
  arguments to get()/post() via keyword instead of position
* HTTP header dictionary-like objects now support multiple values
  for the same header via the get_all() and add() methods.
* Several new options in the httpclient module, including
  prepare_curl_callback and header_callback
* Improved logging configuration in tornado.options.
* UIModule.html_body() can be used to return html to be inserted
  at the end of the document body.

Backwards-incompatible changes:
* RequestHandler.get_error_html() now receives the exception
  object as a keyword argument if the error was caused by an
  uncaught exception.
* Secure cookies are now more secure, but incompatible with
```

```
  cookies set by Tornado 0.2.  To read cookies set by older
  versions of Tornado, pass include_name=False to
  RequestHandler.get_secure_cookie()
* Parameters passed to RequestHandler.get/post() by extraction
  from the path now have %-escapes decoded, for consistency with
  the processing that was already done with other query
  parameters.

Many thanks to everyone who contributed patches, bug reports, and
feedback that went into this release!

-Ben
```

This documentation is also available in PDF and Epub formats.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

t