
Tornado Documentation

Release 2.4.1

Facebook

February 19, 2015

1	Upgrading from Tornado 1.x	3
2	Quick links	5
3	Hello, world	7
4	Installation	9
5	Discussion and support	11
5.1	Tornado Documentation	11
5.2	Indices and tables	95
	Python Module Index	97

Tornado is an open source version of the scalable, non-blocking web server and tools that power FriendFeed. 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 underlying non-blocking infrastructure.

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 it 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](#).)

Upgrading from Tornado 1.x

Tornado 2.0 introduces several potentially backwards-incompatible changes, including in particular automatic escaping of template output. Users who are upgrading from Tornado 1.x should see the *version 2.0 release notes* for information about backwards compatibility.

Quick links

- *Documentation*
- Download version 2.4.1: [tornado-2.4.1.tar.gz](#) (*release notes*)
- [Source \(github\)](#)
- [Mailing list](#)
- [Wiki](#)

Hello, world

Here is the canonical “Hello, world” example app for Tornado:

```
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()
```

See the [Tornado documentation](#) for a detailed walkthrough of the framework.

Installation

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

Manual installation: Download [tornado-2.4.1.tar.gz](#):

```
tar xvzf tornado-2.4.1.tar.gz
cd tornado-2.4.1
python setup.py build
sudo python setup.py install
```

The Tornado source code is [hosted on GitHub](#). On Python 2.6 and 2.7, it is also possible to simply add the `tornado` directory to your `PYTHONPATH` instead of building with `setup.py`, since the standard library includes `epoll` support.

Prerequisites: Tornado runs on Python 2.5, 2.6, 2.7 and 3.2.

- On Python 2.6 and 2.7, there are no dependencies outside the Python standard library, although [PycURL](#) (version 7.18.2 or higher required; version 7.21.1 or higher recommended) may be used if desired.
- On Python 2.5, [PycURL](#) is required, along with [simplejson](#) and the Python development headers (typically obtained by installing a package named something like `python-dev` from your operating system).
- On Python 3.2, the [distribute](#) package is required. Note that Python 3 support is relatively new and may have bugs.

Platforms: Tornado should run on any Unix-like platform, although for the best performance and scalability only Linux and BSD (including BSD derivatives like Mac OS X) are recommended.

Discussion and support

You can discuss Tornado and report bugs on the [Tornado developer mailing list](#). Links to additional resources can be found on the [Tornado wiki](#).

Tornado is one of [Facebook's open source technologies](#). It is available under the [Apache License, Version 2.0](#).

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

5.1 Tornado Documentation

5.1.1 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:

```
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 surrounded by `{% and %}`, e.g., `{% if len(items) > 2 %}`. Expressions are surrounded by `{{ and }}`, e.g., `{{ items[0] }}`.

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

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

- `escape`: alias for `tornado.escape.xhtml_escape`
- `xhtml_escape`: alias for `tornado.escape.xhtml_escape`
- `url_escape`: alias for `tornado.escape.url_escape`
- `json_encode`: alias for `tornado.escape.json_encode`
- `squeeze`: alias for `tornado.escape.squeeze`
- `linkify`: alias for `tornado.escape.linkify`
- `datetime`: the Python `datetime` module
- `handler`: the current `RequestHandler` object
- `request`: alias for `handler.request`
- `current_user`: alias for `handler.current_user`
- `locale`: alias for `handler.locale`
- `_`: alias for `handler.locale.translate`
- `static_url`: alias for `handler.static_url`
- `xsrform_html`: alias for `handler.xsrform_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 `xsrp_cookies`:

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

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

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

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

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

jQuery.postJSON = function(url, args, callback) {
    args._xsrp = getCookie("_xsrp");
    $.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`.

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

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

```
<html>
  <head>
    <title>FriendFeed - {{ _("Home") }}</title>
  </head>
  <body>
    <div></div>
  </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 mode. In this mode, templates will not be cached and 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.

Debug mode is not compatible with `HTTPServer`'s multi-process mode. You must not give `HTTPServer.start` an argument greater than 1 if you are using debug mode.

The automatic reloading feature of debug mode is available as a standalone module in `tornado.autoreload`, and is optionally used by the test runner in `tornado.testing.main`.

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.

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.

5.1.2 Core web framework

`tornado.web` — RequestHandler and Application classes

The Tornado web framework looks a bit like `web.py` (<http://webpy.org/>) or Google's `webapp` (<http://code.google.com/appengine/docs/python/tools/webapp/>), but with additional tools and optimizations to take advantage of the Tornado non-blocking web server and tools.

Here is the canonical “Hello, world” example app:

```
import tornado.ioloop
import tornado.web

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

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

See the Tornado walkthrough on <http://tornadoweb.org> 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` class.

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.

`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 to handle the corresponding HTTP method.

`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 throw an HTTP 400 exception 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.

Output

`RequestHandler.set_status` (*status_code*)

Sets the status code for our response.

`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 string. 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, 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`

`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.

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.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*)

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).

We append `?v=<signature>` to the returned URL, which makes our static file handler set an infinite expiration header on the returned content. The signature is based on the content of the file.

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.

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 constructor of the handler. This pattern is used for the `StaticFileHandler` below (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`
- `xcsrf_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. preforking), do not use this method; create an HTTPServer and call its bind/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.

Note that host patterns are processed sequentially in the order they were added, and only the first matching pattern is used. This means that all handlers for a given host must be added in a single add_handlers call.

add_transform (*transform_class*)

Adds the given OutputTransform to our transform list.

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.

Creates a URLSpec.

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.

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.

```
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()

```

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

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

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

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

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

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

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

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

Everything else

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

An exception that will turn into an HTTP error response.

class `tornado.web.UIModule` (*handler*)

A UI re-usable, modular 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.

To map a path to this handler for a static data directory `/var/www`, you would add a line to your application like:

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

The local root directory of the content should be passed as the “path” argument to the handler.

To support aggressive browser caching, if the argument “v” is given with the path, we set an infinite HTTP expiration header. So, if you want browsers to cache a file indefinitely, send them to, e.g., `/static/images/myimage.png?v=xxx`. Override `get_cache_time` method for more fine-grained cache control.

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 trigger aggressive caching or 0 to mark resource as cacheable, only.

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

classmethod make_static_url (*settings, path*)

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).

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

classmethod get_version (*settings, path*)

Generate the version string to be used in static URLs.

This method may be overridden in subclasses (but note that it is a class method rather than a static method). The default implementation uses a hash of the file’s contents.

`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.

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`.

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

`HTTPHeader` 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` header

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 “`connection`” 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, **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 httpserver
import 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 = httpserver.HTTPServer(handle_request)
http_server.listen(8888)
ioloop.IOLoop.instance().start()
```

`HTTPServer` is a very basic connection handler. Beyond parsing the HTTP request body and headers, the only HTTP semantics implemented in `HTTPServer` is HTTP/1.1 keep-alive connections. We do not, however, implement chunked encoding, so the request callback must provide a `Content-Length` header or implement chunked encoding for HTTP/1.1 requests for the server to run correctly for HTTP/1.1 clients. If the request handler is unable to do this, you can provide the `no_keep_alive` argument to the `HTTPServer` constructor, which will ensure the connection is closed on every request no matter what HTTP version the client is using.

If `xheaders` is `True`, we support the `X-Real-IP` and `X-Scheme` headers, which override the remote IP and HTTP scheme for all requests. These headers are useful when running Tornado behind a reverse proxy or load balancer.

`HTTPServer` can serve SSL traffic with Python 2.6+ and OpenSSL. 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`”:

```
HTTPServer(application, 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.netutil.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)
```

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

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

```
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.

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 ** 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 ** = *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=<object object at 0x7fcd1a529a00>)
```

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.

Creates a template loader.

`root_directory` may be the empty string if this loader does not use the filesystem.

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

reset ()

Resets the cache of compiled templates.

resolve_path (*name*, *parent_path=None*)

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

load (*name*, *parent_path=None*)

Loads a template.

class tornado.template.**Loader** (*root_directory*, ***kwargs*)

A template loader that loads from a single root directory.

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

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 XML or XHTML.

tornado.escape.**xhtml_unescape** (*value*)

Un-escapes an XML-escaped string.

tornado.escape.**url_escape** (*value*)

Returns a valid URL-encoded version of the given value.

tornado.escape.**url_unescape** (*value*, *encoding='utf-8'*)

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.

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 http://tornadoweb.org!`

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

`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()` below.

`tornado.locale.set_default_locale` (*code*)

Sets the default locale, used in `get_closest_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 cyclone file1.py file2.html etc`
2. **Merge against existing POT file:** `msgmerge old.po cyclone.po > new.po`
3. **Compile:** `msgfmt cyclone.po -o {directory}/pt_BR/LC_MESSAGES/cyclone.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.

5.1.3 Asynchronous networking

`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` (*impl=None*)
 A level-triggered I/O loop.

We use `epoll` (Linux) or `kqueue` (BSD and Mac OS X; requires python 2.6+) 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 `queue`.

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.instance()`

Returns a global IOLoop instance.

Most single-threaded applications have a single, global IOLoop. Use this method instead of passing around IOLoop instances throughout your code.

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.instance()
```

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 I/O handlers calls `stop()`, which will make the loop stop after the current event iteration completes.

`IOLoop.running()`

Returns true if this IOLoop is currently running.

`IOLoop.stop()`

Stop the loop after the current event loop iteration is complete. 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.

`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`.

I/O events

`IOLoop.add_handler` (*fd*, *handler*, *events*)

Registers the given handler to receive the given events for *fd*.

`IOLoop.update_handler` (*fd*, *events*)

Changes the events we listen for *fd*.

`IOLoop.remove_handler` (*fd*)

Stop listening for events on *fd*.

Timeouts

`IOLoop.add_callback` (*callback*)

Calls the given callback on the next I/O loop iteration.

It is safe to call this method from any thread at any time. Note that this is the *only* method in `IOLoop` that makes this 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.

`IOLoop.add_timeout` (*deadline*, *callback*)

Calls the given callback at the time *deadline* from the I/O loop.

Returns a handle that may be passed to `remove_timeout` to cancel.

deadline may be a number denoting a unix timestamp (as returned by `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`.

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 python '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

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

`class tornado.iostream.IOStream(socket, io_loop=None, max_buffer_size=104857600, read_chunk_size=4096)`
A utility class to write to and read from a non-blocking 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).

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`.

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

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

```
from tornado import ioloop
from tornado import iostream
import socket

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

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

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

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

`connect(address, callback=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.

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.

read_until_regex (*regex, callback*)

Call callback when we read the given regex pattern.

read_until (*delimiter, callback*)

Call callback when we read the given delimiter.

read_bytes (*num_bytes, callback, streaming_callback=None*)

Call 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.

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.

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

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.

set_close_callback (*callback*)

Call the given callback when the stream is closed.

close ()

Close this stream.

reading ()

Returns true if we are currently reading from the stream.

writing ()

Returns true if we are currently writing to the stream.

closed ()

Returns true if the stream has been closed.

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.

Creates an `SSLIOStream`.

If a dictionary is provided as keyword argument `ssl_options`, it will be used as additional keyword arguments to `ssl.wrap_socket`.

tornado.httpclient — Non-blocking 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.
- `simple_httpclient` only supports SSL on Python 2.6 and above.
- `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, e:
    print "Error:", e
```

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`
A non-blocking HTTP client.

Example usage:

```
import ioloop

def handle_request(response):
```

```

if response.error:
    print "Error:", response.error
else:
    print response.body
ioloop.IOLoop.instance().stop()

http_client = httpclient.AsyncHTTPClient()
http_client.fetch("http://www.google.com/", handle_request)
ioloop.IOLoop.instance().start()

```

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, **kwargs*)

Executes a request, calling `callback` with 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, the `HTTPResponse` given to the callback has a non-None error attribute that contains the exception encountered during the request. You can call `response.rethrow()` to throw the exception (if any) in the callback.

static 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,
                                     connect_timeout=20.0, request_timeout=20.0,
                                     if_modified_since=None, follow_redirects=True,
                                     max_redirects=5, user_agent=None, use_gzip=True,
                                     network_interface=None, streaming_callback=None,
                                     header_callback=None, prepare_curl_callback=None,
                                     proxy_host=None, proxy_port=None,
                                     proxy_username=None, proxy_password='',
                                     allow_nonstandard_methods=False, validate_cert=True,
                                     ca_certs=None, allow_ipv6=None, client_key=None,
                                     client_cert=None)
```

HTTP client request object.

Creates an `HTTPRequest`.

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
- **auth_username** (*string*) – Username for HTTP “Basic” authentication
- **auth_password** (*string*) – Password for HTTP “Basic” authentication
- **connect_timeout** (*float*) – Timeout for initial connection in seconds
- **request_timeout** (*float*) – Timeout for entire request in seconds
- **if_modified_since** (*datetime*) – 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 `body` and `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, and `headers` will be empty in the final response.
- **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 support 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

Response objects

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

HTTP Response object.

Attributes:

- **request**: HTTPRequest object
- **code**: numeric HTTP status code, e.g. 200 or 404
- **headers**: `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.

class `tornado.netutil.TCPServer` (*io_loop=None, ssl_options=None*)
A non-blocking, single-threaded TCP server.

To use `TCPServer`, define a subclass which overrides the `handle_stream` method.

`TCPServer` can serve SSL traffic with Python 2.6+ and OpenSSL. 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`.

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.

`tornado.netutil.bind_sockets` (*port*, *address*=None, *family*=0, *backlog*=128)

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()`.

`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.

5.1.4 Integration with other services

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

Implementations of various third-party authentication schemes.

All the classes in this file are class Mixins designed to be used with `web.py` `RequestHandler` classes. The primary methods for each service are `authenticate_redirect()`, `authorize_redirect()`, and `get_authenticated_user()`. The former should be called to redirect the user to, e.g., the OpenID authentication page on the third party service, and the latter should be called upon return to get the user data from the data returned by the third party service.

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 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.async_callback(self._on_auth))
            return
        self.authenticate_redirect()

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

Common protocols

`class tornado.auth.OpenIdMixin`

Abstract implementation of OpenID and Attribute Exchange.

See GoogleMixin below for example implementations.

authenticate_redirect (*callback_uri=None, ax_attrs=['name', 'email', 'language', 'username']*)

Returns the authentication URL for this service.

After authentication, the service will redirect back to the given callback URI.

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.

get_authenticated_user (*callback, http_client=None*)

Fetches the authenticated user data upon redirect.

This method should be called by the handler that receives the redirect from the `authenticate_redirect()` or `authorize_redirect()` methods.

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.

See TwitterMixin and FriendFeedMixin below for example implementations.

authorize_redirect (*callback_uri=None, extra_params=None, http_client=None*)

Redirects the user to obtain OAuth authorization for this service.

Twitter and FriendFeed both require that you register a Callback URL with your application. You should call this method to log the user in, and then call `get_authenticated_user()` in the handler you registered as your Callback URL to complete the authorization process.

This method sets a cookie called `_oauth_request_token` which is subsequently used (and cleared) in `get_authenticated_user` for security purposes.

get_authenticated_user (*callback, http_client=None*)

Gets the OAuth authorized user and access token on callback.

This method should be called from the handler for your registered OAuth Callback URL to complete the registration process. We call `callback` with the authenticated user, which in addition to standard attributes like `'name'` includes the `'access_key'` attribute, which contains the OAuth access you can use to make authorized requests to this service on behalf of the user.

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 v 2.

authorize_redirect (*redirect_uri=None, client_id=None, client_secret=None, extra_params=None*)

Redirects the user to obtain OAuth authorization for this service.

Some providers require that you register a Callback URL with your application. You should call this method to log the user in, and then call `get_authenticated_user()` in the handler you registered as your Callback URL to complete the authorization process.

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 TwitterHandler(tornado.web.RequestHandler,
                    tornado.auth.TwitterMixin):
    @tornado.web.asynchronous
    def get(self):
        if self.get_argument("oauth_token", None):
            self.get_authenticated_user(self.async_callback(self._on_auth))
            return
        self.authorize_redirect()

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

The user object returned by `get_authenticated_user()` includes the attributes 'username', 'name', and all of the custom Twitter user attributes describe at <http://apiwiki.twitter.com/Twitter-REST-API-Method%3A-users%C2%A0show> 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 `twitter_request()`.

authenticate_redirect (*callback_uri=None*)

Just like `authorize_redirect()`, but auto-redirects if authorized.

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

twitter_request (*path, callback, access_token=None, post_args=None, **args*)

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

The path should not include the format (we automatically append `".json"` and parse the JSON output).

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://apiwiki.twitter.com/Twitter-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.TwitterMixin):
    @tornado.web.authenticated
    @tornado.web.asynchronous
    def get(self):
        self.twitter_request(
            "/statuses/update",
            post_args={"status": "Testing Tornado Web Server"},
            access_token=user["access_token"],
            callback=self.async_callback(self._on_post))
```

```
def _on_post(self, new_entry):
    if not new_entry:
        # Call failed; perhaps missing permission?
        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 FriendFeedHandler(tornado.web.RequestHandler,
                       tornado.auth.FriendFeedMixin):
    @tornado.web.asynchronous
    def get(self):
        if self.get_argument("oauth_token", None):
            self.get_authenticated_user(self.async_callback(self._on_auth))
            return
        self.authorize_redirect()

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

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 (*path, callback, access_token=None, post_args=None, **args*)

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
    def get(self):
        self.friendfeed_request(
            "/entry",
            post_args={"body": "Testing Tornado Web Server"},
```

```
        access_token=self.current_user["access_token"],
        callback=self.async_callback(self._on_post))

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

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. To authenticate with Google, redirect with `authenticate_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 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.async_callback(self._on_auth))
            return
        self.authenticate_redirect()

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

authorize_redirect (*oauth_scope*, *callback_uri=None*, *ax_attrs=['name', 'email', 'language', 'username']*)
Authenticates and authorizes for the given Google resource.

Some of the available resources 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.

get_authenticated_user (*callback*)
Fetches the authenticated user data upon redirect.

Facebook

class tornado.auth.**FacebookMixin**
Facebook Connect authentication.

New applications should consider using `FacebookGraphMixin` below instead of this class.

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
        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 (*callback_uri=None, cancel_uri=None, extended_permissions=None*)
Authenticates/installs this app for the current user.

authorize_redirect (*extended_permissions, callback_uri=None, cancel_uri=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()`.

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.

class tornado.auth.FacebookGraphMixin

Facebook authentication using the new Graph API and OAuth2.

get_authenticated_user(*redirect_uri*, *client_id*, *client_secret*, *code*, *callback*, *extra_fields=None*)

Handles the login for the Facebook user, returning a user object.

Example usage:

```
class FacebookGraphLoginHandler(LoginHandler, tornado.auth.FacebookGraphMixin):
    @tornado.web.asynchronous
    def get(self):
        if self.get_argument("code", False):
            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"),
                callback=self.async_callback(
                    self._on_login))
            return
            self.authorize_redirect(redirect_uri='/auth/facebookgraph/',
                                   client_id=self.settings["facebook_api_key"],
                                   extra_params={"scope": "read_stream,offline_access"})

    def _on_login(self, user):
        logging.error(user)
        self.finish()
```

facebook_request(*path*, *callback*, *access_token=None*, *post_args=None*, ***args*)

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

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

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

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

```

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

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

```

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.

tornado.database — Simple MySQL client wrapper

Warning: The `tornado.database` module is deprecated and will be removed in Tornado 3.0. It is now available separately as the `torndb` module.

A lightweight wrapper around MySQLdb.

```

class tornado.database.Connection(host, database, user=None, password=None,
                                 max_idle_time=25200)

```

A lightweight wrapper around MySQLdb DB-API connections.

The main value we provide is wrapping rows in a dict/object so that columns can be accessed by name. Typical usage:

```

db = database.Connection("localhost", "mydatabase")
for article in db.query("SELECT * FROM articles"):
    print article.title

```

Cursors are hidden by the implementation, but other than that, the methods are very similar to the DB-API.

We explicitly set the timezone to UTC and the character encoding to UTF-8 on all connections to avoid time zone and encoding errors.

close()

Closes this database connection.

reconnect()

Closes the existing database connection and re-opens it.

iter(query, *parameters)

Returns an iterator for the given query and parameters.

query(query, *parameters)

Returns a row list for the given query and parameters.

get(query, *parameters)

Returns the first row returned for the given query.

execute (*query*, **parameters*)

Executes the given query, returning the lastrowid from the query.

execute_lastrowid (*query*, **parameters*)

Executes the given query, returning the lastrowid from the query.

execute_rowcount (*query*, **parameters*)

Executes the given query, returning the rowcount from the query.

executemany (*query*, *parameters*)

Executes the given query against all the given param sequences.

We return the lastrowid from the query.

executemany_lastrowid (*query*, *parameters*)

Executes the given query against all the given param sequences.

We return the lastrowid from the query.

executemany_rowcount (*query*, *parameters*)

Executes the given query against all the given param sequences.

We return the rowcount from the query.

class tornado.database.**Row**

A dict that allows for object-like property access syntax.

tornado.platform.twisted — Run code written for Twisted on Tornado

This module contains a Twisted reactor build on the Tornado IOLoop, which lets you run applications and libraries written for Twisted in a Tornado application. 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()`. This will allow you to use a mixture of Twisted and Tornado code in the same process.

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.

This module has been tested with Twisted versions 11.0.0 and 11.1.0.

```
tornado.platform.twisted.install(io_loop=None)
```

Install this package as the default Twisted reactor.

class tornado.platform.twisted.**TornadoReactor** (*io_loop=None*)

Twisted reactor built on the Tornado IOLoop.

Since it is intended to be used in applications where the top-level event loop is `io_loop.start()` rather than `reactor.run()`, it is implemented a little differently than other Twisted reactors. We override `mainLoop` instead of `doIteration` and must implement timed call functionality on top of `IOLoop.add_timeout` rather than using the implementation in `PosixReactorBase`.

tornado.websocket — Bidirectional communication to the browser

Server-side 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](http://tools.ietf.org/html/rfc6455) 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. 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 Web Socket 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"
```

Web Sockets 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()` 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.

Other

`WebSocketHandler.async_callback` (*callback*, **args*, ***kwargs*)
Wrap callbacks with this if they are used on asynchronous requests.

Catches exceptions properly and closes this WebSocket if an exception is uncaught. (Note that this is usually unnecessary thanks to `tornado.stack_context`)

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 `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:

```
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 AppEngine.

Since no asynchronous methods are available for WSGI applications, the `httpclient` and `auth` modules are both not available for WSGI applications. We support the same interface, but handlers running in a `WSGIApplication` do not support `flush()` or asynchronous methods.

class `tornado.wsgi.HTTPRequest` (*environ*)
Mimics `tornado.httpserver.HTTPRequest` for WSGI applications.

Parses the given WSGI environ 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:

```
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.

5.1.5 Utilities

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

A module to automatically restart the server when a module is modified.

Most applications should not call 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.

This module depends on `IOLoop`, so it will not work in WSGI applications and Google AppEngine. It also will not work correctly when `HTTPServer`'s multi-process mode is used.

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

`tornado.autoreload.start` (*io_loop=None, check_time=500*)

Restarts the process automatically when a module is modified.

We run on the I/O loop, and restarting is a destructive operation, so will terminate any pending requests.

`tornado.autoreload.wait` ()

Wait for a watched file to change, then restart the process.

Intended to be used at the end of scripts like unit test runners, to run the tests again after any source file changes (but see also the command-line interface in `main`)

`tornado.autoreload.watch(filename)`

Add a file to the watch list.

All imported modules are watched by default.

`tornado.autoreload.add_reload_hook(fn)`

Add a function to be called before reloading the process.

Note that for open file and socket handles it is generally preferable to set the `FD_CLOEXEC` flag (using `fcntl` or `tornado.platform.auto.set_close_exec`) instead of using a reload hook to close them.

`tornado.autoreload.main()`

Command-line wrapper to re-run a script whenever its source changes.

Scripts may be specified by filename or module name:

```
python -m tornado.autoreload -m tornado.test.runtests
python -m tornado.autoreload tornado/test/runtests.py
```

Running a script with this wrapper is similar to calling `tornado.autoreload.wait` at the end of the script, but this wrapper can catch import-time problems like syntax errors that would otherwise prevent the script from reaching its call to `wait`.

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):
    @asynchronous
    @gen.engine
    def get(self):
        http_client = AsyncHTTPClient()
        response = yield gen.Task(http_client.fetch, "http://example.com")
        do_something_with_response(response)
        self.render("template.html")
```

`Task` works with any function that takes a `callback` keyword argument. You can also yield a list of `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:

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

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

```
class GenAsyncHandler2(RequestHandler):
    @asynchronous
    @gen.engine
    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`).

Decorator

`tornado.gen.engine` (*func*)
Decorator for asynchronous generators.

Any generator that yields objects from this module must be wrapped in this decorator. The decorator only works on functions that are already asynchronous. For `RequestHandler` `get/post/etc` methods, this means that both the `tornado.web.asynchronous` and `tornado.gen.engine` decorators must be used (for proper exception handling, `asynchronous` should come before `gen.engine`). In most other cases, it means that it doesn't make sense to use `gen.engine` on functions that don't already take a callback argument.

Yield points

Instances of the following classes may be used in yield expressions in the generator.

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 `Waits`. The `key` must be unique among outstanding callbacks within a single run of the generator function, but may be reused across different runs of the same function (so constants generally work fine).

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

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.

Other classes

class `tornado.gen.Arguments`
The result of a yield expression whose callback had more than one argument (or keyword arguments).
The `Arguments` object can be used as a tuple (`args`, `kwargs`) or an object with attributes `args` and `kwargs`.

`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"})
>>> 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.iteritems())
[('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 an HTTP file. For backwards compatibility, its instance attributes are also accessible as dictionary keys.

Variables

- **filename** –
- **body** –
- **content_type** – The content_type comes from the provided HTTP header and should not be trusted outright given that it can be easily forged.

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.options — Command-line parsing

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

Each module defines its own options, 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. Your `main()` method can parse the command line or parse a config file with:

```
import tornado.options
tornado.options.parse_config_file("/etc/server.conf")
tornado.options.parse_command_line()
```

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.define(name, default=None, type=None, help=None, metavar=None, multiple=False, group=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 defined file.

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`.

```
tornado.options.options
```

Global options dictionary. Supports both attribute-style and dict-style access.

```
tornado.options.parse_command_line(args=None)
```

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.

```
tornado.options.parse_config_file(path)
```

Parses and loads the Python config file at the given path.

```
tornado.options.print_help(file=sys.stdout)
```

Prints all the command line options to stdout.

```
tornado.options.enable_pretty_logging()
```

Turns on formatted logging output as configured.

This is called automatically by `parse_command_line`.

exception `tornado.options.Error`

Exception raised by errors in the options module.

tornado.process — Utilities for multiple processes

Utilities for working with multiple processes.

`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`.

tornado.stack_context — Exception handling across asynchronous callbacks

`StackContext` allows applications to maintain threadlocal-like state that follows execution as it moves to other execution contexts.

The motivating examples are to eliminate the need for explicit `async_callback` wrappers (as in `tornado.web.RequestHandler`), and to allow some additional context to be kept for logging.

This is slightly magic, but it's an extension of the idea that an exception handler is a kind of stack-local state and when that stack is suspended and resumed in a new context that state needs to be preserved. `StackContext` shifts the burden of restoring that state from each call site (e.g. wrapping each `AsyncHTTPClient` callback in `async_callback`) to the mechanisms that transfer control from one context to another (e.g. `AsyncHTTPClient` itself, `IOLoop`, thread pools, etc).

Example usage:

```
@contextlib.contextmanager
def die_on_error():
    try:
        yield
    except Exception:
        logging.error("exception in asynchronous operation", exc_info=True)
        sys.exit(1)

with StackContext(die_on_error):
    # Any exception thrown here *or* in callback and its descendants*
    # will cause the process to exit instead of spinning endlessly
```

```
# in the ioloop.
http_client.fetch(url, callback)
ioloop.start()
```

Most applications shouldn't have to work with `StackContext` directly. Here are a few rules of thumb for when it's necessary:

- If you're writing an asynchronous library that doesn't rely on a `stack_context`-aware library like `tornado.ioloop` or `tornado.iostream` (for example, if you're writing a thread pool), use `stack_context.wrap()` before any asynchronous operations to capture the stack context from where the operation was started.
- If you're writing an asynchronous library that has some shared resources (such as a connection pool), create those shared resources within a `with stack_context.NullContext(): block`. This will prevent `StackContexts` from leaking from one request to another.
- If you want to write something like an exception handler that will persist across asynchronous calls, create a new `StackContext` (or `ExceptionStackContext`), and make your asynchronous calls in a `with` block that references your `StackContext`.

class `tornado.stack_context.StackContext` (*context_factory*, *_active_cell=None*)
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*, *_active_cell=None*)
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.testing — Unit testing support for asynchronous code

Support classes for automated testing.

This module contains three parts:

- `AsyncTestCase/AsyncHTTPTestCase`: Subclasses of `unittest.TestCase` with additional support for testing asynchronous (IOLoop-based) code.
- `LogTrapTestCase`: Subclass of `unittest.TestCase` that discards log output from tests that pass and only produces output for failing tests.
- `main()`: A simple test runner (wrapper around `unittest.main()`) with support for the `tornado.autoreload` module to rerun the tests when code changes.

These components may be used together or independently. In particular, it is safe to combine `AsyncTestCase` and `LogTrapTestCase` via multiple inheritance. See the docstrings for each class/function below for more information.

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 method provides the `stop()` and `wait()` methods for this purpose. 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` `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 an asynchronous style similar to most async
# application code.
class MyTestCase(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()

# This test uses the argument passing between self.stop and self.wait
# for a simpler, more synchronous style.
# This style is recommended over the preceding example because it
# keeps the assertions in the test method itself, and is therefore
# less sensitive to the subtleties of stack_context.
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)

```

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).

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=5*)

Runs the IOLoop until stop is called or timeout has passed.

In the event of a timeout, an exception will be thrown.

If condition is not None, the IOLoop will be restarted after stop() until condition() returns true.

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

```

get_app()

Should be overridden by subclasses to return a tornado.web.Application or other HTTPServer callback.

fetch (*path*, ***kwargs*)

Convenience method to synchronously fetch a url.

The given path will be appended to the local server's host and port. Any additional kwargs will be passed directly to AsyncHTTPClient.fetch (and so could be used to pass method="POST", body="...", etc).

get_httpserver_options()

May be overridden by subclasses to return additional keyword arguments for the server.

get_http_port()

Returns the port used by the server.

A new port is chosen for each test.

get_url (*path*)

Returns an absolute url for the given path on the test server.

class `tornado.testing.AsyncHTTPSTestCase` (**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.

Controlling log output

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`.

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.get_unused_port` ()
Returns a (hopefully) unused port number.

5.1.6 Release notes

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 performance of `SimpleAsyncHTTPClient` when downloading files with lots of chunks.
- `SSLIOStream` now works correctly when `ssl_version` is set to a value other than `SSLv23`.
- Idle `IOLoops` no longer wake up several times a second.
- `tornado.ioloop.PeriodicCallback` no longer triggers duplicate callbacks when stopped and started repeatedly.

`tornado.template`

- Exceptions in template code will now show better stack traces that reference lines from the original template file.
- `{#` and `#}` can now be used for comments (and unlike the old `{% comment %}` directive, these can wrap other template directives).
- Template directives may now span multiple lines.

`tornado.web`

- Now behaves better when given malformed `Cookie` headers
- `RequestHandler.redirect` now has a `status` argument to send status codes other than 301 and 302.
- New method `RequestHandler.on_finish` may be overridden for post-request processing (as a counterpart to `RequestHandler.prepare`)
- `StaticFileHandler` now outputs `Content-Length` and `Etag` headers on HEAD requests.
- `StaticFileHandler` now has overridable `get_version` and `parse_url_path` methods for use in subclasses.
- `RequestHandler.static_url` now takes an `include_host` parameter (in addition to the old support for the `RequestHandler.include_host` attribute).

`tornado.websocket`

- Updated to support the latest version of the protocol, as finalized in RFC 6455.
- Many bugs were fixed in all supported protocol versions.
- `tornado.websocket` no longer supports the older “draft 76” version of the websocket protocol by default, although this version can be enabled by overriding `tornado.websocket.WebSocketHandler.allow_draft76`.
- `WebSocketHandler.write_message` now accepts a `binary` argument to send binary messages.
- Subprotocols (i.e. the `Sec-WebSocket-Protocol` header) are now supported; see the `WebSocketHandler.select_subprotocol` method for details.
- `WebSocketHandler.get_websocket_scheme` can be used to select the appropriate url scheme (`ws://` or `wss://`) in cases where `HTTPRequest.protocol` is not set correctly.

Other modules

- `tornado.auth.TwitterMixin.authenticate_redirect` now takes a `callback_uri` parameter.
- `tornado.auth.TwitterMixin.twitter_request` now accepts both URLs and partial paths (complete URLs are useful for the search API which follows different patterns).
- Exception handling in `tornado.gen` has been improved. It is now possible to catch exceptions thrown by a `Task`.
- `tornado.netutil.bind_sockets` now works when `getaddrinfo` returns duplicate addresses.
- `tornado.platform.twisted` compatibility has been significantly improved. Twisted version 11.1.0 is now supported in addition to 11.0.0.
- `tornado.process.fork_processes` correctly reseeds the `random` module even when `os.urandom` is not implemented.
- `tornado.testing.main` supports a new flag `--exception_on_interrupt`, which can be set to `false` to make `Ctrl-C` kill the process more reliably (at the expense of stack traces when it does so).
- `tornado.version_info` is now a four-tuple so official releases can be distinguished from development branches.

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. `xsrif_form_html` and `linkify`, which must now be called with `{% raw %}` instead of `{}`
- * Applications that wish to continue using `curl_httpclient` instead of `simple_httpclient` may do so by calling `AsyncHTTPClient.configure("tornado.curl_httpclient.CurlAsyncHTTPClient")` at the beginning of the process. Users of Python 2.5 will probably want to use `curl_httpclient` as `simple_httpclient` only supports ssl on Python 2.6+.
- * Python 3 compatibility involved many changes throughout the codebase, so users are encouraged to test their applications more thoroughly than usual when upgrading to this release.

Other changes in this release:

- * Templates support several new directives:
 - `{% autoescape ...%}` to control escaping behavior
 - `{% raw ... %}` for unescaped output
 - `{% module ... %}` for calling UIModules
- * `{% module Template(path, **kwargs) %}` may now be used to call another template with an independent namespace
- * All IOStream callbacks are now run directly on the IOLoop via `add_callback`.
- * 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 in `AsyncHTTPTestCase`
- * `AsyncHTTPTestCase.fetch()`: Convenience method for synchronous fetches
- * `IOLoop.set_blocking_signal_threshold()`: Set a callback to be run when the `IOLoop` is blocked.
- * `IOStream.connect()`: Asynchronously connect a client socket
- * `AsyncHTTPClient.handle_callback_exception()`: May be overridden in subclass for custom error handling
- * `httpclient.HTTPRequest` has two new keyword arguments, `validate_cert` and `ca_certs`. Setting `validate_cert=False` will disable all certificate checks when fetching https urls. `ca_certs` may be set to a filename containing trusted certificate authorities (defaults will be used if this is unspecified)
- * `HTTPRequest.get_ssl_certificate()`: Returns the client's SSL certificate (if client certificates were requested in the server's `ssl_options`)
- * `StaticFileHandler` can be configured to return a default file (e.g. `index.html`) when a directory is requested
- * Template directives of the form `"{% from x import y %}"` are now supported (in addition to the existing support for `"{% import x %}"`)
- * `FacebookGraphMixin.get_authenticated_user` now accepts a new parameter `'extra_fields'` which may be used to request additional information about the user

Bug fixes:

- * `auth`: Fixed `KeyError` with Facebook `offline_access`
- * `auth`: Uses `request.uri` instead of `request.path` as the default redirect so that parameters are preserved.
- * `escape`: `xhtml_escape()` now returns a unicode string, not utf8-encoded bytes
- * `ioloop`: Callbacks added with `add_callback` are now run in the order they were added
- * `ioloop`: `PeriodicCallback.stop` can now be called from inside the callback.
- * `iostream`: Fixed several bugs in `SSLIOStream`
- * `iostream`: Detect when the other side has closed the connection even with the `select()`-based `IOLoop`
- * `iostream`: `read_bytes(0)` now works as expected
- * `iostream`: Fixed bug when writing large amounts of data on windows
- * `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

5.2 Indices and tables

- *genindex*
- *modindex*
- *search*

t

tornado.auth, 58
tornado.autoreload, 70
tornado.database, 65
tornado.escape, 43
tornado.gen, 71
tornado.httpclient, 52
tornado.httpserver, 37
tornado.httputil, 73
tornado.ioloop, 47
tornado.iostream, 50
tornado.locale, 45
tornado.netutil, 56
tornado.options, 74
tornado.platform.twisted, 66
tornado.process, 76
tornado.stack_context, 76
tornado.template, 40
tornado.testing, 78
tornado.web, 26
tornado.websocket, 67
tornado.wsgi, 69

A

add() (tornado.httputil.HTTPHeaders method), 73
 add_accept_handler() (in module tornado.netutil), 58
 add_callback() (tornado.ioloop.IOLoop method), 49
 add_handler() (tornado.ioloop.IOLoop method), 49
 add_handlers() (tornado.web.Application method), 34
 add_header() (tornado.web.RequestHandler method), 28
 add_reload_hook() (in module tornado.autoreload), 71
 add_socket() (tornado.netutil.TCPServer method), 57
 add_sockets() (tornado.netutil.TCPServer method), 57
 add_timeout() (tornado.ioloop.IOLoop method), 49
 add_transform() (tornado.web.Application method), 34
 addslash() (in module tornado.web), 35
 allow_draft76() (tornado.websocket.WebSocketHandler method), 68
 Application (class in tornado.web), 32
 application (tornado.web.RequestHandler attribute), 30
 Arguments (class in tornado.gen), 73
 arguments (tornado.httpserver.HTTPRequest attribute), 38
 async_callback() (tornado.web.RequestHandler method), 30
 async_callback() (tornado.websocket.WebSocketHandler method), 68
 asynchronous() (in module tornado.web), 34
 AsyncHTTPClient (class in tornado.httpclient), 52
 AsyncHTTPSTestCase (class in tornado.testing), 80
 AsyncHTTPSTestCase (class in tornado.testing), 79
 AsyncTestCase (class in tornado.testing), 78
 authenticate_redirect() (tornado.auth.FacebookMixin method), 63
 authenticate_redirect() (tornado.auth.OpenIdMixin method), 59
 authenticate_redirect() (tornado.auth.TwitterMixin method), 60
 authenticated() (in module tornado.web), 35
 authorize_redirect() (tornado.auth.FacebookMixin method), 63
 authorize_redirect() (tornado.auth.GoogleMixin method), 62

authorize_redirect() (tornado.auth.OAuth2Mixin method), 59
 authorize_redirect() (tornado.auth.OAuthMixin method), 59

B

BaseLoader (class in tornado.template), 42
 bind() (tornado.netutil.TCPServer method), 57
 bind_sockets() (in module tornado.netutil), 57
 bind_unix_socket() (in module tornado.netutil), 58
 body (tornado.httpserver.HTTPRequest attribute), 37

C

Callback (class in tornado.gen), 72
 check_xsrf_cookie() (tornado.web.RequestHandler method), 30
 clear() (tornado.web.RequestHandler method), 29
 clear_all_cookies() (tornado.web.RequestHandler method), 30
 clear_cookie() (tornado.web.RequestHandler method), 30
 clear_header() (tornado.web.RequestHandler method), 28
 close() (tornado.database.Connection method), 65
 close() (tornado.httpclient.AsyncHTTPClient method), 53
 close() (tornado.httpclient.HTTPClient method), 52
 close() (tornado.ioloop.IOLoop method), 48
 close() (tornado.iostream.IOStream method), 51
 close() (tornado.websocket.WebSocketHandler method), 68
 closed() (tornado.iostream.IOStream method), 51
 compute_etag() (tornado.web.RequestHandler method), 31
 configure() (tornado.httpclient.AsyncHTTPClient static method), 53
 connect() (tornado.iostream.IOStream method), 50
 Connection (class in tornado.database), 65
 connection (tornado.httpserver.HTTPRequest attribute), 38
 cookies (tornado.httpserver.HTTPRequest attribute), 38
 cookies (tornado.web.RequestHandler attribute), 30
 cookies (tornado.wsgi.HTTPRequest attribute), 69

cpu_count() (in module tornado.process), 76
 create_signed_value() (tornado.web.RequestHandler method), 30
 create_template_loader() (tornado.web.RequestHandler method), 31
 css_files() (tornado.web.UIModule method), 35
 CSVLocale (class in tornado.locale), 46

D

decode_argument() (tornado.web.RequestHandler method), 28
 define() (in module tornado.options), 75
 delete() (tornado.web.RequestHandler method), 27
 DictLoader (class in tornado.template), 43

E

embedded_css() (tornado.web.UIModule method), 35
 embedded_javascript() (tornado.web.UIModule method), 35
 enable_pretty_logging() (in module tornado.options), 75
 engine() (in module tornado.gen), 72
 environ() (tornado.wsgi.WSGIContainer static method), 70
 Error, 75
 ErrorHandler (class in tornado.web), 35
 ExceptionStackContext (class in tornado.stack_context), 77
 execute() (tornado.database.Connection method), 65
 execute_lastrowid() (tornado.database.Connection method), 66
 execute_rowcount() (tornado.database.Connection method), 66
 executemany() (tornado.database.Connection method), 66
 executemany_lastrowid() (tornado.database.Connection method), 66
 executemany_rowcount() (tornado.database.Connection method), 66

F

facebook_request() (tornado.auth.FacebookGraphMixin method), 64
 facebook_request() (tornado.auth.FacebookMixin method), 63
 FacebookGraphMixin (class in tornado.auth), 64
 FacebookMixin (class in tornado.auth), 62
 FallbackHandler (class in tornado.web), 35
 fetch() (tornado.httpclient.AsyncHTTPClient method), 53
 fetch() (tornado.httpclient.HTTPClient method), 52
 fetch() (tornado.testing.AsyncHTTPTestCase method), 79
 files (tornado.httptester.HTTPRequest attribute), 38
 finish() (tornado.httptester.HTTPConnection method), 40

finish() (tornado.httptester.HTTPRequest method), 38
 finish() (tornado.web.RequestHandler method), 29
 flush() (tornado.web.RequestHandler method), 29
 fork_processes() (in module tornado.process), 76
 format_date() (tornado.locale.Locale method), 46
 format_day() (tornado.locale.Locale method), 46
 friendfeed_request() (tornado.auth.FriendFeedMixin method), 61
 FriendFeedMixin (class in tornado.auth), 61
 friendly_number() (tornado.locale.Locale method), 46
 full_url() (tornado.httptester.HTTPRequest method), 38
 full_url() (tornado.wsgi.HTTPRequest method), 69

G

generate() (tornado.template.Template method), 42
 get() (in module tornado.locale), 45
 get() (tornado.database.Connection method), 65
 get() (tornado.locale.Locale class method), 46
 get() (tornado.web.RequestHandler method), 27
 get_all() (tornado.httputil.HTTPHeaders method), 73
 get_app() (tornado.testing.AsyncHTTPTestCase method), 79
 get_argument() (tornado.web.RequestHandler method), 28
 get_arguments() (tornado.web.RequestHandler method), 28
 get_auth_http_client() (tornado.auth.FacebookGraphMixin method), 65
 get_auth_http_client() (tornado.auth.FacebookMixin method), 64
 get_auth_http_client() (tornado.auth.OAuthMixin method), 59
 get_auth_http_client() (tornado.auth.OpenIdMixin method), 59
 get_authenticated_user() (tornado.auth.FacebookGraphMixin method), 64
 get_authenticated_user() (tornado.auth.FacebookMixin method), 63
 get_authenticated_user() (tornado.auth.GoogleMixin method), 62
 get_authenticated_user() (tornado.auth.OAuthMixin method), 59
 get_authenticated_user() (tornado.auth.OpenIdMixin method), 59
 get_browser_locale() (tornado.web.RequestHandler method), 31
 get_cache_time() (tornado.web.StaticFileHandler method), 36
 get_closest() (tornado.locale.Locale class method), 46
 get_cookie() (tornado.web.RequestHandler method), 30
 get_current_user() (tornado.web.RequestHandler method), 31

[get_http_port\(\)](#) (tornado.testing.AsyncHTTPTestCase method), 79
[get_https_server_options\(\)](#) (tornado.testing.AsyncHTTPTestCase method), 79
[get_list\(\)](#) (tornado.httputil.HTTPHeaders method), 73
[get_login_url\(\)](#) (tornado.web.RequestHandler method), 31
[get_new_ioloop\(\)](#) (tornado.testing.AsyncTestCase method), 79
[get_secure_cookie\(\)](#) (tornado.web.RequestHandler method), 30
[get_ssl_certificate\(\)](#) (tornado.httpserver.HTTPRequest method), 38
[get_ssl_options\(\)](#) (tornado.testing.AsyncHTTPSTestCase method), 80
[get_status\(\)](#) (tornado.web.RequestHandler method), 31
[get_supported_locales\(\)](#) (in module tornado.locale), 46
[get_template_namespace\(\)](#) (tornado.web.RequestHandler method), 29
[get_template_path\(\)](#) (tornado.web.RequestHandler method), 31
[get_unused_port\(\)](#) (in module tornado.testing), 80
[get_url\(\)](#) (tornado.testing.AsyncHTTPTestCase method), 79
[get_user_locale\(\)](#) (tornado.web.RequestHandler method), 31
[get_version\(\)](#) (tornado.web.StaticFileHandler class method), 36
[get_websocket_scheme\(\)](#) (tornado.websocket.WebSocketHandler method), 68
[GettextLocale](#) (class in tornado.locale), 47
[GoogleMixin](#) (class in tornado.auth), 62

H

[handle_callback_exception\(\)](#) (tornado.ioloop.IOLoop method), 49
[handle_stream\(\)](#) (tornado.netutil.TCPServer method), 57
[head\(\)](#) (tornado.web.RequestHandler method), 27
[headers](#) (tornado.httpserver.HTTPRequest attribute), 37
[host](#) (tornado.httpserver.HTTPRequest attribute), 37
[html_body\(\)](#) (tornado.web.UIModule method), 35
[html_head\(\)](#) (tornado.web.UIModule method), 35
[HTTPClient](#) (class in tornado.httpclient), 52
[HTTPConnection](#) (class in tornado.httpserver), 40
[HTTPError](#), 35, 55
[HTTPFile](#) (class in tornado.httputil), 74
[HTTPHeaders](#) (class in tornado.httputil), 73
[HTTPRequest](#) (class in tornado.httpclient), 54
[HTTPRequest](#) (class in tornado.httpserver), 37
[HTTPRequest](#) (class in tornado.wsgi), 69
[HTTPResponse](#) (class in tornado.httpclient), 55
[HTTPServer](#) (class in tornado.httpserver), 38

I

[initialize\(\)](#) (tornado.web.RequestHandler method), 27
[initialized\(\)](#) (tornado.ioloop.IOLoop static method), 48
[install\(\)](#) (in module tornado.platform.twisted), 66
[install\(\)](#) (tornado.ioloop.IOLoop method), 48
[instance\(\)](#) (tornado.ioloop.IOLoop static method), 48
[IOLoop](#) (class in tornado.ioloop), 47
[IOStream](#) (class in tornado.iostream), 50
[iter\(\)](#) (tornado.database.Connection method), 65

J

[javascript_files\(\)](#) (tornado.web.UIModule method), 35
[json_decode\(\)](#) (in module tornado.escape), 43
[json_encode\(\)](#) (in module tornado.escape), 43

L

[linkify\(\)](#) (in module tornado.escape), 44
[list\(\)](#) (tornado.locale.Locale method), 46
[listen\(\)](#) (tornado.netutil.TCPServer method), 57
[listen\(\)](#) (tornado.web.Application method), 33
[load\(\)](#) (tornado.template.BaseLoader method), 43
[load_gettext_translations\(\)](#) (in module tornado.locale), 46
[load_translations\(\)](#) (in module tornado.locale), 45
[Loader](#) (class in tornado.template), 43
[Locale](#) (class in tornado.locale), 46
[log_request\(\)](#) (tornado.web.Application method), 34
[log_stack\(\)](#) (tornado.ioloop.IOLoop method), 50
[LogTrapTestCase](#) (class in tornado.testing), 80

M

[main\(\)](#) (in module tornado.autoreload), 71
[main\(\)](#) (in module tornado.testing), 80
[make_static_url\(\)](#) (tornado.web.StaticFileHandler class method), 36
[method](#) (tornado.httpserver.HTTPRequest attribute), 37

N

[native_str\(\)](#) (in module tornado.escape), 44
[NullContext](#) (class in tornado.stack_context), 77

O

[OAuth2Mixin](#) (class in tornado.auth), 59
[OAuthMixin](#) (class in tornado.auth), 59
[on_close\(\)](#) (tornado.websocket.WebSocketHandler method), 68
[on_connection_close\(\)](#) (tornado.web.RequestHandler method), 31
[on_finish\(\)](#) (tornado.web.RequestHandler method), 27
[on_message\(\)](#) (tornado.websocket.WebSocketHandler method), 67
[open\(\)](#) (tornado.websocket.WebSocketHandler method), 67
[OpenIdMixin](#) (class in tornado.auth), 58

options (in module tornado.options), 75
options() (tornado.web.RequestHandler method), 27

P

parse() (tornado.httputil.HTTPHeaders class method), 74
parse_body_arguments() (in module tornado.httputil), 74
parse_command_line() (in module tornado.options), 75
parse_config_file() (in module tornado.options), 75
parse_line() (tornado.httputil.HTTPHeaders method), 73
parse_multipart_form_data() (in module tornado.httputil), 74
parse_url_path() (tornado.web.StaticFileHandler method), 37
ParseError, 43
path (tornado.httpserver.HTTPRequest attribute), 37
PeriodicCallback (class in tornado.ioloop), 49
post() (tornado.web.RequestHandler method), 27
prepare() (tornado.web.RequestHandler method), 27
print_help() (in module tornado.options), 75
protocol (tornado.httpserver.HTTPRequest attribute), 37
put() (tornado.web.RequestHandler method), 27

Q

query (tornado.httpserver.HTTPRequest attribute), 37
query() (tornado.database.Connection method), 65

R

read_bytes() (tornado.iostream.IOStream method), 51
read_until() (tornado.iostream.IOStream method), 51
read_until_close() (tornado.iostream.IOStream method), 51
read_until_regex() (tornado.iostream.IOStream method), 51
reading() (tornado.iostream.IOStream method), 51
reconnect() (tornado.database.Connection method), 65
recursive_unicode() (in module tornado.escape), 44
redirect() (tornado.web.RequestHandler method), 29
RedirectHandler (class in tornado.web), 36
remote_ip (tornado.httpserver.HTTPRequest attribute), 37
remove_handler() (tornado.ioloop.IOLoop method), 49
remove_timeout() (tornado.ioloop.IOLoop method), 49
removeslash() (in module tornado.web), 35
render() (tornado.web.RequestHandler method), 29
render() (tornado.web.UIModule method), 35
render_string() (tornado.web.RequestHandler method), 29
render_string() (tornado.web.UIModule method), 35
request (tornado.web.RequestHandler attribute), 28
request_time() (tornado.httpserver.HTTPRequest method), 38
request_time() (tornado.wsgi.HTTPRequest method), 69
RequestHandler (class in tornado.web), 27

require_setting() (tornado.web.RequestHandler method), 31
reset() (tornado.template.BaseLoader method), 43
resolve_path() (tornado.template.BaseLoader method), 43
rethrow() (tornado.httplibclient.HTTPResponse method), 55
reverse_url() (tornado.web.Application method), 34
reverse_url() (tornado.web.RequestHandler method), 31
Row (class in tornado.database), 66
running() (tornado.ioloop.IOLoop method), 48

S

select_subprotocol() (tornado.websocket.WebSocketHandler method), 68
send_error() (tornado.web.RequestHandler method), 29
set_blocking_log_threshold() (tornado.ioloop.IOLoop method), 49
set_blocking_signal_threshold() (tornado.ioloop.IOLoop method), 49
set_close_callback() (tornado.iostream.IOStream method), 51
set_cookie() (tornado.web.RequestHandler method), 30
set_default_headers() (tornado.web.RequestHandler method), 28
set_default_locale() (in module tornado.locale), 45
set_extra_headers() (tornado.web.StaticFileHandler method), 36
set_header() (tornado.web.RequestHandler method), 28
set_secure_cookie() (tornado.web.RequestHandler method), 30
set_status() (tornado.web.RequestHandler method), 28
settings (tornado.web.Application attribute), 32
settings (tornado.web.RequestHandler attribute), 31
squeeze() (in module tornado.escape), 45
SSLIOStream (class in tornado.iostream), 51
StackContext (class in tornado.stack_context), 77
start() (in module tornado.autoreload), 70
start() (tornado.ioloop.IOLoop method), 48
start() (tornado.ioloop.PeriodicCallback method), 49
start() (tornado.netutil.TCPServer method), 57
static_url() (tornado.web.RequestHandler method), 31
StaticFileHandler (class in tornado.web), 36
stop() (tornado.ioloop.IOLoop method), 48
stop() (tornado.ioloop.PeriodicCallback method), 49
stop() (tornado.netutil.TCPServer method), 57
stop() (tornado.testing.AsyncTestCase method), 79
supports_http_1_1() (tornado.httpserver.HTTPRequest method), 38
supports_http_1_1() (tornado.wsgi.HTTPRequest method), 69

T

Task (class in tornado.gen), 72

task_id() (in module tornado.process), 76
 TCPServer (class in tornado.netutil), 56
 Template (class in tornado.template), 42
 to_basestring() (in module tornado.escape), 44
 to_unicode() (in module tornado.escape), 44
 tornado.auth (module), 58
 tornado.autoreload (module), 70
 tornado.database (module), 65
 tornado.escape (module), 43
 tornado.gen (module), 71
 tornado.httpclient (module), 52
 tornado.httpserver (module), 37
 tornado.httputil (module), 73
 tornado.ioloop (module), 47
 tornado.iostream (module), 50
 tornado.locale (module), 45
 tornado.netutil (module), 56
 tornado.options (module), 74
 tornado.platform.twisted (module), 66
 tornado.process (module), 76
 tornado.stack_context (module), 76
 tornado.template (module), 40
 tornado.testing (module), 78
 tornado.web (module), 26
 tornado.websocket (module), 67
 tornado.wsgi (module), 69
 TornadoReactor (class in tornado.platform.twisted), 66
 translate() (tornado.locale.Locale method), 46
 twitter_request() (tornado.auth.TwitterMixin method), 60
 TwitterMixin (class in tornado.auth), 60

U

UIModule (class in tornado.web), 35
 update_handler() (tornado.ioloop.IOLoop method), 49
 uri (tornado.httpserver.HTTPRequest attribute), 37
 url_concat() (in module tornado.httputil), 74
 url_escape() (in module tornado.escape), 43
 url_unescape() (in module tornado.escape), 43
 URLSpec (class in tornado.web), 34
 utf8() (in module tornado.escape), 44

V

version (tornado.httpserver.HTTPRequest attribute), 37

W

Wait (class in tornado.gen), 72
 wait() (in module tornado.autoreload), 70
 wait() (tornado.testing.AsyncTestCase method), 79
 WaitAll (class in tornado.gen), 73
 watch() (in module tornado.autoreload), 70
 WebSocketHandler (class in tornado.websocket), 67
 wrap() (in module tornado.stack_context), 77
 write() (tornado.httpserver.HTTPConnection method), 40
 write() (tornado.httpserver.HTTPRequest method), 38

write() (tornado.iostream.IOStream method), 51
 write() (tornado.web.RequestHandler method), 28
 write_error() (tornado.web.RequestHandler method), 29
 write_message() (tornado.websocket.WebSocketHandler method), 68
 writing() (tornado.iostream.IOStream method), 51
 WSGIApplication (class in tornado.wsgi), 69
 WSGIContainer (class in tornado.wsgi), 70

X

xhtml_escape() (in module tornado.escape), 43
 xhtml_unescape() (in module tornado.escape), 43
 xsrf_form_html() (tornado.web.RequestHandler method), 32