TornadIO2 Documentation

Release 0.1

Serge S. Koval

January 08, 2016

Contents

1 Topics **3**
- 1.1 Migrating from previous TornadIO version . . . 3
- 1.2 Unicode . . . 4
- 1.3 Multiplexed connections . . . 4
- 1.4 Events . . . 5
- 1.5 Acknowledgments . . . 6
- 1.6 Generator-based asynchronous interface . . . 7
- 1.7 Statistics . . . 8
- 1.8 Deployment . . . 8
- 1.9 Bugs and Workarounds . . . 9
- 1.10 API . . . 10

2 Indices and tables **23**

Python Module Index **25**

This is implementation of the Socket.IO realtime transport library on top of the Tornado framework.

TornadIO2 is compatible with 0.7+ version of the Socket.IO and implements most of the features found in original Socket.IO server software.

CHAPTER 1

Topics

1.1 Migrating from previous TornadIO version

TornadIO2 has some incompatible API changes.

1. Instead of having one router handler, TornadIO2 exposes transports as first-class Tornado handlers. This saves some memory per active connection, because instead of having two handlers per request (router and transport), you will now have only one. This change affects how TornadIO2 is initialized and plugged into your Tornado application:

```
ChatServer = tornadio2.router.TornadioRouter(ChatConnection)
# Fill your routes here
routes = [(r"/", IndexHandler)]
# Extend list of routes with Tornadio2 URLs
routes.extend(ChatServer.urls)

application = tornado.web.Application(routes)
```

or alternative approach:

```
ChatServer = tornadio2.router.TornadioRouter(ChatConnection)
application = tornado.web.Application(ChatServer.apply_routes([(r"/", IndexHandler)]))
```

2. *SocketConnection.on_open* was changed to accept single `request` parameter. This parameter is instance of the ConnectionInfo class which contains some helper methods like get_argument(), get_cookie(), etc. Also, if you return `False` from your `on_open` handler, TornadIO2 will reject connection.

Example:

```
class MyConnection(SocketConnection):
    def on_open(self, request):
        self.user_id = request.get_argument('id')

        if not self.user_id:
            return False
```

This variable is also available for multiplexed connections and will contain query string parameters from the socket.io endpoint connection request.

3. There's major behavioral change in exception handling. If something blows up and is not handled, whole connection is closed (including any running multiplexed connections). In previous TornadIO version it was silently dropping currently open transport connection and expecting for socket.io to reconnect.

4. Persistent connections are not dropped immediately - there's a chance that person might reconnect with same session id and we will want to pick it up.

5. Socket.IO 0.7 dropped support for xhr-multipart transport, so you can safely remove it from your configuration file.

1.2 Unicode

TornadIO2 supports unicode for all transports. When you send something, it will be automatically converted to the unicode (assuming that it is not unicode already).

Few rules:

1. `send` has following logic in place:
   - If message is object or dictionary, it will be json encoded into unicode string
   - If message is unicode, it will be sent as is
   - If message is non-unicode, not an object and not a dictionary, it will be converted to string and converted to unicode using utf-8 encoding. If your string is in some other encoding (multi-byte, etc), it is up for you to handle encoding and pass your data in unicode (or utf-8 encoded).
2. `emit` has similar logic:
   - It will convert event name into unicode string (if it is not). It is expected that event name will only use latin characters
   - All `emit` arguments will be json encoded into unicode string

3. All incoming messages will be automatically converted to unicode strings. You can expect to receive unicode strings in your message handler and events.

1.3 Multiplexed connections

Starting from socket.io 0.7, there's new concept of multiplexed connections: you can have multiple "virtual" connections working through one transport connection. TornadIO2 supports this transparently, but you have to tell TornadIO how to route multiplexed connection requests. To accomplish this, you can either use built-in routing mechanism or implement your own.

To use built-in routing, declare and initialize `__endpoints__` dictionary in your main connection class:

```
class ChatConnection(SocketConnection):
    def on_message(self, msg):
        pass

class PingConnection(SocketConnection):
    def on_message(self, msg):
        pass

class MyRouterConnection(SocketConnection):
    __endpoints__ = {'/chat': ChatConnection,
                     '/ping': PingConnection}

    def on_message(self, msg):
        pass

MyRouter = tornadio2.router.TornadioRouter(MyRouterConnection)
```

On client side, create two connections:

```
var chat = io.connect('http://myserver/chat'),
    ping = io.connect('http://myserver/ping');

chat.send('Hey Chat');
ping.send('Hey Ping');
```

So, you have to have three connection classes for 2 virtual connections - that's how socket.io works. If you want, you can send some messages to MyRouterConnection as well, if you will connect like this:

```
var conn = io.connect('http://myserver'),
    chat = io.connect('http://myserver/chat'),
    ping = io.connect('http://myserver/ping');

    conn.send('Hey Connection!')
```

1.4 Events

Instead of having "just" messages, socket.io 0.7 introduced new concept of events. Event is just a name and collection of parameters.

TornadIO2 provides easy-to-use syntax sugar which emulates RPC calls from the client to your python code. Check following example:

```
class MyConnection(SocketConnection):
    @event('hello')
    def test(self, name):
        print 'Hello %s' % name

        self.emit('thanks', name=name)
        self.emit('hello', name, 'foobar')
```

In your client code, to call this event, do something like:

```
sock.emit('hello', {name: 'Joes'});
```

You can also use positional parameters. For previous example, you can also do something like:

```
sock.emit('hello', 'Joes')
```

To handle event on client side, use following code:

```
sock.on('thanks', function(data) {
    alert(data.name);
});
sock.on('hello', function(name, dummy) {
    alert('Hey ' + name + ' ' + dummy);
});
```

However, take care - if method signature does not match (missing parameters, extra parameters, etc), your connection will blow up and self destruct.

`event` decorator can be used without parameter and it will use event handler name in this case:

```
class MyConnection(SocketConnection):
    @event
    def hello(self, name):
        print 'Hello %s' % name
```

If you don't like this event handling approach, just override `on_event` in your socket connection class and handle them by yourself:

```
class MyConnection(SocketConnection):
    def on_event(self, name, *args, **kwargs):
        if name == 'hello':
            print 'Hello %s' % (kwargs['name'])

        self.emit('thanks', name=kwargs['name'])
```

There's also some magic involved in event message parsing to make it easier to work with events.

If you send data from client using following code:

```
sock.emit('test', {a: 10, b: 10});
```

TornadIO2 will unpack dictionary into `kwargs` parameters and pass it to the `on_event` handler. However, if you pass more than one parameter, Tornadio2 won't unpack them into `kwargs` and will just pass parameters as `args`. For example, this code will lead to `args` being passed to `on_event` handler:

```
sock.emit('test', 1, 2, 3, {a: 10, b: 10});
```

1.5 Acknowledgments

New feature of the socket.io 0.7+. When you send message to the client, you now have way to get notified when client receives the message. To use this, pass a callback function when sending a message:

```
class MyConnection(SocketConnection):
    def on_message(self, msg):
        self.send(msg, self.my_callback)

    def my_callback(self, msg, ack_data):
        print 'Got ack for my message: %s' % message
```

`ack_data` contains acknowledgment data sent from the client, if any.

To send event with acknowledgement, use `SocketConnection.emit_ack` method:

```
class MyConnection(SocketConnection):
    def on_message(self, msg):
        self.emit_ack(self.my_callback, 'hello')

    def my_callback(self, event, ack_data):
        print 'Got ack for my message: %s' % msg
```

If you want to send reverse confirmation with a message, just return value you want to send from your event handler:

```
class MyConnection(SocketConnection):
    @event('test')
    def test(self):
        return 'Joes'
```

and then, in your javascript code, you can do something like:

```
sock.emit('test', function(data) {
    console.log(data);  // data will be 'Joes'
});
```

If you want to return multiple arguments, return them as tuple:

```
class MyConnection(SocketConnection):
    @event('test')
    def test(self):
        return 'Joes', 'Mike', 'Mary'
```

On client-side, you can access them by doing something like:

```
sock.emit('test', function(name1, name2, name3) {
    console.log(name1, name2, name3);  // data will be 'Joes'
});
```

1.6 Generator-based asynchronous interface

`tornadio2.gen` module is a wrapper around `tornado.gen` API. You might want to take a look at Tornado documentation for this module, which can be found here.

While you can use `tornado.gen` API without any problems, sometimes you may want to handle your messages in order they were received. If you will decorate your functions with `tornado.gen.engine`, your code will work asynchronously - second message might get handled before first message got processed.

To prevent this situation, TornadIO2 provides helpful decorator: `tornadio2.gen.sync_engine`. `sync_engine` will queue incoming calls if there's another instance of the function running. As a result, it will call your method synchronously without blocking the io_loop. This decorator only works with class methods, don't try to use it for functions - it requires `self` to properly function.

Lets check following example:

```
from tornadio2 import gen

class MyConnection(SocketConnection):
    @gen.sync_engine
    def on_message(self, query):
        http_client = AsyncHTTPClient()
        response = yield gen.Task(http_client.fetch, 'http://google.com?q=' + query)
        self.send(response.body)
```

If client will quickly send two messages, it will work "synchronously" - `on_message` won't be called for second message till handling of first message is finished.

However, if you will change decorator to `gen.engine`, message handling will be asynchronous and might be out of order:

```
from tornadio2 import gen

class MyConnection(SocketConnection):
    @gen.engine
    def on_message(self, query):
        http_client = AsyncHTTPClient()
        response = yield gen.Task(http_client.fetch, 'http://google.com?q=' + query)
        self.send(response.body)
```

If client will quickly send two messages, server will send response as soon as response is ready and if it takes longer to handle first message, response for second message will be sent first.

As a nice feature, you can also decorate your event handlers or even wrap main `on_event` method, so all events can be synchronous when using asynchronous calls.

`tornadio2.gen` API will only work with the `yield` based methods (methods that produce generators). If you implement your asynchronous code using explicit callbacks, it is up for you how to synchronize their execution order.

TBD: performance considerations, python iterator performance.

1.7 Statistics

TornadIO2 captures some counters:

Name	Description
Sessions	
max_sessions	Maximum number of sessions
active_sessions	Number of currently active sessions
Connections	
max_connections	Maximum number of connections
active_connections	Number of currently active connections
connections_ps	Number of opened connections per second
Packets	
packets_sent_ps	Packets sent per second
packets_recv_ps	Packets received per second

Stats are captured by the router object and can be accessed through the `stats` property:

```
MyRouter = tornadio2.TornadioRouter(MyConnection)

print MyRouter.stats.dump()
```

For more information, check stats module API or `stats` example.

1.8 Deployment

1.8.1 Going big

So, you've finished writting your application and want to share it with rest of the world, so you started thinking about scalability, deployment options, etc.

Most of the Tornado servers are deployed behind the nginx, which also used to serve static content. Unfortunately, older versions of the nginx did not support HTTP 1.1 and as a result, proxying of the websocket connections did not work. However, starting from nginx 1.1 there's support of HTTP 1.1 protocol and websocket proxying works. You can get more information here.

Alternative solution is to use HAProxy. Sample HAProxy configuration file can be found here. You can hide your application and TornadIO instances behind one HAProxy instance running on one port to avoid cross-domain AJAX calls, which ensures greater compatibility.

However, HAProxy does not work on Windows, so if you plan to deploy your solution on Windows platform, you might want to take look into MLB.

1.8.2 Scalability

Scalability is completely different beast. It is up for you, as a developer, to design scalable architecture of the application.

For example, if you need to have one large virtual server out of your multiple physical processes (or even servers), you have to come up with some kind of the synchronization mechanism. This can be either common meeting point (and also point of failure), like memcached, redis, etc. Or you might want to use some transporting mechanism to communicate between servers, for example something AMQP based, ZeroMQ or just plain sockets with your custom protocol.

1.8.3 Performance

Unfortunately, TornadIO2 was not properly benchmarked and this is something that will be accomplished in the future.

1.9 Bugs and Workarounds

There are some known bugs in socket.io (valid for socket.io-client 0.8.6). I consider them "show-stoppers", but you can work around them with some degree of luck.

1.9.1 Connect after disconnect

Unfortunately, disconnection is bugged in socket.io client. If you close socket connection, `io.connect()` to the same endpoint will silently fail. If you try to forcibly connect associated socket, it will work, but you have to make sure that you're not setting up callbacks again, as you will end up having your callbacks called twice.

Link: https://github.com/LearnBoost/socket.io-client/issues/251

For now, if your main connection was closed, you have few options:

```
var conn = io.connect(addr, {'force new connection': true});
conn.on('message', function(msg) { alert('msg') });
```

or alternative approach:

```
io.j = [];
io.sockets = [];

var conn = io.connect(addr);
conn.on('message', function(msg) { alert('msg') });
```

or separate reconnection from initial connection:

```
var conn = io.connect(addr);
conn.on('disconnect', function(msg) { conn.socket.reconnect(); });
```

If you use first approach, you will lose multiplexing for good.

If you use second approach, apart of it being quite hackish, it will clean up existing sockets, so socket.io will have to create new one and will use it to connect to endpoints. Also, instead of clearing `io.sockets`, you can remove socket which matches your URL.

If you use third approach, make sure you're not setting up events again.

On a side note, if you can avoid using `disconnect()` for socket, do so.

1.9.2 Query string parameters

Current implementation of socket.io client stores query string parameters on session level. So, if you have multiplexed connections and want to pass parameters to them - it is not possible.

See related bug report: https://github.com/LearnBoost/socket.io-client/issues/331

So, you can not expect query string parametes to be passed to your virtual connections and will have to structure your JS code, so first `io.connect()` will include anything you want to pass to the server.

1.9.3 Windows and anti-virus software

It is known that some anti-virus software (I'm lookin at you, Avast) is messing up with websockets protocol if it is going through the port 80. Avast proxies all traffic through local proxy which does some on-the-fly traffic analysis and their proxy does not support websockets - all messages sent from server are queued in their proxy, connection is kept alive, etc.

Unfortunately, socket.io does not support this situation and won't fallback to other protocol.

There are few workarounds: 1. Disable websockets for everyone (duh) 2. Run TornadIO2 (or your proxy/load balancer) on two different ports and have simple logic

> on client side to switch to another port if connection fails

Socket.IO developers are aware of the problem and next socket.io version will provide official workaround.

Here's more detailed article on the matter: https://github.com/LearnBoost/socket.io/wiki/Socket.IO-and-firewall-software.

1.10 API

1.10.1 `tornadio2.conn`

Connection

tornadio2.conn

Tornadio connection implementation.

class `tornadio2.conn.`**`SocketConnection`**(*session*, *endpoint=None*)

Subclass this class and define at least *on_message()* method to make a Socket.IO connection handler.

To support socket.io connection multiplexing, define *_endpoints_* dictionary on class level, where key is endpoint name and value is connection class:

```
class MyConnection(SocketConnection):
    __endpoints__ = {'/clock'=ClockConnection,
                     '/game'=GameConnection}
```

`ClockConnection` and `GameConnection` should derive from the `SocketConnection` class as well.

`SocketConnection` has useful `event` decorator. Wrap method with it:

```
class MyConnection(SocketConnection):
    @event('test')
    def test(self, msg):
        print msg
```

and then, when client will emit 'test' event, you should see 'Hello World' printed:

```
sock.emit('test', {msg:'Hello World'});
```

Callbacks

SocketConnection.**on_open**(*request*)

Default on_open() handler.

Override when you need to do some initialization or request validation. If you return False, connection will be rejected.

You can also throw Tornado HTTPError to close connection.

request `ConnectionInfo` object which contains caller IP address, query string parameters and cookies associated with this request.

For example:

```
class MyConnection(SocketConnection):
    def on_open(self, request):
        self.user_id = request.get_argument('id', None)

        if not self.user_id:
            return False
```

SocketConnection.**on_message**(*message*)

Default on_message handler. Must be overridden in your application

SocketConnection.**on_event**(*name*, *args=[]*, *kwargs={}*)

Default on_event handler.

By default, it uses decorator-based approach to handle events, but you can override it to implement custom event handling.

name Event name

args Event args

kwargs Event kwargs

There's small magic around event handling. If you send exactly one parameter from the client side and it is dict, then you will receive parameters in dict in *kwargs*. In all other cases you will have *args* list.

For example, if you emit event like this on client-side:

```
sock.emit('test', {msg='Hello World'})
```

you will have following parameter values in your on_event callback:

```
name = 'test'
args = []
kwargs = {msg: 'Hello World'}
```

However, if you emit event like this:

```
sock.emit('test', 'a', 'b', {msg='Hello World'})
```

you will have following parameter values:

```
name = 'test'
args = ['a', 'b', {msg: 'Hello World'}]
kwargs = {}
```

SocketConnection.**on_close**()

Default on_close handler.

Output

SocketConnection.**send**(*message*, *callback=None*, *force_json=False*)

Send message to the client.

message Message to send.

callback Optional callback. If passed, callback will be called when client received sent message and sent acknowledgment back.

force_json Optional argument. If set to True (and message is a string) then the message type will be JSON (Type 4 in socket_io protocol). This is what you want, when you send already json encoded strings.

SocketConnection.**emit**(*name*, **args*, ***kwargs*)

Send socket.io event.

name Name of the event

kwargs Optional event parameters

SocketConnection.**emit_ack**(*callback*, *name*, **args*, ***kwargs*)

Send socket.io event with acknowledgment.

callback Acknowledgment callback

name Name of the event

kwargs Optional event parameters

Management

SocketConnection.**close**()

Forcibly close client connection

Endpoint management

SocketConnection.**get_endpoint**(*endpoint*)

Get connection class by endpoint name.

By default, will get endpoint from associated list of endpoints (from __endpoints__ class level variable).

You can override this method to implement different endpoint connection class creation logic.

Other

SocketConnection.**deque_ack**(*msg_id*, *ack_data*)

Dequeue acknowledgment callback

Events

tornadio2.conn.**event**(*name_or_func*)

Event handler decorator.

Can be used with event name or will automatically use function name if not provided:

```
    # Will handle 'foo' event
    @event('foo')
    def bar(self):
        pass

    # Will handle 'baz' event
    @event
```

```
    def baz(self):
        pass
```

1.10.2 tornadio2.flashserver

tornadio2.flashserver

Flash Socket policy server implementation. Merged with minor modifications from the SocketTornad.IO project.

class tornadio2.flashserver.**FlashPolicyServer**(*io_loop*, *port=843*, *policy_file='flashpolicy.xml'*)

Flash Policy server, listens on port 843 by default (useless otherwise)

__init__(*io_loop*, *port=843*, *policy_file='flashpolicy.xml'*)

Constructor.

io_loop IOLoop instance

port Port to listen on (defaulted to 843)

policy_file Policy file location

1.10.3 tornadio2.gen

tornadio2.gen

Generator-based interface to make it easier to work in an asynchronous environment.

Wrapper

tornadio2.gen.**sync_engine**(*func*)

Queued version of the tornado.gen.engine.

Prevents calling of the wrapped function if there is already one instance of the function running asynchronously. Function will be called synchronously without blocking io_loop.

This decorator can only be used on class methods, as it requires self to make sure that calls are scheduled on instance level (connection) instead of class level (method).

Internal API

class tornadio2.gen.**SyncRunner**(*gen*, *callback*)

Customized tornado.gen.Runner, which will notify callback about completion of the generator.

SyncRunner.**__init__**(*gen*, *callback*)

Constructor.

gen Generator

callback Function that should be called upon generator completion

1.10.4 tornadio2.periodic

tornadio.flashserver

This module implements customized PeriodicCallback from tornado with support of the sliding window.

class tornadio2.periodic.**Callback**(*callback*, *callback_time*, *io_loop*)
Custom implementation of the Tornado.Callback with support of callback timeout delays.

__init__(*callback*, *callback_time*, *io_loop*)
Constructor.

callback Callback function

callback_time Callback timeout value (in milliseconds)

io_loop io_loop instance

calculate_next_run()
Caltulate next scheduled run

start(*timeout=None*)
Start callbacks

stop()
Stop callbacks

delay()
Delay callback

1.10.5 tornadio2.persistent

Persistent transports

tornadio2.persistent

Persistent transport implementations.

class tornadio2.persistent.**TornadioWebSocketHandler**(*application*, *request*, ***kwargs*)
Websocket protocol handler

Callbacks

TornadioWebSocketHandler.**open**(*session_id*)
WebSocket open handler

TornadioWebSocketHandler.**on_message**(*message*)

TornadioWebSocketHandler.**on_close**()

TornadioWebSocketHandler.**session_closed**()

Output

TornadioWebSocketHandler.**send_messages**(*messages*)

class tornadio2.persistent.**TornadioFlashSocketHandler**(*application*, *request*, ***kwargs*)

1.10.6 tornadio2.polling

tornadio2.polling

This module implements socket.io polling transports.

class tornadio2.polling.**TornadioPollingHandlerBase**(*application*, *request*, ***kwargs*)
Polling handler base

Request

TornadioPollingHandlerBase.**get**(**args*, ***kwargs*)
Default GET handler.

TornadioPollingHandlerBase.**post**(*session_id*)
Handle incoming POST request

Callbacks

TornadioPollingHandlerBase.**session_closed**()
Called by the session when it was closed

TornadioPollingHandlerBase.**on_connection_close**()
Called by Tornado, when connection was closed

Output

TornadioPollingHandlerBase.**send_messages**(*messages*)
Called by the session when some data is available

Sessions

TornadioPollingHandlerBase.**_get_session**(*session_id*)
Get session if exists and checks if session is closed.

TornadioPollingHandlerBase.**_detach**()
Detach from the session

class tornadio2.polling.**TornadioXHRPollingHandler**(*application*, *request*, ***kwargs*)
xhr-polling transport implementation

class tornadio2.polling.**TornadioHtmlFileHandler**(*application*, *request*, ***kwargs*)
IE HtmlFile protocol implementation.

Uses hidden frame to stream data from the server in one connection.

class tornadio2.polling.**TornadioJSONPHandler**(*application*, *request*, ***kwargs*)

1.10.7 tornadio2.preflight

tornadio2.preflight

Transport protocol router and main entry point for all socket.io clients.

class tornadio2.preflight.**PreflightHandler**(*application*, *request*, ***kwargs*)
CORS preflight handler

options(**args*, ***kwargs*)
XHR cross-domain OPTIONS handler

preflight()
Handles request authentication

1.10.8 tornadio2.proto

tornadio2.proto

Socket.IO protocol related functions

Packets

tornadio2.proto.**disconnect**(*endpoint=None*)
Generate disconnect packet.

endpoint Optional endpoint name

tornadio2.proto.**connect**(*endpoint=None*)
Generate connect packet.

endpoint Optional endpoint name

tornadio2.proto.**heartbeat**()
Generate heartbeat message.

tornadio2.proto.**message**(*endpoint*, *msg*, *message_id=None*, *force_json=False*)
Generate message packet.

endpoint Optional endpoint name

msg Message to encode. If message is ascii/unicode string, will send message packet. If object or dictionary, will json encode and send as is.

message_id Optional message id for ACK

force json Disregard msg type and send the message with JSON type. Usefull for already JSON encoded strings.

tornadio2.proto.**event**(*endpoint*, *name*, *message_id*, **args*, ***kwargs*)
Generate event message.

endpoint Optional endpoint name

name Event name

message_id Optional message id for ACK

args Optional event arguments.

kwargs Optional event arguments. Will be encoded as dictionary.

tornadio2.proto.**ack**(*endpoint*, *message_id*, *ack_response=None*)
Generate ACK packet.

endpoint Optional endpoint name

message_id Message id to acknowledge

ack_response Acknowledgment response data (will be json serialized)

tornadio2.proto.**error**(*endpoint*, *reason*, *advice=None*)
Generate error packet.

endpoint Optional endpoint name

reason Error reason

advice Error advice

tornadio2.proto.**noop**()
Generate noop packet.

JSON

tornadio2.proto.**json_dumps**(*msg*)
Dump object as a json string

msg Object to dump

tornadio2.proto.**json_load**(*msg*)
Load json-encoded object

msg json encoded object

Frames

tornadio2.proto.**decode_frames**(*data*)
Decode socket.io encoded messages. Returns list of packets.

data encoded messages

tornadio2.proto.**encode_frames**(*packets*)
Encode list of packets.

packets List of packets to encode

1.10.9 tornadio2.router

tornadio2.router

Transport protocol router and main entry point for all socket.io clients.

class tornadio2.router.**TornadioRouter**(*connection*, *user_settings={}*, *namespace='socket.io'*, *io_loop=None*)
TornadIO2 router implementation

Public

TornadioRouter.**__init__**(*connection*, *user_settings={}*, *namespace='socket.io'*, *io_loop=None*)
Constructor.

connection SocketConnection class instance

user_settings Settings

namespace Router namespace, defaulted to 'socket.io'

io_loop IOLoop instance, optional.

TornadioRouter.**urls**
List of the URLs to be added to the Tornado application

TornadioRouter.**apply_routes**(*routes*)
Feed list of the URLs to the routes list. Returns list

Sessions

TornadioRouter.**create_session**(*request*)
Creates new session object and returns it.

request Request that created the session. Will be used to get query string parameters and cookies.

TornadioRouter.**get_session**(*session_id*)
Get session by session id

1.10.10 tornadio2.server

tornadio2.server

Implements handy wrapper to start FlashSocket server (if FlashSocket protocol is enabled). Shamesly borrowed from the SocketTornad.IO project.

class tornadio2.server.**SocketServer**(*application*, *no_keep_alive=False*, *io_loop=None*, *xheaders=False*, *ssl_options=None*, *auto_start=True*)
HTTP Server which does some configuration and automatic setup of Socket.IO based on configuration. Starts the IOLoop and listening automatically in contrast to the Tornado default behavior. If FlashSocket is enabled, starts up the policy server also.

__init__(*application*, *no_keep_alive=False*, *io_loop=None*, *xheaders=False*, *ssl_options=None*, *auto_start=True*)
Initializes the server with the given request callback.

If you use pre-forking/start() instead of the listen() method to start your server, you should not pass an IOLoop instance to this constructor. Each pre-forked child process will create its own IOLoop instance after the forking process.

application Tornado application

no_keep_alive Support keep alive for HTTP connections or not

io_loop Optional io_loop instance.

xheaders Extra headers

ssl_options Tornado SSL options

auto_start Set auto_start to False in order to have opportunities to work with server object and/or perform some actions after server is already created but before ioloop will start. Attention: if you use auto_start param set to False you should start ioloop manually

1.10.11 tornadio2.session

Session

tornadio2.session

Active TornadIO2 connection session.

class tornadio2.session.**Session**(*conn*, *server*, *request*, *expiry=None*)

Socket.IO session implementation.

Session has some publicly accessible properties:

server Server association. Server contains io_loop instance, settings, etc.

remote_ip Remote IP

is_closed Check if session is closed or not.

Constructor

Session.**__init__**(*conn*, *server*, *request*, *expiry=None*)

Session constructor.

conn Default connection class

server Associated server

handler Request handler that created new session

expiry Session expiry

Callbacks

Session.**on_delete**(*forced*)

Session expiration callback

forced If session item explicitly deleted, forced will be set to True. If item expired, will be set to False.

Handlers

Session.**set_handler**(*handler*)

Set active handler for the session

handler Associate active Tornado handler with the session

Session.**remove_handler**(*handler*)

Remove active handler from the session

handler Handler to remove

Output

Session.**send_message**(*pack*)

Send socket.io encoded message

pack Encoded socket.io message

Session.**flush**()

Flush message queue if there's an active connection running

State

`Session.close(endpoint=None)`
Close session or endpoint connection.

endpoint If endpoint is passed, will close open endpoint connection. Otherwise will close whole socket.

`Session.is_closed`
Check if session was closed

Heartbeats

`Session.reset_heartbeat()`
Reset hearbeat timer

`Session.stop_heartbeat()`
Stop active heartbeat

`Session.delay_heartbeat()`
Delay active heartbeat

`Session._heartbeat()`
Heartbeat callback

Endpoints

`Session.connect_endpoint(url)`
Connect endpoint from URL.

url socket.io endpoint URL.

`Session.disconnect_endpoint(endpoint)`
Disconnect endpoint

endpoint endpoint name

Messages

`Session.raw_message(msg)`
Socket.IO message handler.

msg Raw socket.io message to handle

Connection information

class `tornadio2.session.ConnectionInfo(ip, arguments, cookies)`
Connection information object.

Will be passed to the `on_open` handler of your connection class.

Has few properties:

ip Caller IP address

cookies Collection of cookies

arguments Collection of the query string arguments

`get_argument(name)`
Return single argument by name

`get_cookie(name)`
Return single cookie by its name

1.10.12 tornadio2.sessioncontainer

tornadio2.sessioncontainer

Simple heapq-based session implementation with sliding expiration window support.

class tornadio2.sessioncontainer.**SessionBase**(*session_id=None*, *expiry=None*)
Represents one session object stored in the session container. Derive from this object to store additional data.

__init__(*session_id=None*, *expiry=None*)
Constructor.

session_id Optional session id. If not provided, will generate new session id.

expiry Expiration time. If not provided, will never expire.

is_alive()
Check if session is still alive

promote()
Mark object as alive, so it won't be collected during next run of the garbage collector.

on_delete(*forced*)
Triggered when object was expired or deleted.

class tornadio2.sessioncontainer.**SessionContainer**

add(*session*)
Add session to the container.

session Session object

get(*session_id*)
Return session object or None if it is not available

session_id Session identifier

remove(*session_id*)
Remove session object from the container

session_id Session identifier

expire(*current_time=None*)
Expire any old entries

current_time Optional time to be used to clean up queue (can be used in unit tests)

1.10.13 tornadio2.stats

tornadio2.stats

Statistics module

class tornadio2.stats.**StatsCollector**
Statistics collector

dump()
Return current statistics

class tornadio2.stats.**MovingAverage**(*period=10*)
Moving average class implementation

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

t

tornadio2.conn, 10
tornadio2.flashserver, 13
tornadio2.gen, 13
tornadio2.periodic, 14
tornadio2.persistent, 14
tornadio2.polling, 15
tornadio2.preflight, 15
tornadio2.proto, 16
tornadio2.router, 17
tornadio2.server, 18
tornadio2.session, 19
tornadio2.sessioncontainer, 21
tornadio2.stats, 21

Symbols

__init__() (tornadio2.flashserver.FlashPolicyServer method), 13
__init__() (tornadio2.gen.SyncRunner method), 13
__init__() (tornadio2.periodic.Callback method), 14
__init__() (tornadio2.router.TornadioRouter method), 17
__init__() (tornadio2.server.SocketServer method), 18
__init__() (tornadio2.session.Session method), 19
__init__() (tornadio2.sessioncontainer.SessionBase method), 21
_detach() (tornadio2.polling.TornadioPollingHandlerBase method), 15
_get_session() (tornadio2.polling.TornadioPollingHandlerBase method), 15
_heartbeat() (tornadio2.session.Session method), 20

A

ack() (in module tornadio2.proto), 16
add() (tornadio2.sessioncontainer.SessionContainer method), 21
apply_routes() (tornadio2.router.TornadioRouter method), 18

C

calculate_next_run() (tornadio2.periodic.Callback method), 14
Callback (class in tornadio2.periodic), 14
close() (tornadio2.conn.SocketConnection method), 12
close() (tornadio2.session.Session method), 20
connect() (in module tornadio2.proto), 16
connect_endpoint() (tornadio2.session.Session method), 20
ConnectionInfo (class in tornadio2.session), 20
create_session() (tornadio2.router.TornadioRouter method), 18

D

decode_frames() (in module tornadio2.proto), 17
delay() (tornadio2.periodic.Callback method), 14
delay_heartbeat() (tornadio2.session.Session method), 20
deque_ack() (tornadio2.conn.SocketConnection method), 12
disconnect() (in module tornadio2.proto), 16
disconnect_endpoint() (tornadio2.session.Session method), 20
dump() (tornadio2.stats.StatsCollector method), 21

E

emit() (tornadio2.conn.SocketConnection method), 12
emit_ack() (tornadio2.conn.SocketConnection method), 12
encode_frames() (in module tornadio2.proto), 17
error() (in module tornadio2.proto), 17
event() (in module tornadio2.conn), 12
event() (in module tornadio2.proto), 16
expire() (tornadio2.sessioncontainer.SessionContainer method), 21

F

FlashPolicyServer (class in tornadio2.flashserver), 13
flush() (tornadio2.session.Session method), 19

G

get() (tornadio2.polling.TornadioPollingHandlerBase method), 15
get() (tornadio2.sessioncontainer.SessionContainer method), 21
get_argument() (tornadio2.session.ConnectionInfo method), 20
get_cookie() (tornadio2.session.ConnectionInfo method), 20
get_endpoint() (tornadio2.conn.SocketConnection method), 12
get_session() (tornadio2.router.TornadioRouter method), 18

H

heartbeat() (in module tornadio2.proto), 16

I

is_alive() (tornadio2.sessioncontainer.SessionBase method), 21
is_closed (tornadio2.session.Session attribute), 20

J

json_dumps() (in module tornadio2.proto), 17
json_load() (in module tornadio2.proto), 17

M

message() (in module tornadio2.proto), 16
MovingAverage (class in tornadio2.stats), 21

N

noop() (in module tornadio2.proto), 17

O

on_close() (tornadio2.conn.SocketConnection method), 11
on_close() (tornadio2.persistent.TornadioWebSocketHandler method), 14
on_connection_close() (tornadio2.polling.TornadioPollingHandlerBase method), 15
on_delete() (tornadio2.session.Session method), 19
on_delete() (tornadio2.sessioncontainer.SessionBase method), 21
on_event() (tornadio2.conn.SocketConnection method), 11
on_message() (tornadio2.conn.SocketConnection method), 11
on_message() (tornadio2.persistent.TornadioWebSocketHandler method), 14
on_open() (tornadio2.conn.SocketConnection method), 11
open() (tornadio2.persistent.TornadioWebSocketHandler method), 14
options() (tornadio2.preflight.PreflightHandler method), 16

P

post() (tornadio2.polling.TornadioPollingHandlerBase method), 15
preflight() (tornadio2.preflight.PreflightHandler method), 16
PreflightHandler (class in tornadio2.preflight), 15
promote() (tornadio2.sessioncontainer.SessionBase method), 21

R

raw_message() (tornadio2.session.Session method), 20
remove() (tornadio2.sessioncontainer.SessionContainer method), 21
remove_handler() (tornadio2.session.Session method), 19
reset_heartbeat() (tornadio2.session.Session method), 20

S

send() (tornadio2.conn.SocketConnection method), 12
send_message() (tornadio2.session.Session method), 19
send_messages() (tornadio2.persistent.TornadioWebSocketHandler method), 14
send_messages() (tornadio2.polling.TornadioPollingHandlerBase method), 15
Session (class in tornadio2.session), 19
session_closed() (tornadio2.persistent.TornadioWebSocketHandler method), 14
session_closed() (tornadio2.polling.TornadioPollingHandlerBase method), 15
SessionBase (class in tornadio2.sessioncontainer), 21
SessionContainer (class in tornadio2.sessioncontainer), 21
set_handler() (tornadio2.session.Session method), 19
SocketConnection (class in tornadio2.conn), 10
SocketServer (class in tornadio2.server), 18
start() (tornadio2.periodic.Callback method), 14
StatsCollector (class in tornadio2.stats), 21
stop() (tornadio2.periodic.Callback method), 14
stop_heartbeat() (tornadio2.session.Session method), 20
sync_engine() (in module tornadio2.gen), 13
SyncRunner (class in tornadio2.gen), 13

T

tornadio2.conn (module), 10
tornadio2.flashserver (module), 13
tornadio2.gen (module), 13
tornadio2.periodic (module), 14
tornadio2.persistent (module), 14
tornadio2.polling (module), 15
tornadio2.preflight (module), 15
tornadio2.proto (module), 16
tornadio2.router (module), 17
tornadio2.server (module), 18
tornadio2.session (module), 19
tornadio2.sessioncontainer (module), 21
tornadio2.stats (module), 21
TornadioFlashSocketHandler (class in tornadio2.persistent), 14
TornadioHtmlFileHandler (class in tornadio2.polling), 15
TornadioJSONPHandler (class in tornadio2.polling), 15
TornadioPollingHandlerBase (class in tornadio2.polling), 15
TornadioRouter (class in tornadio2.router), 17

TornadioWebSocketHandler (class in tornadio2.persistent), 14
TornadioXHRPollingHandler (class in tornadio2.polling), 15

U

urls (tornadio2.router.TornadioRouter attribute), 18