ws4py Documentation

Release 0.5.1

Author

Feb 28, 2018

Contents

Author Sylvain Hellegouarch

Release 0.5.1

License BSD

Source code https://github.com/Lawouach/WebSocket-for-Python

Build status https://travis-ci.org/Lawouach/WebSocket-for-Python

ws4py is a Python package implementing the WebSocket protocol as defined in RFC 6455.

It comes with various server and client implementations and runs on CPython 2/3, PyPy and Android.

CHAPTER 1

Overview

1.1 Requirements

1.1.1 Python

Tested environments:

- Python 2.7+
- Python 3.3+
- PyPy 1.8+
- Android 2.3 via SL4A

Note: ws4py will not try to automatically install dependencies and will let you decide which one you need.

1.1.2 Client

ws4py comes with three client implementations:

- Built-in: That client is solely based on the Python stdlib, it uses a thread to run the inner processing loop.
- Tornado: The Tornado client requires Tornado 2.0+
- gevent: The gevent client requires gevent 0.13.6 or 1.0.0-dev

1.1.3 Server

ws4py comes with three server implementations:

- Built-in: The server relies on the built-in `wsgiref` module.

- CherryPy: The CherryPy server requires 3.2.2+
- gevent: The gevent server requires gevent 1.0.0
- asyncio: **PEP 3156** implementation for Python 3.3+

1.1.4 Testing

ws4py uses the Autobahn functional test suite to ensure it respects the standard. You must install it to run that test suite against ws4py.

- Autobahn python
- Autobahn test suite

1.2 Install ws4py

1.2.1 Get the code

ws4py is hosted on github and can be retrieved from there:

```
$ git clone git@github.com:Lawouach/WebSocket-for-Python.git
```

Installing the ws4py package is performed as usual:

```
$ python setup.py install
```

However, since ws4py is referenced in PyPI, it can also be installed through easy_install, distribute or pip:

```
$ pip install ws4py
$ easy_install ws4py
```

Note: ws4py explicitly will not automatically pull out its dependencies. Please install them manually depending on which implementation you'll be using.

1.3 Conformance

ws4py tries hard to be as conformant as it can to the specification. In order to validate this conformance, each release is run against the Autobahn testsuite which provides an extensive coverage of various aspects of the protocol.

Online test reports can be found at: http://www.defuze.org/oss/ws4py/testreports/servers/0.3.5.

1.4 Browser Support

ws4py has been tested using:

- Chromium 22
- Firefox 16

See http://caniuse.com/websockets to determine the current implementation's status of various browser vendors.

Bear in mind that time is a premium and maintaining obsolete and unsecure protocols is not one of ws4py's goals. It's therefore unlikely it will ever support older versions of the protocol.

1.5 Performances

ws4py doesn't perform too bad but it's far from being the fastest WebSocket lib under heavy load. The reason is that it was first designed to implement the protocol with simplicity and clarity in mind. Future developments will look at performances.

Note: ws4py runs faster in some cases on PyPy than it does on CPython.

Note: The wsaccel package replaces some internal bottleneck with a Cython implementation.

1.6 Credits

Many thanks to the pywebsocket and Tornado projects which have provided a the starting point for ws4py. Thanks also to Jeff Lindsay (progrium) for the initial gevent server support. A well deserved thank you to Tobias Oberstein for Autobahn test suite.

Obviously thanks to all the various folks who have provided bug reports and fixes.

CHAPTER 2

Tutorial

2.1 Basics

ws4py provides a high-level, yet simple, interface to provide your application with WebSocket support. It is simple as:

```
from ws4py.websocket import WebSocket
```

The *WebSocket* class should be sub-classed by your application. To the very least we suggest you override the *received_message(message)* method so that you can process incoming messages.

For instance a straightforward echo application would look like this:

```
class EchoWebSocket(WebSocket):
    def received_message(self, message):
        self.send(message.data, message.is_binary)
```

Other useful methods to implement are:

- *opened()* which is called whenever the WebSocket handshake is done.
- *closed(code, reason=None)* which is called whenever the WebSocket connection is terminated.

You may want to know if the connection is currently usable or *terminated*.

At that stage, the subclass is still not connected to any data source. The way ws4py is designed, you don't necessarily need a connected socket, in fact, you don't even need a socket at all.

```
>>> from ws4py.messaging import TextMessage
>>> def data_source():
>>>     yield TextMessage(u'hello world')

>>> from mock import MagicMock
>>> source = MagicMock(side_effect=data_source)
>>> ws = EchoWebSocket(sock=source)
>>> ws.send(u'hello there')
```

2.2 Client

ws4py comes with various client implementations and they roughly share the same interface.

2.2.1 Built-in

The built-in client relies only on modules provided by the Python stdlib. The client's inner loop runs within a thread and therefore holds the thread alive until the websocket is closed.

```
from ws4py.client.threadedclient import WebSocketClient

class DummyClient(WebSocketClient):
    def opened(self):
        def data_provider():
            for i in range(1, 200, 25):
                yield "#" * i

        self.send(data_provider())

        for i in range(0, 200, 25):
            print i
            self.send("*" * i)

    def closed(self, code, reason=None):
        print "Closed down", code, reason

    def received_message(self, m):
        print m
        if len(m) == 175:
            self.close(reason='Bye bye')

if __name__ == '__main__':
    try:
        ws = DummyClient('ws://localhost:9000/', protocols=['http-only', 'chat'])
        ws.connect()
        ws.run_forever()
    except KeyboardInterrupt:
        ws.close()
```

In this snippet, when the handshake is successful, the *opened()* method is called and within this method we immediately send a bunch of messages to the server. First we demonstrate how you can use a generator to do so, then we simply send strings.

Assuming the server echoes messages as they arrive, the *received_message(message)* method will print out the messages returned by the server and simply close the connection once it receives the last sent messages, which length is 175.

Finally the *closed(code, reason=None)* method is called with the code and reason given by the server.

See also:

Managing a pool of WebSockets.

2.2.2 Tornado

If you are using a Tornado backend you may use the Tornado client that ws4py provides as follow:

```
from ws4py.client.tornadoclient import TornadoWebSocketClient
from tornado import ioloop

class MyClient(TornadoWebSocketClient):
     def opened(self):
         for i in range(0, 200, 25):
             self.send("*" * i)

     def received_message(self, m):
         print m
         if len(m) == 175:
             self.close(reason='Bye bye')

     def closed(self, code, reason=None):
         ioloop.IOLoop.instance().stop()

ws = MyClient('ws://localhost:9000/echo', protocols=['http-only', 'chat'])
ws.connect()

ioloop.IOLoop.instance().start()
```

2.2.3 gevent

If you are using a gevent backend you may use the gevent client that ws4py provides as follow:

```
from ws4py.client.geventclient import WebSocketClient
```

This client can benefit from gevent's concepts as demonstrated below:

```
ws = WebSocketClient('ws://localhost:9000/echo', protocols=['http-only', 'chat'])
ws.connect()

def incoming():
    """
    Greenlet waiting for incoming messages
    until ``None`` is received, indicating we can
    leave the loop.
    """
    while True:
        m = ws.receive()
        if m is not None:
           print str(m)
        else:
           break

def send_a_bunch():
    for i in range(0, 40, 5):
       ws.send("*" * i)

greenlets = [
    gevent.spawn(incoming),
    gevent.spawn(send_a_bunch),
]
gevent.joinall(greenlets)
```

2.3 Server

ws4py comes with a few server implementations built around the main *WebSocket* class.

2.3.1 CherryPy

ws4py provides an extension to CherryPy 3 to enable WebSocket from the framework layer. It is based on the CherryPy plugin and tool mechanisms.

The *WebSocket tool* plays at the request level on every request received by the server. Its goal is to perform the WebSocket handshake and, if it succeeds, to create the *WebSocket* instance (well a subclass you will be implementing) and push it to the plugin.

The *WebSocket plugin* works at the CherryPy system level and has a single instance throughout. Its goal is to track websocket instances created by the tool and free their resources when connections are closed.

Here is a simple example of an echo server:

```
import cherrypy
from ws4py.server.cherrypyserver import WebSocketPlugin, WebSocketTool
from ws4py.websocket import EchoWebSocket

cherrypy.config.update({'server.socket_port': 9000})
WebSocketPlugin(cherrypy.engine).subscribe()
cherrypy.tools.websocket = WebSocketTool()

class Root(object):
    @cherrypy.expose
    def index(self):
        return 'some HTML with a websocket javascript connection'

    @cherrypy.expose
    def ws(self):
        # you can access the class instance through
        handler = cherrypy.request.ws_handler

cherrypy.quickstart(Root(), '/', config={'/ws': {'tools.websocket.on': True,
                                                 'tools.websocket.handler_cls':␣
↪EchoWebSocket}})
```

Note how we specify the class which should be instanciated by the server on each connection. The great aspect of the tool mechanism is that you can specify a different class on a per-path basis.

2.3.2 gevent

gevent is a coroutine, called greenlets, implementation for very concurrent applications. ws4py offers a server implementation for this library on top of the WSGI protocol. Using it is as simple as:

```
from gevent import monkey; monkey.patch_all()
from ws4py.websocket import EchoWebSocket
from ws4py.server.geventserver import WSGIServer
from ws4py.server.wsgiutils import WebSocketWSGIApplication

server = WSGIServer(('localhost', 9000), WebSocketWSGIApplication(handler_
↪cls=EchoWebSocket))
server.serve_forever()
```

First we patch all the standard modules so that the stdlib runs well with as gevent. Then we simply create a WSGI server and specify the class which will be instanciated internally each time a connection is successful.

2.3.3 wsgiref

`wsgiref` is a built-in WSGI package that provides various classes and helpers to develop against WSGI. Mostly it provides a basic WSGI server that can be usedfor testing or simple demos. ws4py provides support for websocket on wsgiref for testing purpose as well. It's not meant to be used in production, since it can only initiate web socket connections one at a time, as a result of being single threaded. However, once accepted, ws4py takes over, which is multithreaded by default.

```
from wsgiref.simple_server import make_server
from ws4py.websocket import EchoWebSocket
from ws4py.server.wsgirefserver import WSGIServer, WebSocketWSGIRequestHandler
from ws4py.server.wsgiutils import WebSocketWSGIApplication

server = make_server('', 9000, server_class=WSGIServer,
                     handler_class=WebSocketWSGIRequestHandler,
                     app=WebSocketWSGIApplication(handler_cls=EchoWebSocket))
server.initialize_websockets_manager()
server.serve_forever()
```

2.3.4 asyncio

`asyncio` is the implementation of **PEP 3156**, the new asynchronous framework for concurrent applications.

```
from ws4py.async_websocket import EchoWebSocket

loop = asyncio.get_event_loop()

def start_server():
    proto_factory = lambda: WebSocketProtocol(EchoWebSocket)
    return loop.create_server(proto_factory, '', 9007)

s = loop.run_until_complete(start_server())
print('serving on', s.sockets[0].getsockname())
loop.run_forever()
```

Warning: The provided HTTP server used for the handshake is clearly not production ready. However, once the handshake is performed, the rest of the code runs the same stack as the other server implementations. It should be easy to replace the HTTP interface with any asyncio aware HTTP framework.

2.4 Managing a pool of WebSockets

ws4py provides a *ws4py.manager.WebSocketManager* class that takes care of *ws4py.websocket.WebSocket* instances once they the HTTP upgrade handshake has been performed.

The manager is not compulsory but makes it simpler to track and let them run in your application's process.

When you *add(websocket)* a websocket to the manager, the file-descriptor is registered with the manager's poller and the *opened()* method on is called.

2.4.1 Polling

The manager uses a polling mechanism to dispatch on socket incoming events. Two pollers are implemented, one using the traditionnal `select` and another one based on select.epoll which is used only if available on the system.

The polling is executed in its own thread, it keeps looping until the manager `stop()` method.

On every loop, the poller is called to poll for all registered file-descriptors. If any one of them is ready, we retrieve the websocket using that descriptor and, if the websocket is not yet terminated, we call its *once* method so that the incoming bytes are processed.

If the processing fails in anyway, the manager terminates the websocket and remove it from itself.

2.4.2 Client example

Below is a simple example on how to start 2000 clients against a single server.

```
from ws4py.client import WebSocketBaseClient
from ws4py.manager import WebSocketManager
from ws4py import format_addresses, configure_logger

logger = configure_logger()

m = WebSocketManager()

class EchoClient(WebSocketBaseClient):
    def handshake_ok(self):
        logger.info("Opening %s" % format_addresses(self))
        m.add(self)

    def received_message(self, msg):
        logger.info(str(msg))

if __name__ == '__main__':
    import time

    try:
        m.start()
        for i in range(2000):
            client = EchoClient('ws://localhost:9000/ws')
            client.connect()

        logger.info("%d clients are connected" % i)

        while True:
            for ws in m.websockets.itervalues():
                if not ws.terminated:
                   break
            else:
                break
            time.sleep(3)
    except KeyboardInterrupt:
        m.close_all()
        m.stop()
        m.join()
```

Once those are created against the `echo_cherrypy_server` example for instance, point your browser to http://localhost:9000/ and enter a message. It will be broadcasted to all connected peers.

When a peer is closed, its connection is automatically removed from the manager so you should never need to explicitely remove it.

Note: The CherryPy and wsgiref servers internally use a manager to handle connected websockets. The gevent server relies only on a greenlet group instead.

2.5 Built-in Examples

2.5.1 Real-time chat

The `echo_cherrypy_server` example provides a simple Echo server. It requires CherryPy 3.2.3. Open a couple of tabs pointing at http://localhost:9000 and chat around.

2.5.2 Android sensors and HTML5

The `droid_sensor_cherrypy_server` broadcasts sensor metrics to clients. Point your browser to http://localhost:9000 Then run the `droid_sensor` module from your Android device using SL4A.

A screenshot of what this renders to can be found here.

You will find a lovely video of this demo in action on YouTube thanks to Mat Bettinson for this.

CHAPTER 3

Maintainer Guide

This section describes the steps to work on ws4py itself and its release process, as well as other conventions and best practices.

3.1 Coding Rules

Python is a rather flexible language which favors conventions over configurations. This is why, over the years, some community rules were published that most Python developers follow. ws4py tries to follow those principles for the most part and therefore.

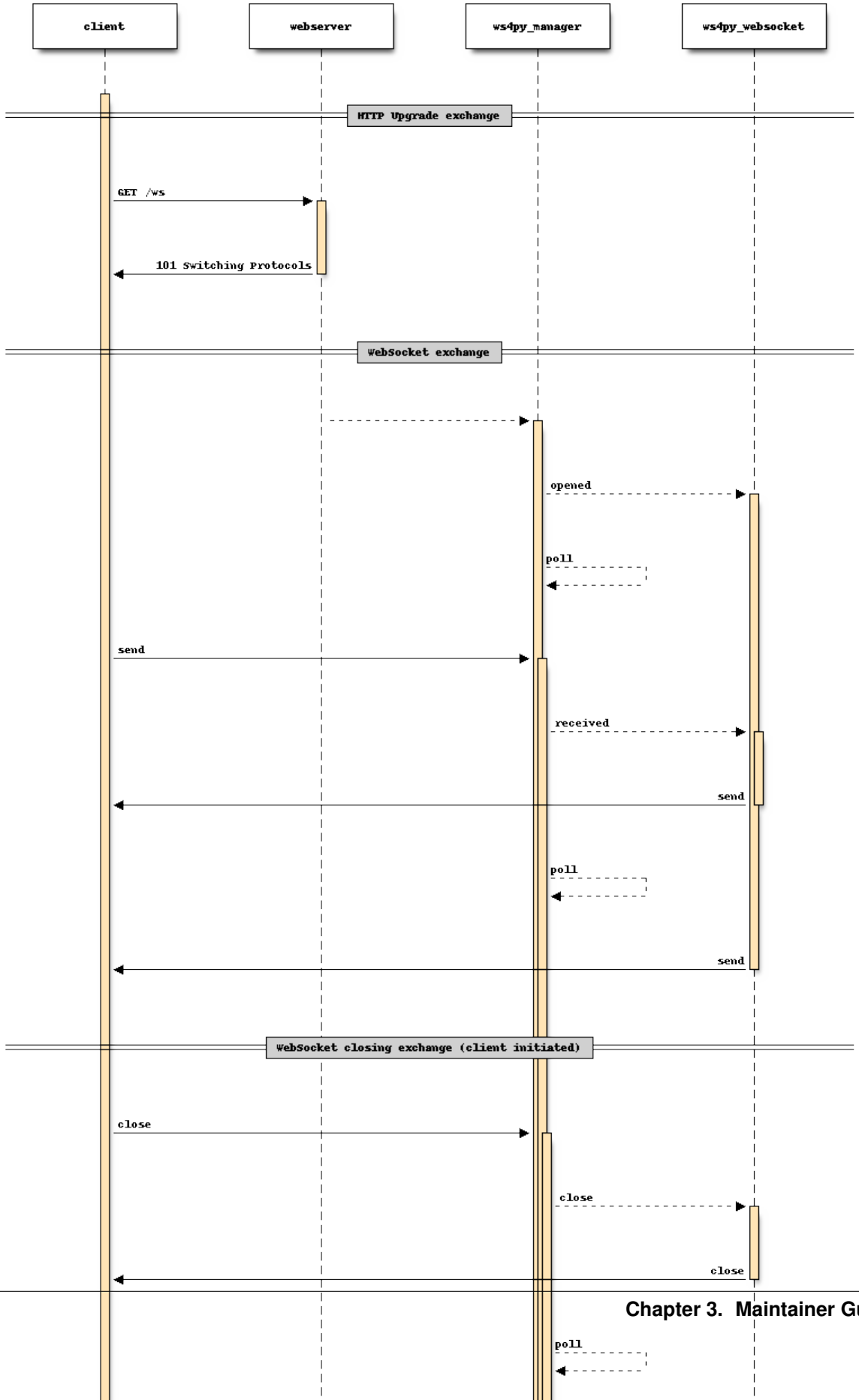
Therefore please carefully read:

- **PEP 8** that suggests a set of coding rules and naming conventions
- **PEP 20** for the spirit behind Python coding

3.2 Design

3.2.1 Workflow

ws4py's design is actually fairly simple and straigtforward. At a high level this is what is going on:


client
webserver
ws4py_manager
ws4py_websocket
HTTP Upgrade exchange
GET /ws
101 Switching Protocols
WebSocket exchange
opened
poll
send
received
send
poll
send
WebSocket closing exchange (client initiated)
close
close
close
poll

The initial connection is made by a WebSocket aware client to a WebSocket aware web server. The first exchange is dedicated to perform the upgrade handshake as defined by **RFC 6455#section-4**.

If the exchange succeeds, the socket is kept opened, wrapped into a *ws4py.websocket.WebSocket* instance which is passed on to the global ws4py *ws4py.manager.WebSocketManager* instance which will handle its lifecycle.

Most notably, the manager will poll for the socket's receive events so that, when bytes are available, the websocket object can read and process them.

3.2.2 Implementation

ws4py data model is rather simple and follows the protocol itself:

- a highlevel *ws4py.websocket.WebSocket* class that determines actions to carry based on messages that are parsed.
- a *ws4py.streaming.Stream* class that handles a single message at a time
- a *ws4py.framing.Frame* class that performs the low level protocol parsing of frames

Each are inter-connected as russian dolls generators. The process heavily relies on the capacity to send to a generator. So everytime one of those layers requires something, it yields and then its holder sends it back whatever was required.

The Frame parser yields the number of bytes it needs at any time, the stream parser forwards it back to the WebSocket class which gets data from the underlying data provider it holds a reference to (a socket typically). The WebSocket class sends bytes as they are read from the socket down to the stream parser which forwards them to the frame parser.

Eventually a frame is parsed and handled by the stream parser which in turns yields a complete message made of all parsed frames.

The interesting aspect here is that the socket provider is totally abstracted from the protocol implementation which simply requires bytes as they come.

This means one could write a ws4py socket provider that doesn't read from the wire but from any other source.

It's also pretty fast and easy to read.

3.3 Testing Overview

ws4py is a Python library which means it can be tested in various fashion:

- unit testing
- functional testing
- load testing

Though all of them are of useful, ws4py mostly relies on functional testing.

3.3.1 Unit Testing

Unit testing solves complex issues in a simple fashion:

- Micro-validation of classes and functions
- Ensure non-regression after modifications
- Critique the design as early as possible for minimum impact

Too often, developers focus solely on the first two and fail to realise how much feedback they can get by writing a simple unit tests. Usually starting writing unit tests can take time because your code is too tightly coupled with itself or external dependencies. This should not the case, most of the time anyway. So make sure to reflect on your code design whenever you have difficulties setting up proper unit tests.

Note: Unfortunately, for now ws4py has a rather shallow coverage as it relies more on the functional testing to ensure the package is sane. I hope to change this in the future.

Framework

ws4py uses the Python built-in `unittest` module. Make sure you read its extensive documentation.

Execution

Test execution can be done as follow:

```
cd test

python -m unittest discover  # execute all tests in the current directory
```

Tests can obviously be executed via nose, unittest2 or py.test if you prefer.

3.3.2 Functional Testing

ws4py relies heavily on the extensive testing suite provided by the Autobahn project.

The server test suite is used by many other WebSocket implementation out there and provides a great way to validate interopability so it must be executed before each release to the least. Please refer to the *Requirements* page to install the test suite.

Execution

- Start the CherryPy server with PyPy 1.9

  ```
  pypy test/autobahn_test_servers.py --run-cherrypy-server-pypy
  ```

- Start the CherryPy server with Python 3.2 and/or 3.3 if you can.

  ```
  python3 test/autobahn_test_servers.py --run-cherrypy-server-py3k
  ```

- Start all servers with Python 2.7

  ```
  python2 test/autobahn_test_servers.py --run-all
  ```

- Finally, execute the test suite as follow:

  ```
  wstest -m fuzzingclient -s test/fuzzingclient.json
  ```

The whole test suite will take a while to complete so be patient.

3.4 Documentation process

3.4.1 Basic principles

Document in that order: why, what, how

Documenting ws4py is an important process since the code doesn't always carry enough information to understand its design and context. Thus, documenting should target the question "why?" first then the "what?" and "how?". It's actually trickier than it sound.

Explicit is better than implicit

When you have your nose in the code it may sound straightforward enough not to document certain aspects of pyalm. Remember that **PEP 20** principle: **Explicit is better than implicit**.

Be clear, not verbose

Finding the right balance between too much and not enough is hard. Writing good documentation is just difficult. However, you should not be too verbose either.

Add enough details to a section to provide context but don't flood the reader with irrelevant information.

Show me where you come from

Every piece of code should be contextualized and almost every time you should explicitely indicate the import statement so that the reader doesn't wonder where an object comes from.

Consistency for the greater good

A user of ws4py should feel at ease with any part of the library and shouldn't have to switch to a completely new mental model when going through the library or documentation. Please refer again to **PEP 8**.

3.4.2 Documentation Toolkit

pyalm uses the Sphinx documentation generator toolkit so refer to its documentation to learn more on its usage.

Building the documentation is as simple as:

```
cd docs
make html
```

The generated documentation will be available in `docs\_build\html`.

3.5 Release Process

ws4py's release process is as follow:

1. Update the release minor or micro version.

If necessary change also the major version. This should be saved only for major modifications and/or API compatibility breakup.

Edit `ws4py/__init__.py` accordingly. This will propagate to the `setup.py` and `docs/conf.py` appropriately on its own.

See also:

How to version? You should read this.

2. Run the unit test suites

   It's simple, fast and will make you sleep well at night. So do it.

   If the test suite fails, do not release. It's a simple rule we constantly fail for some reason. So if it fails, go back and fix it.

3. Rebuild the documentation

   It may sound funny but a release with an out of date documentation has little value. Keeping your documentation up to date is as important as having no failing unit tests.

   Add to subversion any new documentation pages, both their sources and the resulting HTML files.

4. Build the source package

   First delete the `build` directory.

   Run the following command:

   ```
   python setup.py sdist --formats=gztar
   ```

   This will produce a tarball in the `dist` directory.

5. Push the release to PyPI
6. Tag the release in github
7. Announce it to the world :)

CHAPTER 4

Packages

4.1 ws4py Package

4.1.1 ws4py Package

ws4py.__init__.**configure_logger**(*stdout=True*, *filepath=None*, *level=20*)

ws4py.__init__.**format_addresses**(*ws*)

4.1.2 exc Module

exception ws4py.exc.**WebSocketException**
Bases: exceptions.Exception

exception ws4py.exc.**ProtocolException**
Bases: *ws4py.exc.WebSocketException*

exception ws4py.exc.**FrameTooLargeException**
Bases: *ws4py.exc.WebSocketException*

exception ws4py.exc.**UnsupportedFrameTypeException**
Bases: *ws4py.exc.WebSocketException*

exception ws4py.exc.**UnsupportedFrameTypeException**
Bases: *ws4py.exc.WebSocketException*

exception ws4py.exc.**TextFrameEncodingException**
Bases: *ws4py.exc.WebSocketException*

exception ws4py.exc.**TextFrameEncodingException**
Bases: *ws4py.exc.WebSocketException*

exception ws4py.exc.**InvalidBytesError**
Bases: *ws4py.exc.WebSocketException*

exception ws4py.exc.**StreamClosed**
Bases: exceptions.Exception

exception ws4py.exc.**HandshakeError**(*msg*)
Bases: *ws4py.exc.WebSocketException*

4.1.3 framing Module

class ws4py.framing.**Frame**(*opcode=None*, *body=''*, *masking_key=None*, *fin=0*, *rsv1=0*, *rsv2=0*, *rsv3=0*)
Bases: object

Implements the framing protocol as defined by RFC 6455.

```
>>> test_mask = 'XXXXXX' # perhaps from os.urandom(4)
>>> f = Frame(OPCODE_TEXT, 'hello world', masking_key=test_mask, fin=1)
>>> bytes = f.build()
>>> bytes.encode('hex')
'818bbe04e66ad6618a06d1249105cc6882'
>>> f = Frame()
>>> f.parser.send(bytes[0])
1
>>> f.parser.send(bytes[1])
4
```

See also:

Data Framing http://tools.ietf.org/html/rfc6455#section-5.2

parser

build()
Builds a frame from the instance's attributes and returns its bytes representation.

mask(*data*)
Performs the masking or unmasking operation on data using the simple masking algorithm:

unmask(*data*)
Performs the masking or unmasking operation on data using the simple masking algorithm:

4.1.4 manager Module

The manager module provides a selected classes to handle websocket's execution.

Initially the rationale was to:

- Externalize the way the CherryPy server had been setup as its websocket management was too tightly coupled with the plugin implementation.
- Offer a management that could be used by other server or client implementations.
- Move away from the threaded model to the event-based model by relying on *select* or *epoll* (when available).

A simple usage for handling websocket clients:

```
from ws4py.client import WebSocketBaseClient
from ws4py.manager import WebSocketManager

m = WebSocketManager()
```

(continues on next page)

(continued from previous page)

```
class EchoClient(WebSocketBaseClient):
    def handshake_ok(self):
        m.add(self)  # register the client once the handshake is done

    def received_message(self, msg):
        print str(msg)

m.start()

client = EchoClient('ws://localhost:9000/ws')
client.connect()

m.join()  # blocks forever
```

Managers are not compulsory but hopefully will help your workflow. For clients, you can still rely on threaded, gevent or tornado based implementations of course.

class ws4py.manager.**SelectPoller**(*timeout=0.1*)
Bases: object

A socket poller that uses the *select* implementation to determines which file descriptors have data available to read.

It is available on all platforms.

release()
Cleanup resources.

register(*fd*)
Register a new file descriptor to be part of the select polling next time around.

unregister(*fd*)
Unregister the given file descriptor.

poll()
Polls once and returns a list of ready-to-be-read file descriptors.

class ws4py.manager.**EPollPoller**(*timeout=0.1*)
Bases: object

An epoll poller that uses the `epoll` implementation to determines which file descriptors have data available to read.

Available on Unix flavors mostly.

release()
Cleanup resources.

register(*fd*)
Register a new file descriptor to be part of the select polling next time around.

unregister(*fd*)
Unregister the given file descriptor.

poll()
Polls once and yields each ready-to-be-read file-descriptor

class ws4py.manager.**KQueuePoller**(*timeout=0.1*)
Bases: object

An epoll poller that uses the `epoll` implementation to determines which file descriptors have data available to read.

Available on Unix flavors mostly.

release()
Cleanup resources.

register(*fd*)
Register a new file descriptor to be part of the select polling next time around.

unregister(*fd*)
Unregister the given file descriptor.

poll()
Polls once and yields each ready-to-be-read file-descriptor

class ws4py.manager.**WebSocketManager**(*poller=None*)
Bases: `threading.Thread`

An event-based websocket manager. By event-based, we mean that the websockets will be called when their sockets have data to be read from.

The manager itself runs in its own thread as not to be the blocking mainloop of your application.

The poller's implementation is automatically chosen with `epoll` if available else `select` unless you provide your own `poller`.

add(*websocket*)
Manage a new websocket.

First calls its *opened()* method and register its socket against the poller for reading events.

remove(*websocket*)
Remove the given `websocket` from the manager.

This does not call its *closed()* method as it's out-of-band by your application or from within the manager's run loop.

stop()
Mark the manager as terminated and releases its resources.

run()
Manager's mainloop executed from within a thread.

Constantly poll for read events and, when available, call related websockets' *once* method to read and process the incoming data.

If the *once()* method returns a *False* value, its *terminate()* method is also applied to properly close the websocket and its socket is unregistered from the poller.

Note that websocket shouldn't take long to process their data or they will block the remaining websockets with data to be handled. As for what long means, it's up to your requirements.

close_all(*code=1001*, *message='Server is shutting down'*)
Execute the *close()* method of each registered websockets to initiate the closing handshake. It doesn't wait for the handshake to complete properly.

broadcast(*message*, *binary=False*)
Broadcasts the given message to all registered websockets, at the time of the call.

Broadcast may fail on a given registered peer but this is silent as it's not the method's purpose to handle websocket's failures.

4.1.5 messaging Module

class ws4py.messaging.**Message**(*opcode*, *data=''*, *encoding='utf-8'*)
Bases: object

A message is a application level entity. It's usually built from one or many frames. The protocol defines several kind of messages which are grouped into two sets:

- data messages which can be text or binary typed
- control messages which provide a mechanism to perform in-band control communication between peers

The opcode indicates the message type and data is the possible message payload.

The payload is held internally as a a bytearray as they are faster than pure strings for append operations.

Unicode data will be encoded using the provided encoding.

single(*mask=False*)
Returns a frame bytes with the fin bit set and a random mask.

If mask is set, automatically mask the frame using a generated 4-byte token.

fragment(*first=False*, *last=False*, *mask=False*)
Returns a *ws4py.framing.Frame* bytes.

The behavior depends on the given flags:

- first: the frame uses self.opcode else a continuation opcode
- last: the frame has its fin bit set
- mask: the frame is masked using a automatically generated 4-byte token

completed
Indicates the the message is complete, meaning the frame's fin bit was set.

extend(*data*)
Add more data to the message.

class ws4py.messaging.**TextMessage**(*text=None*)
Bases: *ws4py.messaging.Message*

is_binary

is_text

class ws4py.messaging.**BinaryMessage**(*bytes=None*)
Bases: *ws4py.messaging.Message*

is_binary

is_text

class ws4py.messaging.**CloseControlMessage**(*code=1000*, *reason=''*)
Bases: *ws4py.messaging.Message*

class ws4py.messaging.**PingControlMessage**(*data=None*)
Bases: *ws4py.messaging.Message*

class ws4py.messaging.**PongControlMessage**(*data*)
Bases: *ws4py.messaging.Message*

4.1.6 streaming Module

class ws4py.streaming.**Stream**(*always_mask=False*, *expect_masking=True*)
Bases: object

Represents a websocket stream of bytes flowing in and out.

The stream doesn't know about the data provider itself and doesn't even know about sockets. Instead the stream simply yields for more bytes whenever it requires them. The stream owner is responsible to provide the stream with those bytes until a frame can be interpreted.

```
>>> s = Stream()
>>> s.parser.send(BYTES)
>>> s.has_messages
False
>>> s.parser.send(MORE_BYTES)
>>> s.has_messages
True
>>> s.message
<TextMessage ... >
```

Set `always_mask` to mask all frames built.

Set `expect_masking` to indicate masking will be checked on all parsed frames.

message = None
Parsed test or binary messages. Whenever the parser reads more bytes from a fragment message, those bytes are appended to the most recent message.

pings = None
Parsed ping control messages. They are instances of *ws4py.messaging.PingControlMessage*

pongs = None
Parsed pong control messages. They are instances of *ws4py.messaging.PongControlMessage*

closing = None
Parsed close control messsage. Instance of *ws4py.messaging.CloseControlMessage*

errors = None
Detected errors while parsing. Instances of *ws4py.messaging.CloseControlMessage*

parser

text_message(*text*)
Returns a *ws4py.messaging.TextMessage* instance ready to be built. Convenience method so that the caller doesn't need to import the *ws4py.messaging.TextMessage* class itself.

binary_message(*bytes*)
Returns a *ws4py.messaging.BinaryMessage* instance ready to be built. Convenience method so that the caller doesn't need to import the *ws4py.messaging.BinaryMessage* class itself.

has_message
Checks if the stream has received any message which, if fragmented, is now completed.

close(*code=1000*, *reason=''*)
Returns a close control message built from a *ws4py.messaging.CloseControlMessage* instance, using the given status `code` and `reason` message.

ping(*data=''*)
Returns a ping control message built from a *ws4py.messaging.PingControlMessage* instance.

pong(*data=''*)
Returns a ping control message built from a *ws4py.messaging.PongControlMessage* instance.

receiver()
Parser that keeps trying to interpret bytes it is fed with as incoming frames part of a message.

Control message are single frames only while data messages, like text and binary, may be fragmented accross frames.

The way it works is by instanciating a `wspy.framing.Frame` object, then running its parser generator which yields how much bytes it requires to performs its task. The stream parser yields this value to its caller and feeds the frame parser.

When the frame parser raises `StopIteration`, the stream parser tries to make sense of the parsed frame. It dispatches the frame's bytes to the most appropriate message type based on the frame's opcode.

Overall this makes the stream parser totally agonstic to the data provider.

4.1.7 `utf8validator` Module

class ws4py.utf8validator.**Utf8Validator**
Bases: `object`

Incremental UTF-8 validator with constant memory consumption (minimal state).

Implements the algorithm "Flexible and Economical UTF-8 Decoder" by Bjoern Hoehrmann (http://bjoern.hoehrmann.de/utf-8/decoder/dfa/).

UTF8VALIDATOR_DFA = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

UTF8_ACCEPT = 0

UTF8_REJECT = 1

decode(*b*)
Eat one UTF-8 octet, and validate on the fly.

Returns UTF8_ACCEPT when enough octets have been consumed, in which case self.codepoint contains the decoded Unicode code point.

Returns UTF8_REJECT when invalid UTF-8 was encountered.

Returns some other positive integer when more octets need to be eaten.

reset()
Reset validator to start new incremental UTF-8 decode/validation.

validate(*ba*)
Incrementally validate a chunk of bytes provided as bytearray.

Will return a quad (valid?, endsOnCodePoint?, currentIndex, totalIndex).

As soon as an octet is encountered which renders the octet sequence invalid, a quad with valid? == False is returned. currentIndex returns the index within the currently consumed chunk, and totalIndex the index within the total consumed sequence that was the point of bail out. When valid? == True, currentIndex will be len(ba) and totalIndex the total amount of consumed bytes.

4.1.8 `websocket` Module

class ws4py.websocket.**Heartbeat**(*websocket*, *frequency=2.0*)
Bases: `threading.Thread`

Runs at a periodic interval specified by *frequency* by sending an unsolicitated pong message to the connected peer.

If the message fails to be sent and a socket error is raised, we close the websocket socket automatically, triggering the *closed* handler.

stop()

run()
Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class ws4py.websocket.**WebSocket**(*sock*, *protocols=None*, *extensions=None*, *environ=None*, *heartbeat_freq=None*)

Bases: object

The `sock` is an opened connection resulting from the websocket handshake.

If `protocols` is provided, it is a list of protocols negotiated during the handshake as is `extensions`.

If `environ` is provided, it is a copy of the WSGI environ dictionnary from the underlying WSGI server.

stream = None
Underlying websocket stream that performs the websocket parsing to high level objects. By default this stream never masks its messages. Clients using this class should set the `stream.always_mask` fields to `True` and `stream.expect_masking` fields to `False`.

protocols = None
List of protocols supported by this endpoint. Unused for now.

extensions = None
List of extensions supported by this endpoint. Unused for now.

sock = None
Underlying connection.

client_terminated = None
Indicates if the client has been marked as terminated.

server_terminated = None
Indicates if the server has been marked as terminated.

reading_buffer_size = None
Current connection reading buffer size.

environ = None
WSGI environ dictionary.

heartbeat_freq = None
At which interval the heartbeat will be running. Set this to *0* or *None* to disable it entirely.

local_address
Local endpoint address as a tuple

peer_address
Peer endpoint address as a tuple

opened()
Called by the server when the upgrade handshake has succeeded.

close(*code=1000*, *reason=''*)
Call this method to initiate the websocket connection closing by sending a close frame to the connected peer. The `code` is the status code representing the termination's reason.

Once this method is called, the `server_terminated` attribute is set. Calling this method several times is safe as the closing frame will be sent only the first time.

See also:

Defined Status Codes http://tools.ietf.org/html/rfc6455#section-7.4.1

closed(*code*, *reason=None*)
Called when the websocket stream and connection are finally closed. The provided `code` is status set by the other point and `reason` is a human readable message.

See also:

Defined Status Codes http://tools.ietf.org/html/rfc6455#section-7.4.1

terminated
Returns `True` if both the client and server have been marked as terminated.

connection

close_connection()
Shutdowns then closes the underlying connection.

ping(*message*)
Send a ping message to the remote peer. The given *message* must be a unicode string.

ponged(*pong*)
Pong message, as a `messaging.PongControlMessage` instance, received on the stream.

received_message(*message*)
Called whenever a complete `message`, binary or text, is received and ready for application's processing.

The passed message is an instance of `messaging.TextMessage` or `messaging.BinaryMessage`.

Note: You should override this method in your subclass.

---

unhandled_error(*error*)
Called whenever a socket, or an OS, error is trapped by ws4py but not managed by it. The given error is an instance of *socket.error* or *OSError*.

Note however that application exceptions will not go through this handler. Instead, do make sure you protect your code appropriately in *received_message* or *send*.

The default behaviour of this handler is to log the error with a message.

send(*payload*, *binary=False*)
Sends the given `payload` out.

If `payload` is some bytes or a bytearray, then it is sent as a single message not fragmented.

If `payload` is a generator, each chunk is sent as part of fragmented message.

If `binary` is set, handles the payload as a binary message.

once()
Performs the operation of reading from the underlying connection in order to feed the stream of bytes.

Because this needs to support SSL sockets, we must always read as much as might be in the socket at any given time, however process expects to have itself called with only a certain number of bytes at a time. That number is found in self.reading_buffer_size, so we read everything into our own buffer, and then from there feed self.process.

Then the stream indicates whatever size must be read from the connection since it knows the frame payload length.

It returns *False* if an error occurred at the socket level or during the bytes processing. Otherwise, it returns *True*.

terminate()
Completes the websocket by calling the *closed* method either using the received closing code and reason, or when none was received, using the special *1006* code.

Finally close the underlying connection for good and cleanup resources by unsetting the *environ* and *stream* attributes.

process(*bytes*)
Takes some bytes and process them through the internal stream's parser. If a message of any kind is found, performs one of these actions:

- A closing message will initiate the closing handshake
- Errors will initiate a closing handshake
- A message will be passed to the `received_message` method
- Pings will see pongs be sent automatically
- Pongs will be passed to the `ponged` method

The process should be terminated when this method returns `False`.

run()
Performs the operation of reading from the underlying connection in order to feed the stream of bytes.

We start with a small size of two bytes to be read from the connection so that we can quickly parse an incoming frame header. Then the stream indicates whatever size must be read from the connection since it knows the frame payload length.

Note that we perform some automatic opererations:

- On a closing message, we respond with a closing message and finally close the connection
- We respond to pings with pong messages.
- Whenever an error is raised by the stream parsing, we initiate the closing of the connection with the appropiate error code.

This method is blocking and should likely be run in a thread.

class ws4py.websocket.**EchoWebSocket**(*sock*, *protocols=None*, *extensions=None*, *environ=None*, *heartbeat_freq=None*)
Bases: *ws4py.websocket.WebSocket*

The `sock` is an opened connection resulting from the websocket handshake.

If `protocols` is provided, it is a list of protocols negotiated during the handshake as is `extensions`.

If `environ` is provided, it is a copy of the WSGI environ dictionnary from the underlying WSGI server.

received_message(*message*)
Automatically sends back the provided `message` to its originating endpoint.

4.1.9 Subpackages

client Package

client Package

class ws4py.client.**WebSocketBaseClient**(*url*, *protocols=None*, *extensions=None*, *heartbeat_freq=None*, *ssl_options=None*, *headers=None*, *exclude_headers=None*)

Bases: *ws4py.websocket.WebSocket*

A websocket client that implements **RFC 6455** and provides a simple interface to communicate with a websocket server.

This class works on its own but will block if not run in its own thread.

When an instance of this class is created, a `socket` is created. If the connection is a TCP socket, the nagle's algorithm is disabled.

The address of the server will be extracted from the given websocket url.

The websocket key is randomly generated, reset the *key* attribute if you want to provide yours.

For instance to create a TCP client:

```
>>> from ws4py.client import WebSocketBaseClient
>>> ws = WebSocketBaseClient('ws://localhost/ws')
```

Here is an example for a TCP client over SSL:

```
>>> from ws4py.client import WebSocketBaseClient
>>> ws = WebSocketBaseClient('wss://localhost/ws')
```

Finally an example of a Unix-domain connection:

```
>>> from ws4py.client import WebSocketBaseClient
>>> ws = WebSocketBaseClient('ws+unix:///tmp/my.sock')
```

Note that in this case, the initial Upgrade request will be sent to `/`. You may need to change this by setting the resource explicitely before connecting:

```
>>> from ws4py.client import WebSocketBaseClient
>>> ws = WebSocketBaseClient('ws+unix:///tmp/my.sock')
>>> ws.resource = '/ws'
>>> ws.connect()
```

You may provide extra headers by passing a list of tuples which must be unicode objects.

bind_addr
Returns the Unix socket path if or a tuple `(host, port)` depending on the initial URL's scheme.

close(*code=1000*, *reason=''*)
Initiate the closing handshake with the server.

connect()
Connects this websocket and starts the upgrade handshake with the remote endpoint.

handshake_headers
List of headers appropriate for the upgrade handshake.

handshake_request
Prepare the request to be sent for the upgrade handshake.

process_response_line(*response_line*)
Ensure that we received a HTTP *101* status code in response to our request and if not raises `HandshakeError`.

process_handshake_header(*headers*)
Read the upgrade handshake's response headers and validate them against **RFC 6455**.

handshake_ok()

geventclient Module

threadedclient Module

class ws4py.client.threadedclient.**WebSocketClient**(*url*, *protocols=None*, *extensions=None*, *heartbeat_freq=None*, *ssl_options=None*, *headers=None*, *exclude_headers=None*)

Bases: *ws4py.client.WebSocketBaseClient*

```
from ws4py.client.threadedclient import WebSocketClient

class EchoClient(WebSocketClient):
    def opened(self):
       for i in range(0, 200, 25):
          self.send("*" * i)

    def closed(self, code, reason):
       print(("Closed down", code, reason))

    def received_message(self, m):
       print("=> %d %s" % (len(m), str(m)))

try:
    ws = EchoClient('ws://localhost:9000/echo', protocols=['http-only', 'chat'])
    ws.connect()
except KeyboardInterrupt:
   ws.close()
```

daemon
True if the client's thread is set to be a daemon thread.

run_forever()
Simply blocks the thread until the websocket has terminated.

handshake_ok()
Called when the upgrade handshake has completed successfully.

Starts the client's thread.

tornadoclient Module

server Package

cherrypyserver Module

WebSocket within CherryPy is a tricky bit since CherryPy is a threaded server which would choke quickly if each thread of the server were kept attached to a long living connection that WebSocket expects.

In order to work around this constraint, we take some advantage of some internals of CherryPy as well as the introspection Python provides.

Basically, when the WebSocket handshake is complete, we take over the socket and let CherryPy take back the thread that was associated with the upgrade request.

These operations require a bit of work at various levels of the CherryPy framework but this module takes care of them and from your application's perspective, this is abstracted.

Here are the various utilities provided by this module:

- **WebSocketTool: The tool is in charge to perform the** HTTP upgrade and detach the socket from CherryPy. It runs at various hook points of the request's processing. Enable that tool at any path you wish to handle as a WebSocket handler.
- **WebSocketPlugin: The plugin tracks the instanciated web socket handlers.** It also cleans out websocket handler which connection have been closed down. The websocket connection then runs in its own thread that this plugin manages.

Simple usage example:

```
import cherrypy
from ws4py.server.cherrypyserver import WebSocketPlugin, WebSocketTool
from ws4py.websocket import EchoWebSocket

cherrypy.config.update({'server.socket_port': 9000})
WebSocketPlugin(cherrypy.engine).subscribe()
cherrypy.tools.websocket = WebSocketTool()

class Root(object):
    @cherrypy.expose
    def index(self):
        return 'some HTML with a websocket javascript connection'

    @cherrypy.expose
    def ws(self):
        pass

cherrypy.quickstart(Root(), '/', config={'/ws': {'tools.websocket.on': True,
                                                 'tools.websocket.handler_cls':␣
↪EchoWebSocket}})
```

Note that you can set the handler class on per-path basis, meaning you could also dynamically change the class based on other envrionmental settings (is the user authenticated for ex).

class ws4py.server.cherrypyserver.**WebSocketTool**

Bases: cherrypy._cptools.Tool

upgrade(*protocols=None*, *extensions=None*, *version=(8, 13)*, *handler_cls=<class 'ws4py.websocket.WebSocket'>*, *heartbeat_freq=None*)

Performs the upgrade of the connection to the WebSocket protocol.

The provided protocols may be a list of WebSocket protocols supported by the instance of the tool.

When no list is provided and no protocol is either during the upgrade, then the protocol parameter is not taken into account. On the other hand, if the protocol from the handshake isn't part of the provided list, the upgrade fails immediatly.

complete()
Sets some internal flags of CherryPy so that it doesn't close the socket down.

cleanup_headers()
Some clients aren't that smart when it comes to headers lookup.

start_handler()
Runs at the end of the request processing by calling the opened method of the handler.

class ws4py.server.cherrypyserver.**WebSocketPlugin**(*bus*)
Bases: cherrypy.process.plugins.SimplePlugin

start()

stop()

handle(*ws_handler*, *peer_addr*)
Tracks the provided handler.

Parameters

- **ws_handler** – websocket handler instance
- **peer_addr** – remote peer address for tracing purpose

cleanup()
Terminate all connections and clear the pool. Executed when the engine stops.

broadcast(*message*, *binary=False*)
Broadcasts a message to all connected clients known to the server.

Parameters

- **message** – a message suitable to pass to the send() method of the connected handler.
- **binary** – whether or not the message is a binary one

geventserver Module

wsgirefserver Module

Add WebSocket support to the built-in WSGI server provided by the wsgiref. This is clearly not meant to be a production server so please consider this only for testing purpose.

Mostly, this module overrides bits and pieces of the built-in classes so that it supports the WebSocket workflow.

```
from wsgiref.simple_server import make_server
from ws4py.websocket import EchoWebSocket
from ws4py.server.wsgirefserver import WSGIServer, WebSocketWSGIRequestHandler
from ws4py.server.wsgiutils import WebSocketWSGIApplication

server = make_server('', 9000, server_class=WSGIServer,
                     handler_class=WebSocketWSGIRequestHandler,
                     app=WebSocketWSGIApplication(handler_cls=EchoWebSocket))
server.initialize_websockets_manager()
server.serve_forever()
```

Note: For some reason this server may fail against autobahntestsuite.

class ws4py.server.wsgirefserver.**WebSocketWSGIHandler**(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Bases: wsgiref.handlers.SimpleHandler

setup_environ()
Setup the environ dictionary and add the *'ws4py.socket'* key. Its associated value is the real socket underlying socket.

finish_response()
Completes the response and performs the following tasks:

- Remove the *'ws4py.socket'* and *'ws4py.websocket'* environ keys.
- Attach the returned websocket, if any, to the WSGI server using its `link_websocket_to_server` method.

class ws4py.server.wsgirefserver.**WebSocketWSGIRequestHandler**(*request*, *client_address*, *server*)

Bases: wsgiref.simple_server.WSGIRequestHandler

class WebSocketWSGIHandler(*stdin*, *stdout*, *stderr*, *environ*, *multithread=True*, *multiprocess=False*)

Bases: wsgiref.handlers.SimpleHandler

finish_response()
Completes the response and performs the following tasks:
- Remove the *'ws4py.socket'* and *'ws4py.websocket'* environ keys.
- Attach the returned websocket, if any, to the WSGI server using its `link_websocket_to_server` method.

setup_environ()
Setup the environ dictionary and add the *'ws4py.socket'* key. Its associated value is the real socket underlying socket.

handle()
Unfortunately the base class forces us to override the whole method to actually provide our wsgi handler.

class ws4py.server.wsgirefserver.**WSGIServer**(*server_address*, *RequestHandlerClass*, *bind_and_activate=True*)

Bases: wsgiref.simple_server.WSGIServer

Constructor. May be extended, do not override.

initialize_websockets_manager()
Call thos to start the underlying websockets manager. Make sure to call it once your server is created.

shutdown_request(*request*)
The base class would close our socket if we didn't override it.

link_websocket_to_server(*ws*)
Call this from your WSGI handler when a websocket has been created.

server_close()
Properly initiate closing handshakes on all websockets when the WSGI server terminates.

wsgitutils Module

This module provides a WSGI application suitable for a WSGI server such as gevent or wsgiref for instance.

PEP 333 couldn't foresee a protocol such as WebSockets but luckily the way the initial protocol upgrade was designed means that we can fit the handshake in a WSGI flow.

The handshake validates the request against some internal or user-provided values and fails the request if the validation doesn't complete.

On success, the provided WebSocket subclass is instanciated and stored into the *'ws4py.websocket'* environ key so that the WSGI server can handle it.

The WSGI application returns an empty iterable since there is little value to return some content within the response to the handshake.

A server wishing to support WebSocket via ws4py should:

- Provide the real socket object to ws4py through the *'ws4py.socket'* environ key. We can't use *'wsgi.input'* as it may be wrapper to the socket we wouldn't know how to extract the socket from.
- Look for the *'ws4py.websocket'* key in the environ when the application has returned and probably attach it to a `ws4py.manager.WebSocketManager` instance so that the websocket runs its life.
- Remove the *'ws4py.websocket'* and *'ws4py.socket'* environ keys once the application has returned. No need for these keys to persist.
- Not close the underlying socket otherwise, well, your websocket will also shutdown.

Warning: The WSGI application sets the *'Upgrade'* header response as specified by RFC 6455. This is not tolerated by PEP 333 since it's a hop-by-hop header. We expect most servers won't mind.

class ws4py.server.wsgiutils.**WebSocketWSGIApplication**(*protocols=None*, *extensions=None*, *handler_cls=<class 'ws4py.websocket.WebSocket'>*)

Bases: `object`

WSGI application usable to complete the upgrade handshake by validating the requested protocols and extensions as well as the websocket version.

If the upgrade validates, the *handler_cls* class is instanciated and stored inside the WSGI *environ* under the *'ws4py.websocket'* key to make it available to the WSGI handler.

make_websocket(*sock*, *protocols*, *extensions*, *environ*)

Initialize the *handler_cls* instance with the given negociated sets of protocols and extensions as well as the *environ* and *sock*.

Stores then the instance in the *environ* dict under the *'ws4py.websocket'* key.

CHAPTER 5

Indices and tables

- modindex
- search

Python Module Index

W

Index

T

U

V

W