# WebSocket Programming

autobahn.readthedocs.io/en/latest/websocket/programming.html

This guide introduces WebSocket programming with **Autobahn**.

You'll see how to create WebSocket server ("Creating Servers") and client applications ("Creating Clients").

*Resources:*

- Example Code for this Guide: Twisted-based or asyncio-based
- More WebSocket Examples

## Creating Servers

Using **Autobahn** you can create WebSocket servers that will be able to talk to any (compliant) WebSocket client, including browsers.

We'll cover how to define the behavior of your WebSocket server by writing *protocol classes* and show some boilerplate for actually running a WebSocket server using the behavior defined in the server protocol.

### Server Protocols

To create a WebSocket server, you need to **write a protocol class to specify the behavior** of the server.

For example, here is a protocol class for a WebSocket echo server that will simply echo back any WebSocket message it receives:

```
class MyServerProtocol(WebSocketServerProtocol):

   def onMessage(self, payload, isBinary):
      ## echo back message verbatim
      self.sendMessage(payload, isBinary)
```

This is just three lines of code, but we will go through each one carefully, since writing protocol classes like above really is core to WebSocket programming using Autobahn.

The **first thing** to note is that you **derive** your protocol class from a base class provided by Autobahn. Depending on whether you write a Twisted or a asyncio based application, here are the base classes to derive from:

- `autobahn.twisted.websocket.WebSocketServerProtocol`
- `autobahn.asyncio.websocket.WebSocketServerProtocol`

So a Twisted-based echo protocol would import the base protocol from `autobahn.twisted.websocket` and derive from `autobahn.twisted.websocket.WebSocketServerProtocol`

*Twisted:*

```
from autobahn.twisted.websocket import WebSocketServerProtocol

class MyServerProtocol(WebSocketServerProtocol):

   def onMessage(self, payload, isBinary):
      ## echo back message verbatim
      self.sendMessage(payload, isBinary)
```

while an asyncio echo protocol would import the base protocol from `autobahn.asyncio.websocket` and derive from `autobahn.asyncio.websocket.WebSocketServerProtocol`

*asyncio:*

```
from autobahn.asyncio.websocket import WebSocketServerProtocol

class MyServerProtocol(WebSocketServerProtocol):

   def onMessage(self, payload, isBinary):
      ## echo back message verbatim
      self.sendMessage(payload, isBinary)
```

Note

In this example, only the imports differ between the Twisted and the asyncio variant. The rest of the code is identical. However, in most real world programs you probably won't be able to or don't want to avoid using network framework specific code.

---

## Receiving Messages

The **second thing** to note is that we **override a callback** `onMessage` which is called by Autobahn whenever the callback related event happens.

In case of `onMessage`, the callback will be called whenever a new WebSocket message was received. There are more WebSocket related callbacks, but for now the `onMessage` callback is all we need.

When our server receives a WebSocket message, the `autobahn.websocket.interfaces.IWebSocketChannel.onMessage()` will fire with the message `payload` received.

The `payload` is always a Python byte string. Since WebSocket is able to transmit **text** (UTF8) and **binary** payload, the actual payload type is signaled via the `isBinary` flag.

When the `payload` is **text** (`isBinary == False`), the bytes received will be an UTF8 encoded string. To process **text** payloads, the first thing you often will do is decoding the UTF8 payload into a Python string:

```
s = payload.decode('utf8')
```

Tip

You don't need to validate the bytes for actually being valid UTF8 - Autobahn does that already when receiving the message.

When using WebSocket text messages with JSON `payload`, typical code for receiving and decoding messages into Python objects that works on both Python 2 and 3 would look like this:

```
import json
obj = json.loads(payload.decode('utf8'))
```

We are using the Python standard JSON module `json`.

The `payload` (which is of type `bytes` on Python 3 and `str` on Python 2) is decoded from UTF8 into a native Python string, and then parsed from JSON into a native Python object.

---

## Sending Messages

The **third thing** to note is that we **use methods** like `sendMessage` provided by the base class to perform WebSocket related actions, like sending a WebSocket message.

As there are more methods for performing other actions (like closing the connection), we'll come back to this later, but for now, the `sendMessage` method is all we need.

`autobahn.websocket.interfaces.IWebSocketChannel.sendMessage()` takes the `payload` to send in a WebSocket message as Python bytes. Since WebSocket is able to transmit payloads of **text** (UTF8) and **binary** type, you need to tell Autobahn the actual type of the `payload` bytes. This is done using the `isBinary` flag.

Hence, to send a WebSocket text message, you will usually *encode* the payload to UTF8:

```
payload = s.encode('utf8')
self.sendMessage(payload, isBinary = False)
```

Warning

Autobahn will NOT validate the bytes of a text `payload` being sent for actually being valid UTF8. You MUST ensure that you only provide valid UTF8 when sending text messages. If you produce invalid UTF8, a conforming WebSocket peer will close the WebSocket connection due to the protocol violation.

When using WebSocket text messages with JSON `payload`, typical code for encoding and sending Python objects that works on both Python 2 and 3 would look like this:

```
import json
payload = json.dumps(obj, ensure_ascii = False).encode('utf8')
```

We are using the Python standard JSON module `json`.

The `ensure_ascii == False` option allows the JSON serializer to use Unicode strings. We can do this since we are encoding to UTF8 afterwards anyway. And UTF8 can represent the full Unicode character set.

---

## Running a Server

Now that we have defined the behavior of our WebSocket server in a protocol class, we need to actually start a server based on that behavior.

Doing so involves two steps:

1. Create a **Factory** for producing instances of our protocol class
2. Create a TCP **listening server** using the former Factory

Here is one way of doing that when using Twisted

*Twisted:*

```
if __name__ == '__main__':

   import sys

   from twisted.python import log
   from twisted.internet import reactor
   log.startLogging(sys.stdout)

   from autobahn.twisted.websocket import
WebSocketServerFactory
   factory = WebSocketServerFactory()
   factory.protocol = MyServerProtocol

   reactor.listenTCP(9000, factory)
   reactor.run()
```

What we are doing here is

1. Setup Twisted logging
2. Create a `autobahn.twisted.websocket.WebSocketServerFactory` and set our `MyServerProtocol` on the factory (the highlighted lines)
3. Start a server using the factory, listening on TCP port 9000

Similar, here is the asyncio way

*asyncio:*

```
if __name__ == '__main__':

   try:
      import asyncio
   except ImportError:
      ## Trollius >= 0.3 was renamed
      import trollius as asyncio

   from autobahn.asyncio.websocket import
WebSocketServerFactory
   factory = WebSocketServerFactory()
   factory.protocol = MyServerProtocol

   loop = asyncio.get_event_loop()
   coro = loop.create_server(factory, '127.0.0.1', 9000)
   server = loop.run_until_complete(coro)

   try:
      loop.run_forever()
   except KeyboardInterrupt:
      pass
   finally:
      server.close()
      loop.close()
```

What we are doing here is

1. Import asyncio, or the Trollius backport
2. Create a `autobahn.asyncio.websocket.WebSocketServerFactory` and set our `MyServerProtocol` on the factory (the highlighted lines)
3. Start a server using the factory, listening on TCP port 9000

Note

As can be seen, the boilerplate to create and run a server differ from Twisted, but the core code of creating a factory and setting our protocol (the highlighted lines) is identical (other than the differing import for the WebSocket factory).

You can find complete code for above examples here:

- WebSocket Echo (Twisted-based)
- WebSocket Echo (Asyncio-based)

## Connection Lifecycle

As we have seen above, Autobahn will fire *callbacks* on your protocol class whenever the event related to the respective callback occurs.

It is in these callbacks that you will implement application specific code.

The core WebSocket interface `autobahn.websocket.interfaces.IWebSocketChannel` provides the following *callbacks*:

- `autobahn.websocket.interfaces.IWebSocketChannel.onConnect()`
- `autobahn.websocket.interfaces.IWebSocketChannel.onOpen()`
- `autobahn.websocket.interfaces.IWebSocketChannel.onMessage()`
- `autobahn.websocket.interfaces.IWebSocketChannel.onClose()`

We have already seen the callback for Receiving Messages. This callback will usually fire many times during the lifetime of a WebSocket connection.

In contrast, the other three callbacks above each only fires once for a given connection.

## Opening Handshake

Whenever a new client connects to the server, a new protocol instance will be created and the `autobahn.websocket.interfaces.IWebSocketChannel.onConnect()` callback fires as soon as the WebSocket opening handshake is begun by the client.

For a WebSocket server protocol, `onConnect()` will fire with `autobahn.websocket.protocol.ConnectionRequest` providing information on the client wishing to connect via WebSocket.

```
class MyServerProtocol(WebSocketServerProtocol):

   def onConnect(self, request):
      print("Client connecting:
{}".format(request.peer))
```

On the other hand, for a WebSocket client protocol, `onConnect()` will fire with `autobahn.websocket.protocol.ConnectionResponse` providing information on the WebSocket connection that was accepted by the server.

```
class MyClientProtocol(WebSocketClientProtocol):

   def onConnect(self, response):
      print("Connected to Server:
{}".format(response.peer))
```

In this callback you can do things like

- checking or setting cookies or other HTTP headers
- verifying the client IP address
- checking the origin of the WebSocket request
- negotiate WebSocket subprotocols

For example, a WebSocket client might offer to speak several WebSocket subprotocols. The server can inspect the offered protocols in `onConnect()` via the supplied instance of `autobahn.websocket.protocol.ConnectionRequest`. When the server accepts the client, it'll chose one of the offered subprotocols. The client can then inspect the selected subprotocol in it's `onConnect()` callback in the supplied instance of `autobahn.websocket.protocol.ConnectionResponse`.

### Connection Open

The `autobahn.websocket.interfaces.IWebSocketChannel.onOpen()` callback fires when the WebSocket opening handshake has been successfully completed. You now can send and receive messages over the connection.

```
class MyProtocol(WebSocketProtocol):

   def onOpen(self):
      print("WebSocket connection
open.")
```

### Closing a Connection

The core WebSocket interface `autobahn.websocket.interfaces.IWebSocketChannel` provides the following *methods*:

- `autobahn.websocket.interfaces.IWebSocketChannel.sendMessage()`
- `autobahn.websocket.interfaces.IWebSocketChannel.sendClose()`

We've already seen one of above in Sending Messages.

The `autobahn.websocket.interfaces.IWebSocketChannel.sendClose()` will initiate a WebSocket closing handshake. After starting to close a WebSocket connection, no messages can be sent. Eventually, the `autobahn.websocket.interfaces.IWebSocketChannel.onClose()` callback will fire.

After a WebSocket connection has been closed, the protocol instance will get recycled. Should the client reconnect, a new protocol instance will be created and a new WebSocket opening handshake performed.

### Connection Close

When the WebSocket connection has closed, the `autobahn.websocket.interfaces.IWebSocketChannel.onClose()` callback fires.

```
class MyProtocol(WebSocketProtocol):

   def onClose(self, wasClean, code, reason):
      print("WebSocket connection closed:
{}".format(reason))
```

When the connection has closed, no messages will be received anymore and you cannot send messages also. The protocol instance won't be reused. It'll be garbage collected. When the client reconnects, a completely new protocol instance will be created.

## Creating Clients

Note

Creating WebSocket clients using **Autobahn** works very similar to creating WebSocket servers. Hence you should have read through Creating Servers first.

As with servers, the behavior of your WebSocket client is defined by writing a *protocol class*.

## Client Protocols

To create a WebSocket client, you need to write a protocol class to **specify the behavior** of the client.

For example, here is a protocol class for a WebSocket client that will send a WebSocket text message as soon as it is connected and log any WebSocket messages it receives:

```
class MyClientProtocol(WebSocketClientProtocol):

   def onOpen(self):
      self.sendMessage(u"Hello, world!".encode('utf8'))

   def onMessage(self, payload, isBinary):
      if isBinary:
         print("Binary message received: {0} bytes".format(len(payload)))
      else:
         print("Text message received:
{0}".format(payload.decode('utf8')))
```

Similar to WebSocket servers, you **derive** your WebSocket client protocol class from a base class provided by Autobahn. Depending on whether you write a Twisted or a asyncio based application, here are the base classes to derive from:

- `autobahn.twisted.websocket.WebSocketClientProtocol`
- `autobahn.asyncio.websocket.WebSocketClientProtocol`

So a Twisted-based protocol would import the base protocol from `autobahn.twisted.websocket` and derive from `autobahn.twisted.websocket.WebSocketClientProtocol`

*Twisted:*

```
from autobahn.twisted.websocket import WebSocketClientProtocol

class MyClientProtocol(WebSocketClientProtocol):

   def onOpen(self):
      self.sendMessage(u"Hello, world!".encode('utf8'))

   def onMessage(self, payload, isBinary):
      if isBinary:
         print("Binary message received: {0} bytes".format(len(payload)))
      else:
         print("Text message received:
{0}".format(payload.decode('utf8')))
```

while an asyncio-based protocol would import the base protocol from `autobahn.asyncio.websocket` and derive from `autobahn.asyncio.websocket.WebSocketClientProtocol`

*asyncio:*

```
from autobahn.asyncio.websocket import WebSocketClientProtocol

class MyClientProtocol(WebSocketClientProtocol):

   def onOpen(self):
      self.sendMessage(u"Hello, world!".encode('utf8'))

   def onMessage(self, payload, isBinary):
      if isBinary:
         print("Binary message received: {0} bytes".format(len(payload)))
      else:
         print("Text message received:
{0}".format(payload.decode('utf8')))
```

Note

In this example, only the imports differs between the Twisted and the asyncio variant. The rest of the code is identical. However, in most real world programs you probably won’t be able to or don’t want to avoid using network framework specific code.

---

Receiving and sending WebSocket messages as well as connection lifecycle in clients works exactly the same as with servers. Please see

- Receiving Messages
- Sending Messages
- Connection Lifecycle

### Running a Client

Now that we have defined the behavior of our WebSocket client in a protocol class, we need to actually start a client based on that behavior.

Doing so involves two steps:

1. Create a **Factory** for producing instances of our protocol class
2. Create a TCP **connecting client** using the former Factory

Here is one way of doing that when using Twisted

*Twisted:*

```
if __name__ == '__main__':

   import sys

   from twisted.python import log
   from twisted.internet import reactor
   log.startLogging(sys.stdout)

   from autobahn.twisted.websocket import
WebSocketClientFactory
   factory = WebSocketClientFactory()
   factory.protocol = MyClientProtocol

   reactor.connectTCP("127.0.0.1", 9000, factory)
   reactor.run()
```

What we are doing here is

1. Setup Twisted logging
2. Create a `autobahn.twisted.websocket.WebSocketClientFactory` and set our `MyClientProtocol` on the factory (the highlighted lines)
3. Start a client using the factory, connecting to localhost `127.0.0.1` on TCP port 9000

Similar, here is the asyncio way

*asyncio:*

```
if __name__ == '__main__':

   try:
      import asyncio
   except ImportError:
      ## Trollius >= 0.3 was renamed
      import trollius as asyncio

   from autobahn.asyncio.websocket import
WebSocketClientFactory
   factory = WebSocketClientFactory()
   factory.protocol = MyClientProtocol

   loop = asyncio.get_event_loop()
   coro = loop.create_connection(factory, '127.0.0.1', 9000)
   loop.run_until_complete(coro)
   loop.run_forever()
   loop.close()
```

What we are doing here is

1. Import asyncio, or the Trollius backport
2. Create a `autobahn.asyncio.websocket.WebSocketClientFactory` and set our `MyClientProtocol` on the factory (the highlighted lines)

3. Start a client using the factory, connecting to localhost `127.0.0.1` on TCP port 9000

Note

As can be seen, the boilerplate to create and run a client differ from Twisted, but the core code of creating a factory and setting our protocol (the highlighted lines) is identical (other than the differing import for the WebSocket factory).

You can find complete code for above examples here:

- WebSocket Echo (Twisted-based)
- WebSocket Echo (Asyncio-based)

## WebSocket Options

You can pass various options on both client and server side WebSockets; these are accomplished by calling `autobahn.websocket.WebSocketServerFactory.setProtocolOptions()` or `autobahn.websocket.WebSocketClientFactory.setProtocolOptions()` with keyword arguments for each option.

### Common Options (server and client)

> - *logOctets: if True, log every byte*
> - *logFrames: if True, log information about each frame*
> - *trackTimings: if True, enable debug timing code*
> - *utf8validateIncoming: if True (default), validate all incoming UTF8*
> - *applyMask: if True (default) apply mask to frames, when available*
> - *maxFramePayloadSize: if 0 (default), unlimited-sized frames allowed*
> - *maxMessagePayloadSize: if 0 (default), unlimited re-assembled payloads*
> - *autoFragmentSize: if 0 (default), don't fragment*
> - *failByDrop: if True (default), failed connections are terminated immediately*
> - *echoCloseCodeReason: if True, echo back the close reason/code*
> - *openHandshakeTimeout: timeout in seconds after which opening handshake will be failed (default: no timeout)*
> - *closeHandshakeTimeout: timeout in seconds after which close handshake will be failed (default: no timeout)*
> - *tcpNoDelay: if True (default), set NODELAY (Nagle) socket option*
> - *autoPingInterval: if set, seconds between auto-pings*
> - *autoPingTimeout: if set, seconds until a ping is considered timed-out*
> - *autoPingSize: bytes of random data to send in ping messages (between 4 [default] and 125)*

### Server-Only Options

- versions: what versions to claim support for (default 8, 13)

- webStatus: if True (default), show a web page if visiting this endpoint without an Upgrade header
- requireMaskedClientFrames: if True (default), client-to-server frames must be masked
- maskServerFrames: if True, server-to-client frames must be masked
- perMessageCompressionAccept: if provided, a single-argument callable
- serveFlashSocketPolicy: if True, server a flash policy file (default: False)
- flashSocketPolicy: the actual flash policy to serve (default one allows everything)
- allowedOrigins: a list of origins to allow, with embedded *s for wildcards; these are turned into regular expressions (e.g. *https://*.example.com:443* becomes *^https://.*.example.com:443$*). When doing the matching, the origin is **always** of the form *scheme://host:port* with an explicit port. By default, we match with * (that is, anything). To match all subdomains of *example.com* on any scheme and port, you'd need **://*.example.com:**
- maxConnections: total concurrent connections allowed (default 0, unlimited)
- trustXForwardedFor: number of trusted web servers (reverse proxies) in front of this server which set the X-Forwarded-For header

### Client-Only Options

- version: which version we are (default: 18)
- acceptMaskedServerFrames: if True, accept masked server-to-client frames (default False)
- maskClientFrames: if True (default), mask client-to-server frames
- serverConnectionDropTimeout: how long (in seconds) to wait for server to drop the connection when closing (default 1)
- perMessageCompressionOffers:
- perMessageCompressionAccept:

## Upgrading

### From < 0.7.0

Starting with release 0.7.0, **Autobahn** now supports both Twisted and asyncio as the underlying network library. This required renaming some modules.

Hence, code for Autobahn **< 0.7.0**

```
from autobahn.websocket import WebSocketServerProtocol
```

should be modified for Autobahn **>= 0.7.0** for (using Twisted)

```
from autobahn.twisted.websocket import WebSocketServerProtocol
```

or (using asyncio)

```
from autobahn.asyncio.websocket import WebSocketServerProtocol
```

Two more small changes:

1. The method `WebSocketProtocol.sendMessage` had parameter `binary` renamed to `isBinary` (for consistency with `onMessage`)
2. The `ConnectionRequest` object no longer provides `peerstr`, but only `peer`, and the latter is a plain, descriptive string (this was needed since we now support both Twisted and asyncio, and also non-TCP transports)