Using SSL

asyncio has built-in support for enabling SSL communication on sockets. Passing an SSLContext instance to the coroutines that create server or client connections enables the support and ensures that the SSL protocol setup is taken care of before the socket is presented as ready for the application to use.

The coroutine-based echo server and client from the previous section can be updated with a few small changes. The first step is to create the certificate and key files. A self-signed certificate can be created with a command like:

```
$ openssl req -newkey rsa:2048 -nodes -keyout pymotw.key \
-x509 -days 365 -out pymotw.crt
```

The openssl command will prompt for several values that are used to generate the certificate, and then produce the output files requested.

The insecure socket setup in the previous server example uses start_server() to create the listening socket.

```
factory = asyncio.start_server(echo, *SERVER_ADDRESS)
server = event_loop.run_until_complete(factory)
```

To add encryption, create an SSLContext with the certificate and key just generated and then pass the context to start_server().

```
# The certificate is created with pymotw.com as the hostname,
# which will not match when the example code runs elsewhere,
# so disable hostname verification.
ssl_context = ssl.create_default_context(ssl.Purpose.CLIENT_AUTH)
ssl_context.check_hostname = False
ssl_context.load_cert_chain('pymotw.crt', 'pymotw.key')

# Create the server and let the loop finish the coroutine before
# starting the real event loop.
factory = asyncio.start_server(echo, *SERVER_ADDRESS,
                               ssl=ssl_context)
```

Similar changes are needed in the client. The old version uses open_connection() to create the socket connected to the server.

```
    reader, writer = await asyncio.open_connection(*address)
```

An SSLContext is needed again to secure the client-side of the socket. Client identity is not being enforced, so only the certificate needs to be loaded.

```
    # The certificate is created with pymotw.com as the hostname,
    # which will not match when the example code runs
    # elsewhere, so disable hostname verification.
    ssl_context = ssl.create_default_context(
        ssl.Purpose.SERVER_AUTH,
    )
    ssl_context.check_hostname = False
    ssl_context.load_verify_locations('pymotw.crt')
    reader, writer = await asyncio.open_connection(
        *server_address, ssl=ssl_context)
```

One other small changes is needed in the client. Because the SSL connection does not support sending an end-of-file (EOF), the client uses a NULL byte as a message terminator instead.

The old version of the client send loop uses write_eof().

```
    # This could be writer.writelines() except that
    # would make it harder to show each part of the message
```

```
    # being sent.
    for msg in messages:
        writer.write(msg)
        log.debug('sending {!r}'.format(msg))
    if writer.can_write_eof():
        writer.write_eof()
    await writer.drain()
```

The new version sends a zero byte (`b'\x00'`).

```
    # This could be writer.writelines() except that
    # would make it harder to show each part of the message
    # being sent.
    for msg in messages:
        writer.write(msg)
        log.debug('sending {!r}'.format(msg))
    # SSL does not support EOF, so send a null byte to indicate
    # the end of the message.
    writer.write(b'\x00')
    await writer.drain()
```

The `echo()` coroutine in the server must look for the NULL byte and close the client connection when it is received.

```
async def echo(reader, writer):
    address = writer.get_extra_info('peername')
    log = logging.getLogger('echo_{}_{}'.format(*address))
    log.debug('connection accepted')
    while True:
        data = await reader.read(128)
        terminate = data.endswith(b'\x00')
        data = data.rstrip(b'\x00')
        if data:
            log.debug('received {!r}'.format(data))
            writer.write(data)
            await writer.drain()
            log.debug('sent {!r}'.format(data))
        if not data or terminate:
            log.debug('message terminated, closing connection')
            writer.close()
            return
```

Running the server in one window, and the client in another, produces this output.

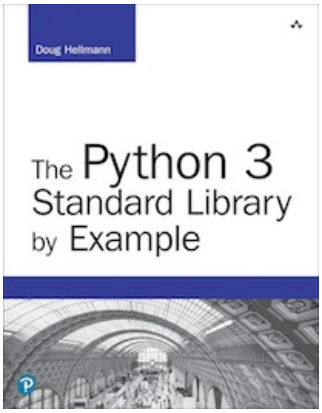
```
$ python3 asyncio_echo_server_ssl.py
asyncio: Using selector: KqueueSelector
main: starting up on localhost port 10000
echo_::1_53957: connection accepted
echo_::1_53957: received b'This is the message. '
echo_::1_53957: sent b'This is the message. '
echo_::1_53957: received b'It will be sent in parts.'
echo_::1_53957: sent b'It will be sent in parts.'
echo_::1_53957: message terminated, closing connection
```

```
$ python3 asyncio_echo_client_ssl.py
asyncio: Using selector: KqueueSelector
echo_client: connecting to localhost port 10000
echo_client: sending b'This is the message. '
echo_client: sending b'It will be sent '
echo_client: sending b'in parts.'
echo_client: waiting for response
echo_client: received b'This is the message. '
echo_client: received b'It will be sent in parts.'
echo_client: closing
main: closing event loop
```

Asynchronous I/O Using Coroutines and Streams

Interacting with Domain Name Services

This page was last updated 2016-12-23.



Get the book

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?