# socketserver — Creating Network Servers

**Purpose:** Creating network servers.

The `socketserver` module is a framework for creating network servers. It defines classes for handling synchronous network requests (the server request handler blocks until the request is completed) over TCP, UDP, Unix streams, and Unix datagrams. It also provides mix-in classes for easily converting servers to use a separate thread or process for each request.

Responsibility for processing a request is split between a server class and a request handler class. The server deals with the communication issues, such as listening on a socket and accepting connections, and the request handler deals with the "protocol" issues like interpreting incoming data, processing it, and sending data back to the client. This division of responsibility means that many applications can use one of the existing server classes without any modifications, and provide a request handler class for it to work with the custom protocol.

## Server Types

There are five different server classes defined in `socketserver`. `BaseServer` defines the API, and is not intended to be instantiated and used directly. `TCPServer` uses TCP/IP sockets to communicate. `UDPServer` uses datagram sockets. `UnixStreamServer` and `UnixDatagramServer` use Unix-domain sockets and are only available on Unix platforms.

## Server Objects

To construct a server, pass it an address on which to listen for requests and a request handler *class* (not instance). The address format depends on the server type and the socket family used. Refer to the `socket` module documentation for details.

Once the server object is instantiated, use either `handle_request()` or `serve_forever()` to process requests. The `serve_forever()` method calls `handle_request()` in an infinite loop, but if an application needs to integrate the server with another event loop or use `select()` to monitor several sockets for different servers, it can call `handle_request()` directly.

## Implementing a Server

When creating a server, it is usually sufficient to reuse one of the existing classes and provide a custom request handler class. For other cases, `BaseServer` includes several methods that can be overridden in a subclass.

- `verify_request(request, client_address)`: Return `True` to process the request or `False` to ignore it. For example, a server could refuse requests from an IP range or if it is overloaded.
- `process_request(request, client_address)`: Calls `finish_request()` to actually do the work of handling the request. It can also create a separate thread or process, as the mix-in classes do.
- `finish_request(request, client_address)`: Creates a request handler instance using the class given to the server's constructor. Calls `handle()` on the request handler to process the request.

## Request Handlers

Request handlers do most of the work of receiving incoming requests and deciding what action to take. The handler is responsible for implementing the protocol on top of the socket layer (i.e., HTTP, XML-RPC, or AMQP). The request handler reads the request from the incoming data channel, processes it, and writes a response back out. There are three methods available to be over-ridden.

- `setup()`: Prepares the request handler for the request. In the `StreamRequestHandler` the `setup()` method creates file-like objects for reading from and writing to the socket.
- `handle()`: Does the real work for the request. Parse the incoming request, process the data, and send a response.
- `finish()`: Cleans up anything created during `setup()`.

Many handlers can be implemented with only a `handle()` method.

## Echo Example

This example implements a simple server/request handler pair that accepts TCP connections and echos back any data sent by the client. It starts with the request handler.

```
# socketserver_echo.py
```

```
import logging
import sys
import socketserver

logging.basicConfig(level=logging.DEBUG,
                    format='%(name)s: %(message)s',
                    )


class EchoRequestHandler(socketserver.BaseRequestHandler):

    def __init__(self, request, client_address, server):
        self.logger = logging.getLogger('EchoRequestHandler')
        self.logger.debug('__init__')
        socketserver.BaseRequestHandler.__init__(self, request,
                                                 client_address,
                                                 server)
        return

    def setup(self):
        self.logger.debug('setup')
        return socketserver.BaseRequestHandler.setup(self)

    def handle(self):
        self.logger.debug('handle')

        # Echo the back to the client
        data = self.request.recv(1024)
        self.logger.debug('recv()->"%s"', data)
        self.request.send(data)
        return

    def finish(self):
        self.logger.debug('finish')
        return socketserver.BaseRequestHandler.finish(self)
```

The only method that actually needs to be implemented is `EchoRequestHandler.handle()`, but versions of all of the methods described earlier are included to illustrate the sequence of calls made. The `EchoServer` class does nothing different from `TCPServer`, except log when each method is called.

```
class EchoServer(socketserver.TCPServer):

    def __init__(self, server_address,
                 handler_class=EchoRequestHandler,
                 ):
        self.logger = logging.getLogger('EchoServer')
        self.logger.debug('__init__')
        socketserver.TCPServer.__init__(self, server_address,
                                        handler_class)
        return

    def server_activate(self):
        self.logger.debug('server_activate')
        socketserver.TCPServer.server_activate(self)
        return

    def serve_forever(self, poll_interval=0.5):
        self.logger.debug('waiting for request')
        self.logger.info(
            'Handling requests, press <Ctrl-C> to quit'
        )
        socketserver.TCPServer.serve_forever(self, poll_interval)
        return

    def handle_request(self):
        self.logger.debug('handle_request')
        return socketserver.TCPServer.handle_request(self)

    def verify_request(self, request, client_address):
        self.logger.debug('verify_request(%s, %s)',
                          request, client_address)
```

```
        return socketserver.TCPServer.verify_request(
            self, request, client_address,
        )

    def process_request(self, request, client_address):
        self.logger.debug('process_request(%s, %s)',
                          request, client_address)
        return socketserver.TCPServer.process_request(
            self, request, client_address,
        )

    def server_close(self):
        self.logger.debug('server_close')
        return socketserver.TCPServer.server_close(self)

    def finish_request(self, request, client_address):
        self.logger.debug('finish_request(%s, %s)',
                          request, client_address)
        return socketserver.TCPServer.finish_request(
            self, request, client_address,
        )

    def close_request(self, request_address):
        self.logger.debug('close_request(%s)', request_address)
        return socketserver.TCPServer.close_request(
            self, request_address,
        )

    def shutdown(self):
        self.logger.debug('shutdown()')
        return socketserver.TCPServer.shutdown(self)
```

The last step is to add a main program that sets up the server to run in a thread, and sends it data to illustrate which methods are called as the data is echoed back.

```
if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0)  # let the kernel assign a port
    server = EchoServer(address, EchoRequestHandler)
    ip, port = server.server_address  # what port was assigned?

    # Start the server in a thread
    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)  # don't hang on exit
    t.start()

    logger = logging.getLogger('client')
    logger.info('Server on %s:%s', ip, port)

    # Connect to the server
    logger.debug('creating socket')
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    logger.debug('connecting to server')
    s.connect((ip, port))

    # Send the data
    message = 'Hello, world'.encode()
    logger.debug('sending data: %r', message)
    len_sent = s.send(message)

    # Receive a response
    logger.debug('waiting for response')
    response = s.recv(len_sent)
    logger.debug('response from server: %r', response)

    # Clean up
    server.shutdown()
    logger.debug('closing socket')
    s.close()
```

```
    logger.debug('done')
    server.socket.close()
```

Running the program produces the following output.

```
$ python3 socketserver_echo.py

EchoServer: __init__
EchoServer: server_activate
EchoServer: waiting for request
EchoServer: Handling requests, press <Ctrl-C> to quit
client: Server on 127.0.0.1:55484
client: creating socket
client: connecting to server
client: sending data: b'Hello, world'
EchoServer: verify_request(<socket.socket fd=7, family=AddressFamily
.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1',
55484), raddr=('127.0.0.1', 55485)>, ('127.0.0.1', 55485))
EchoServer: process_request(<socket.socket fd=7, family=AddressFamil
y.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1',
 55484), raddr=('127.0.0.1', 55485)>, ('127.0.0.1', 55485))
EchoServer: finish_request(<socket.socket fd=7, family=AddressFamily
.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1',
55484), raddr=('127.0.0.1', 55485)>, ('127.0.0.1', 55485))
EchoRequestHandler: __init__
EchoRequestHandler: setup
EchoRequestHandler: handle
client: waiting for response
EchoRequestHandler: recv()->"b'Hello, world'"
EchoRequestHandler: finish
client: response from server: b'Hello, world'
EchoServer: shutdown()
EchoServer: close_request(<socket.socket fd=7, family=AddressFamily.
AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 5
5484), raddr=('127.0.0.1', 55485)>)
client: closing socket
client: done
```

### Note

The port number used will change each time the program runs because the kernel allocates an available port automatically. To make the server listen on a specific port each time, provide that number in the address tuple instead of the `0`.

Here is a condensed version of the same server, without the logging calls. Only the `handle()` method in the request handler class needs to be provided.

```
# socketserver_echo_simple.py

import socketserver


class EchoRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        # Echo the back to the client
        data = self.request.recv(1024)
        self.request.send(data)
        return


if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0)  # let the kernel assign a port
    server = socketserver.TCPServer(address, EchoRequestHandler)
    ip, port = server.server_address  # what port was assigned?
```

```
    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)  # don't hang on exit
    t.start()

    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Send the data
    message = 'Hello, world'.encode()
    print('Sending : {!r}'.format(message))
    len_sent = s.send(message)

    # Receive a response
    response = s.recv(len_sent)
    print('Received: {!r}'.format(response))

    # Clean up
    server.shutdown()
    s.close()
    server.socket.close()
```

In this case, no special server class is required since the `TCPServer` handles all of the server requirements.

```
$ python3 socketserver_echo_simple.py

Sending : b'Hello, world'
Received: b'Hello, world'
```

## Threading and Forking

To add threading or forking support to a server, include the appropriate mix-in in the class hierarchy for the server. The mix-in classes override `process_request()` to start a new thread or process when a request is ready to be handled, and the work is done in the new child.

For threads, use `ThreadingMixIn`.

```
# socketserver_threaded.py

import threading
import socketserver


class ThreadedEchoRequestHandler(
        socketserver.BaseRequestHandler,
):

    def handle(self):
        # Echo the back to the client
        data = self.request.recv(1024)
        cur_thread = threading.currentThread()
        response = b'%s: %s' % (cur_thread.getName().encode(),
                                data)
        self.request.send(response)
        return


class ThreadedEchoServer(socketserver.ThreadingMixIn,
                         socketserver.TCPServer,
                         ):
    pass


if __name__ == '__main__':
    import socket

    address = ('localhost', 0)  # let the kernel assign a port
    server = ThreadedEchoServer(address,
                                ThreadedEchoRequestHandler)
    ip, port = server.server_address  # what port was assigned?
```

```
    ip, port = server.server_address  # what port was assigned?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)  # don't hang on exit
    t.start()
    print('Server loop running in thread:', t.getName())

    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Send the data
    message = b'Hello, world'
    print('Sending : {!r}'.format(message))
    len_sent = s.send(message)

    # Receive a response
    response = s.recv(1024)
    print('Received: {!r}'.format(response))

    # Clean up
    server.shutdown()
    s.close()
    server.socket.close()
```

The response from this threaded server includes the identifier of the thread where the request is handled.

```
$ python3 socketserver_threaded.py

Server loop running in thread: Thread-1
Sending : b'Hello, world'
Received: b'Thread-2: Hello, world'
```

For separate processes, use the `ForkingMixIn`.

```
# socketserver_forking.py

import os
import socketserver


class ForkingEchoRequestHandler(socketserver.BaseRequestHandler):

    def handle(self):
        # Echo the back to the client
        data = self.request.recv(1024)
        cur_pid = os.getpid()
        response = b'%d: %s' % (cur_pid, data)
        self.request.send(response)
        return


class ForkingEchoServer(socketserver.ForkingMixIn,
                        socketserver.TCPServer,
                        ):
    pass


if __name__ == '__main__':
    import socket
    import threading

    address = ('localhost', 0)  # let the kernel assign a port
    server = ForkingEchoServer(address,
                               ForkingEchoRequestHandler)
    ip, port = server.server_address  # what port was assigned?

    t = threading.Thread(target=server.serve_forever)
    t.setDaemon(True)  # don't hang on exit
    t.start()
    print('Server loop running in process:', os.getpid())
```

```
    # Connect to the server
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((ip, port))

    # Send the data
    message = 'Hello, world'.encode()
    print('Sending : {!r}'.format(message))
    len_sent = s.send(message)

    # Receive a response
    response = s.recv(1024)
    print('Received: {!r}'.format(response))

    # Clean up
    server.shutdown()
    s.close()
    server.socket.close()
```

In this case, the process ID of the child is included in the response from the server:

```
$ python3 socketserver_forking.py

Server loop running in process: 22599
Sending : b'Hello, world'
Received: b'22600: Hello, world'
```

## See also

- Standard library documentation for socketserver
- `socket` – Low-level network communication
- `select` – Low-level asynchronous I/O tools
- `asyncio` – Asynchronous I/O, event loop, and concurrency tools
- `SimpleXMLRPCServer` – XML-RPC server built using `socketserver`.
- *Unix Network Programming, Volume 1: The Sockets Networking API, 3/E* By W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. Published by Addison-Wesley Professional, 2004. ISBN-10: 0131411551
- *Foundations of Python Network Programminng, 3/E* By Brandon Rhodes and John Goerzen. Published by Apress, 2014. ISBN-10: 1430258543

select — Wait for I/O Efficiently

The Internet

**Quick Links**

Server Types
Server Objects
Implementing a Server
Request Handlers
Echo Example
Threading and Forking

*This page was last updated 2016-12-18.*

**Navigation**

select — Wait for I/O Efficiently
The Internet



*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?*

**This Site**

Module Index
Index



**Other Writing**

Blog
The Python Standard Library By Example