Earwax-server

Chris Norman

Nov 16, 2020

CONTENTS:

CHAPTER

ONE

EARWAX_SERVER

1.1 earwax_server package

1.1.1 Module contents

A lightweight and event driven server framework.

This module is designed for creating servers (particularly for games) with minimal code.

Using a Pyglet-style event framework, you can create servers quickly and efficiently:

```
from earwax_server import Server

s = Server()
s.run(1234)
```

The above code creates a very minimal server. This server does nothing, since any data sent to it simply disappears.

You can verify it is working by enabling logging, and watching for incoming connections:

```
import logging
from earwax_server import Server
logging.basicConfig(level='INFO')
s = Server()
s.run(1234)
```

You can connect to the running instance with telnet.

To get the sent data, provide a handler for the *Server.on_data* event:

```
@s.event
def on_data(ctx, data) -> None:
    print(data)
```

The provided data will be a bytes-like object.

There are of course events which are dispatched when a connection is made, and when a connection disconnects.

There is even a rudimentary way of blocking connections, by subclassing *Server*, and overriding the *Server.can_connect()* method.

class earwax_server.**ConnectionContext**(*socket: gevent._socket3.socket*, *address: Tuple[str, int, int, int]*)

Bases: object

A context for holding connection information.

An instances of this class is created every time a new connection is made to a *Server* instance. As such, contexts are used a lot when dispatching events.

Variables

- ***socket*** – The socket that this context represents.
- ***address*** – The address that the socket is connected from.
- ***hostname*** – The hostname of the remote client.
- ***port*** – The port that the socket is connected on.
- ***logger*** – A logger for this context.

  The logger will already have a name constructed from *hostname*, and *port*.

address: Tuple[str, int, int, int]

disconnect() → None
Disconnect this context.

Disconnects the underlying *socket*.

hostname: str

logger: logging.Logger

port: int

send_bytes(*buf: bytes*, *encoding: Optional[str] = None*) → None
Send an encoded string to this context.

Sendds a bytes-like object to *self.socket*.

The string will have `'\r\n'` appended to it.

Parameters

- **buf** – The bytes-like object to send.

  This value must have already been encoded.
- **encoding** – The value to use for encoding the line terminator.

  If not specified, the system default encoding will be used.

send_raw(*data: bytes*) → None
Send data to this context.

Sends raw data to *self.socket*.

Parameters **data** – The data to send.

send_string(*string: str*) → None
Send an unencoded string to this context.

Sends the string to *self.socket*.

The string is automatically encoded to a bytes-like object, and `'\r\n'` is appended.

Parameters **string** – The string to send (minus the end of line terminator).

This value must be an unencoded string.

socket: gevent._socket3.socket

exception earwax_server.**EventNameError**
Bases: Exception

There was a problem with an event name.

class earwax_server.**Server**
Bases: object

A server instance.

By attaching event handlers to instances of this class, you can build servers with very little code.

When you have attached all the events, use the *run()* method to start listening for connections.

Variables

- ***connections*** – Every context that is connected to this server.
- ***stream_server*** – The underlying gevent server.

can_connect(*ctx:* earwax_server.ConnectionContext) → bool
Determine if a context can connect or not.

Return True if the connection is allowed, False otherwise.

Parameters **ctx** – The context that is trying to connect.

connections: List[*earwax_server.ConnectionContext*]

dispatch_event(*name: str*, **args*, ***kwargs*) → None
Dispatch an event.

If the given name has not been registered with the *Server.register_event_type()* method, then *EventNameError* will be raised.

Parameters

- **name** – The name of the event type to dispatch.
- **args** – The positional arguments to be passed to the event handlers.
- **kwargs** – The keyword arguments to pass to the event handlers.

event(*value: Union[Callable[[...], Optional[bool]], str]*) → Union[Callable[[...], Optional[bool]], Callable[[Callable[[...], Optional[bool]]], Callable[[...], Optional[bool]]]]
Register a new event.

The new event handler will be prepended to the event handlers list, thus allowing newer event handlers to override older ones.

When the *Server.dispatch_event()* is used, the list of handlers will be iterated over, and each handler executed.

If a handler returns EVENT_HANDLED, execution ends.

If the provided event name (see below) is not a recognised event type, then *EventNameError* will be raised.

Parameters **value** – Either the name of an event type this handler should listen to, or an event handler.

If value is a string, then it will be considered the name of an event type, and a callable will be returned so this method can be used as a decorator.

If value is a callable, then it is assumed to be a handler function, and its __name__ attribute is used as the name. In this case, the handler function is returned directly.

handle (*socket: gevent._socket3.socket*, *address: Tuple[str, int, int, int]*) → None
Deal with new connections.

This function is used with *self.stream_server*.

Parameters

- **socket** – The socket that has just connected.
- **address** – The address of the new conection.

on_block (*ctx:* earwax_server.ConnectionContext) → None
Handle a blocked connection.

This event is dispatched when an address has been blocked.

Parameters **ctx** – The connection context that has been blocked.

on_connect (*ctx:* earwax_server.ConnectionContext) → None
Deal with new connections.

This event is dispatched when a new connection is established.

By the time this event is dispatched, it has already been established by the *can_connect()* method that this address is allowed to connect.

Parameters **ctx** – The context that has connected.

on_data (*ctx:* earwax_server.ConnectionContext, *data: bytes*) → None
Handle incoming data.

This event is dispatched when data is received over a connection.

Parameters

- **ctx** – The originating connection context.
- **data** – The data which has been received.

  This value will be unchanged from when it was received. As such, no decoding will have yet been performed, hence why a bytes object is passed, rather than a string.

on_disconnect (*ctx:* earwax_server.ConnectionContext) → None
Deal with disconnections.

This event is dispatched when a connection is closed.

Parameters **ctx** – The context that is disconnecting.

register_event_type (*name: str*) → str
Register a new event type.

The name of the new event type will be returned.

If the name already exists, *EventNameError* will be raised.

Parameters **name** – The name of the new type.

run (*port: int*, *host: str = ''*, ***kwargs*) → None
Start the server running.

Set *self.stream_server* to an instance of `gevent.server.StreamServer`, and call its `serve_forever` method.

All extra keyword arguments are passed to the constructor of `StreamServer`.

Parameters

- **port** – The port to listen on.
- **host** – The interface to listen on.

stream_server: Optional[gevent.server.StreamServer]

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX