Networking with Node

In this lab, you will learn to use Node’s low-level networking capabilities to create a simple TCP echo server and interact with it from a network client.

# Objectives

In this lab, you will learn to

* use various server & client functions of the net module,
* use the net.Server, net.Socket, and Buffer objects, and
* write asynchronous callback functions to handle events from sockets.

# Write the server logic

We'll jump right into writing the server.

## Create a socket server

1. Create & store in a variable called server a new socket server via net's createServer factory function. For its callback, use the function onClientConnected, whose stub has already been provided for you.

## Start listening on a port

2. Tell the server to start listening on port PORT, and provide the function onServerPortBound as the callback, whose full implementation has also already been provided for you.

## Test the server

At this point, believe it or not, we have a functioning TCP server that can listen on port PORT, and accept connections & handle disconnections.

3. Execute server.js using node. You should see a message that looks something like the following.

$ node server.js

server listening on port 1986

4. Open another terminal window and enter the command telnet localhost 1986. You should see output similar to the following.

$ telnet localhost 1986

Trying 127.0.0.1...

Connected to localhost.

Escape character is '^]'.

Kill the server (via Ctrl-C if in a terminal) and you should see the client terminal echo something similar to the following.

Connection closed by foreign host.

When you are confident that the server is running and you're seeing these messages, move on to the next step.

## Start tracking client connections

We're now going to start adding logic to our onClientConnected function to keep track of the connections that this server has made.

5. First, push the socket given to the callback onto the sockets array for later use.

## Add a disconnection handler

6. Now, add a listener to the disconnection event of the socket passed into onClientConnected; the event is 'end' and the callback function we'll use is an inline function called onEnd that delegates to onSocketEnd(socket).

To add a listener to any EventEmitter (like net.Socket), use EventEmitter's on method, which takes the event of notification and the callback function.

The inline function that we created is also a closure, since it uses the socket given to the onClientConnected callback.

## Test the server

Now that we've coded a little more, let's test a little more.

7. Run server.js via node again.

8. Once you see the message that the server is listening, connect to it from another terminal again.

9. Now that the client is connected, type Ctrl-] (the close-square-bracket key, or whatever your terminal states the escape character is) to escape to the telnet prompt, then type quit and enter. The client terminal should look something like the following.

$ telnet localhost 1986

Trying 127.0.0.1...

Connected to localhost.

Escape character is '^]'.

^]

telnet> quit

Connection closed.

10. Confirm that your client has closed the connection and that you see the server's disconnection message. The server's messages should look something like the following.

$ node server.js

server listening on port 1986

client connected

client disconnected

Next, let's write some usage information to the client so that the client knows how to interact with the server.

11. Use the following code at the end of the onClientConnected function to write the usage message to the client.

socket.write('Hello. Usage:\r\n'

+ '/name:yourName to tell me your name\r\n'

+ '/quit to quit\r\n'

+ '/shutdown to shutdown the server\r\n'

+ "otherwise I'll just echo 'You said: ' plus what you entered!\r\n");

Now, let's add logic to handle the behavior that the usage message describes.

## Add a data listener

12. Right after the line(s) in onClientConnected that register the 'end' listener, add another listener for the 'data' event, this time creating an inline function called onData that takes a single argument named data and that delegates to onSocketData(data, socket), another function whose stub is provided for you.

The 'data' event from a socket passes in data as an instance of Node's Buffer object, unless the stream's setEncoding was previously called with a valid encoding string, in which case data is given as a string with the desired encoding.

We're now going to complete the implementation of the behavior described in the usage that’s given to clients right after they connect, namely that any string is echoed back to the client with the prefix 'You said: ', except for '/quit', which closes the client connection, '/shutdown', which shuts down the server completely, or '/name:yourName', which causes the server to prefix echoed strings with '<yourName> said: '. All of these edits will be going into the function onSocketData.

13. Start by setting data to data.toString(), just to make sure it's a string.

14. Add a check to see if the string is literally '/shutdown'. If it is, invoke shutdown() and return. We'll flesh out the function shutdown later.

15. Add a check to see if the string is literally '/quit'. If it is, invoke shutdown(socket). Again, we'll flesh out the function shutdown later.

16. Next, see if the string begins with '/name:'. If it does, then parse the string following the colon, and save it as a new property, username, on socket.

You can use String's indexOf method to see if a string begins with another string.

Remember that in JavaScript, you can add arbitrary properties to any object. Our requirement is to track the username for each client independently. Since each socket already uniquely identifies a client, we can just add a username property to the socket and set its value to the name the user gave.

17. Write to the socket a message that the server will now call the user by their new name and return.

18. If none of the prior conditions were met, then simply write to the socket '<name> said: ', then the string they entered, where <name> is the current value of socket.username.

19. Lastly, give socket.username a default value of 'You' at the end of the onClientConnected function.

## Test the server

20. Now that we’ve coded a little more, let's test a little more. Run the server, connect with a telnet client again, and make sure that messages are being echoed, that '/name:', '/quit', and '/shutdown' messages are being handled correctly.

Once your server is behaving properly, move on to the next step.

## Implement behavior for /quit and /shutdown messages

21. Now, add logic to the shutdown function to handle the '/quit' and '/shutdown' messages.

In function shutdown, if socket is truey (basically, not null or undefined), find socket in the sockets array, call its end() method, then remove it from the sockets array and return.

You can use Array's indexOf method to find the index of the element you're looking for. If it's not found, indexOf will return -1.

If socket is falsey (basically, null or undefined), then the shutdown function is supposed to interpret the call as a shutdown of the entire server, including all client connections (which is, of course, something that you probably would *not* want to do in a real server application). First, cause the server to stop accepting new connections via Server's close method and give it an inline function called onClosed that simply logs to the console that the server has been closed. Then, loop through the sockets array, telling each socket to end(). Make sure to stop referencing each socket object, after you've closed it!

Note that server.close() does not actually shutdown the server and close all existing client connections. It merely stops the server from accepting new connections; you must still close any open connections the server has.

## Test the server

22. By now, you know the deal: code a little, test a little. Make sure the server now closes the client connection on '/quit' messages and shuts down the entire server on '/shutdown' messages.

Once your server is behaving properly, move on to the next step.

## Handle expected server errors

The last step in this lab will be to add some reasonable error handling in the server.

23. Add a server error listener just before the line where you invoke server.listen. The event we want to listen for is 'error', and our callback can be written inline with the name onError and a single parameter for the Error object.

The only error that we're going to handle in this lab is when the port is already in use by some other process. In that case, the error's code property will be 'EADDRINUSE'. Check for that condition, and if it's met, invoke server.listen again after a one second timeout and return. Otherwise, we'll just let any other error crash the server's initialization and log the error to the console.

## Test the server

24. Run one instance of the server in order to occupy the port; make sure it successfully starts up.

25. Now, run another instance of the server in another terminal and make sure that the onError handler is getting called and issuing a deferred server.listen call each second.

26. After a few failed attempts by the second server to start, kill the first server (however you want to). This will free the port, allowing the next invocation of server.listen in the second server instance to succeed.

When you see the second server process running successfully, you have completed the lab!