# WebSocket communications

This chapter explores bidirectional communications using WebSockets.

### Lesson 1: Communicating by using WebSocket

**Recap**: Long polling is where the client asks the server for information, and the server keeps the connection open until information is found.

**WebSocket**: provides a standard way for the server to send messages to the client without first being asked by the client, and vice versa

### Defining the WebSocket API

At the heart of the WebSocket API is the WebSocket Object, which is defined on the browser's window object. *You can easily test if the WebSocket Object exists to determine whether a browser supports WebSockets or not.*

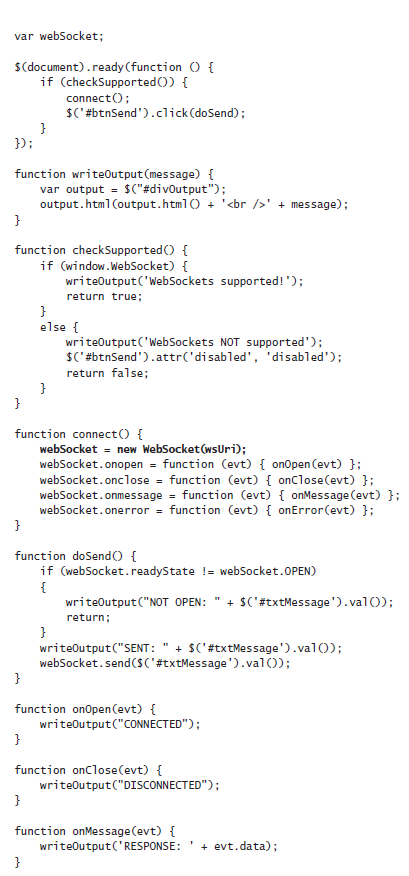
The WebSocket object contains the following members:

* **WebSocket** constructor A method that requires a URL argument and can optionally accept additional parameters to define the sub-protocol that you’ll use, such as chat or rpc. The client and the server are typically matched to use the same protocol.
* **close** A method that closes WebSocket.
* **send** A method that sends data to the server.
* **binaryType** A property that indicates the binary data format the onmessage event receives.
* **bufferedAmount** A property containing the number of data bytes queued using the send method.
* **extensions** A property that indicates the extensions the server selected.
* **onclose** An event property that’s called when the socket is closed.
* **onerror** An event property that’s called when there is an error.
* **onmessage** An event property that’s called when a message is received.
* **onopen** An event property that’s called when WebSocket establishes a connection.
* **protocol** A property that indicates the protocol that the server selected.
* **readyState** A property that indicates the state of the WebSocket connection.
* **url** A property that indicates the current URL of the WebSocket object.

### Implementing the WebSocket object

* WebSocket protocol communications typically use TCP port number 80
* The WebSocket URL begins with ws:// or wss:// for secure WebSocket protocol

The example before shows how to create a WebSocket object and configure its onopen, onerror, and onclose events, as well as calling the send method to send message, and the onmessage event triggers if there is a response.





The example above does the following:

* The connect function instantiates WebSocket.
* The constructor accepts a URI argument. Creating the WebSocket object automatically initiates communications to the URI to attempt to open the connection asynchronously.
* The connect function also subscribes to the onopen, onclose, onmessage, and onerror events.
* **NOTE**: It’s important to subscribe to these events imme-diately because the connection might open quickly, and you want to ensure that you are subscribed to the onopen event as soon as possible so you don’t miss the event.
* The doSend function checks the readyState property of the WebSocket object, then sends a message to the server if the connection is OPEN. (The readyState properties include: **CONNECTING = 0, OPEN = 1, CLOSING = 2, CLOSED = 3**)
* When a message is received from the server, the onMessage function is called with the event object passed
* When an error is received, the onError function is called with the event object passed

For an example showcasing the above, go to https://jsfiddle.net/9gxLLqr2/1/

### Dealing with timeouts

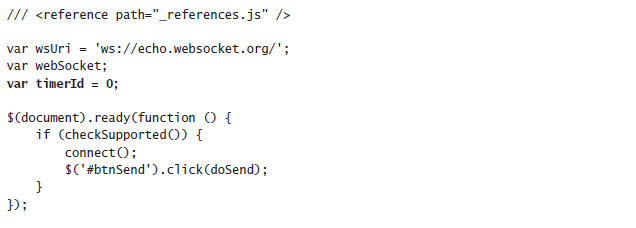
Reasons why timeouts occur:

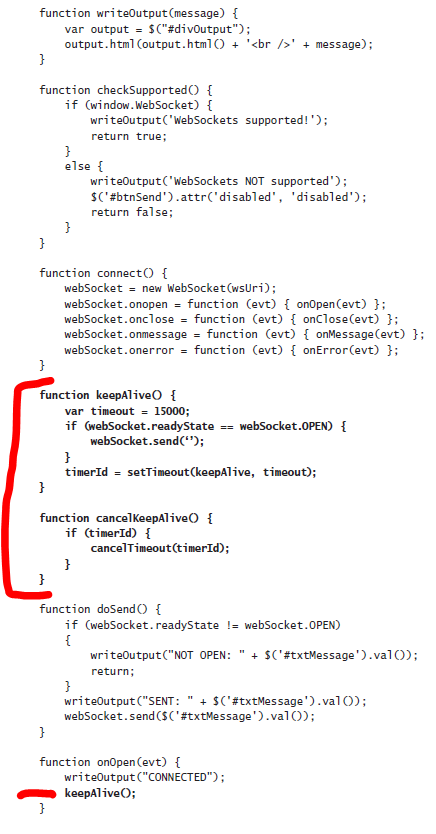
* Firewalls between your browser and the server
* a long period of application inactivity

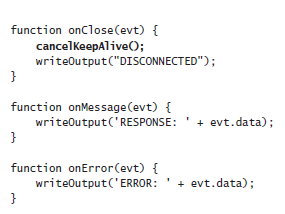
How to deal with timeouts:

* One way of dealing with timeouts is to send an empty message to the server periodically (assuming the timeout occurred due to inactivity)

An example of code that sends an empty message to the server periodically is shown below







Look at the following JSFiddle page for more information: https://jsfiddle.net/dtyf3ahc/3/

### Handling connection disconnects

You might also need to deal with connections that close due to network errors.  
This can require you to **call the connect function from within the onClose function**.Note: The server might not recognize that the "connect" call is coming from an existing client.  
Workaround idea: We can define a custom way to call the server whereby we pass client identifying data when sending information to the server (no example provided)

### Dealing with web farms

Sometimes in production, you need your application to run in a web farm, when multiple servers handle incoming requests. These requests are typically load balanced to provide the best performance.

Question: How can you have an open connection to a server and still have load balancing?

Answer:

* When there are multiple web servers, you can **implement sticky servers**, by which the client continuously goes to the same server it originally went to (This takes care of the open connection problem)
* You can **share state** across multiple web servers by using products such as Redis (remote dictionary service) and the Microsoft App Fabric Caching Service

Note: Remember to test your web farm environment early for compatibility and problems because you will surely need to adjust for this environment

## Using WebSocket libraries

WebSocket libraries

1. Help deal with timeouts, dropped connections, incompatible browsers, and web farms in a consistent manner.
2. Are client and server libraries (Must be used at both the client and server)

Examples of WebSocket libraries

* **SignalR**: Uses WebSocket when it’s available but gracefully falls back to techniques such as long polling when WebSocket is not available.
* **Socket.IO**: uses feature detection to decide whether the connection will be established with WebSocket, AJAX long polling, or Flash polling, which requires Flash at the browser.