**使用 Node.js 與 Socket.IO 建立即時性（Realtime）網頁應用程式 App**  
在大部分新的瀏覽器中，Socket.IO 其實都是使用 WebSocket 來傳輸，所以 Socket.IO 也可以視為一個 WebSocket 的包裝工具，但是他所提供的功能比 WebSocket 還要豐富，例如 Socket.IO 的 heartbeats、timeouts 與 disconnection 等功能對於即時性的應用程式而言都是很重要的，但是原生的 WebSocket API 卻沒有這些功能。

**安裝 Node.js 與 Socket.IO**

由於 Socket.IO 是建立在 [Node.js](http://nodejs.org/) 架構之上的工具，所以要先把 Node.js 先安裝好，請參考[在 Windows、Mac OS X 與 Linux 中安裝 Node.js 網頁應用程式開發環境](http://sealmemory.blogspot.tw/2013/12/install-node-js-in-windows-mac-os-x-linux.html" \t "_blank)。  
  
接著使用 npm 安裝 Socket.IO：

npm install socket.io

**基本 Server**

在 Node.js 中，有許多方式可以建立網頁伺服器，不過都大同小異，這裡我們使用 [http](http://nodejs.org/api/http.html) 函式庫：

var http = require('http');

var server = http.createServer();

server.listen(8001);

這樣就是一個最基本的網頁伺服器了，你可以將這段程式碼儲存成 server.js，然後執行

node server.js

但是這樣的伺服器完全任何的功能，還無法運作，接下來我們讓伺服器自動送出一個 Hello, World. 字串：

var http = require('http');

var server = http.createServer(function(request, response){

console.log('Connection');

response.writeHead(200, {'Content-Type': 'text/html'});

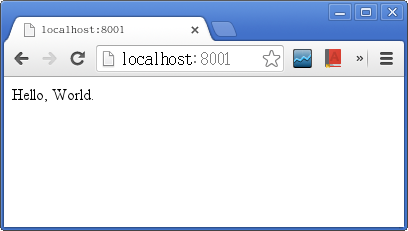
response.write('Hello, World.');

response.end();

});

server.listen(8001);

修改成這樣之後，再重新執行 node server.js，然後用瀏覽器打開 http://localhost:8001/ 這個網址，就可以看到 Hello, World. 字串顯示在瀏覽器上了。（如果你是在遠端的伺服器上測試，就把 localhost 改成對應的 IP 位址即可）

[](http://4.bp.blogspot.com/-jophW-2nRho/UxXH9ReV-HI/AAAAAAAARuo/br6B1jhEjZA/s1600/nodejs-socket-io-1.png)

而當瀏覽器連線至伺服器時，在終端機中也會顯示 Connection 這個連線的訊息，這樣就完成了一個陽春版的網頁伺服器了。  
  
接下來我們詳細說明一下上面這段程式碼的細節，首先是使用 http.createServer 建立伺服器的部份：

var server = http.createServer(function(request, response){});

server.listen(8001);

這裡放的匿名函數（anonymous function）是用來定義當伺服器接收到請求時，該做什麼事情，以及該如何回應。  
  
在這個子中，我們讓伺服器一接到請求時，就在終端機中輸出一行 Connection 的訊息：

console.log('Connection');

接著使用 response.writeHead() 設定 HTTP 回應的標頭資訊：

response.writeHead(200, {'Content-Type': 'text/html'});

第一個參數是 HTTP 協定中三位數的 status code（例如找不到網頁就是 404），第二個參數則是指定標頭資訊中的各個欄位屬性，這裡指定 content type 為一般的文字或 html。  
  
接著設定最主要的網頁內容：

response.write('Hello, World.');

最後結束整個定義過程：

response.end();

執行完這行，伺服器就會送出回應的訊息給瀏覽器。

**建立 Router**

上面我們所建立的是一個最基本的伺服器，不管使用者輸入的網頁是什麼，都只會送出一個 Hello, World. 字串，現在我們要改善這個問題，加入其他的網頁內容，將原本的 server.js 改為這樣：

var http = require("http");

var url = require('url');

var fs = require('fs');

var server = http.createServer(function(request, response) {

console.log('Connection');

var path = url.parse(request.url).pathname;

switch (path) {

case '/':

response.writeHead(200, {'Content-Type': 'text/html'});

response.write('Hello, World.');

response.end();

break;

case '/socket.html':

fs.readFile(\_\_dirname + path, function(error, data) {

if (error){

response.writeHead(404);

response.write("opps this doesn't exist - 404");

} else {

response.writeHead(200, {"Content-Type": "text/html"});

response.write(data, "utf8");

}

response.end();

});

break;

default:

response.writeHead(404);

response.write("opps this doesn't exist - 404");

response.end();

break;

}

});

server.listen(8001);

修改好之後，重新執行 node server.js 並打開 http://localhost:8001/ 這個網址，可會發現沒有什麼改變，但是當你輸入 http://localhost:8001/socket.html 這個網址時，則會出現 404 的錯誤訊息，告訴你 socket.html 這張網頁不存在。  
  
以下我們說明段程式碼的細節，首先是引入模組的部分：

var url = require('url');

var fs = require('fs');

這裡加入了 [url](http://nodejs.org/api/url.html) 與 [fs](http://nodejs.org/api/fs.html) 兩個模組，其中 url 是用來解析 URL 網址用的，而 fs 則是用處理檔案的模組。  
  
當伺服器接收到請求時，使用 url.parse() 函數解析出網址中所指定的路徑：

var path = url.parse(request.url).pathname;

假設瀏覽器所輸入的網址為 localhost:8001，則解析出來的路徑就會是 /，而如果網址是 localhost:8001/socket.html，則解析出來的路徑就會變成 /socket.html，如果你想要實際測試看看解析的狀況，可以在這行後面加上 console.log(path) 將解析的結果顯示在終端機中。  
  
在解析出路徑之後，我們使用一個 switch 判斷式，依照不同的路徑來進行不同的處理方式，如果路徑是根目錄 /，則輸出原來的 Hello, World. 字串，而如果路徑是 /socket.html 的話，就使用 fs 模組來讀取 socket.html 這個檔案的內容：

fs.readFile(\_\_dirname + path, function(error, data){...});

readFile() 跟許多的 Node.js 函數一樣，需要指定一個回呼函數（callback function），定義檔案在讀取之後要進行什麼動作，這個回呼函數的第一個參數代表錯誤代碼，如果在讀取檔案出問題時（例如檔案不存在），這個錯誤代碼就會被設定，而第二個參數則是檔案的內容。  
  
這裡我們先檢查讀取的過程是否有發生錯誤：

if (error){

response.writeHead(404);

response.write("opps this doesn't exist - 404");

}

如果有發生讀取錯誤，就回應 404 的錯誤訊息，告知使用者檔案不存在。而如果檔案讀取成功的話，就輸出檔案的內容：

else {

response.writeHead(200, {"Content-Type": "text/html"});

response.write(data, "utf8");

}

這裡輸出的做法跟之前輸出 Hello, World. 類似，但另外指定輸出的編碼為 utf8。  
  
最後一樣要記得加上 response.end() 結束整個處理流程的定義，讓它開始執行。  
  
由於到目前為止我們還沒有建立 socket.html 這個檔案，所以如果輸入 http://localhost:8001/socket.html 這個網址才會看到 404 的錯誤訊息。

**加入 Socket.IO**

接著我們建立一個 socket.html 檔案，內容如下：

<html>

<head></head>

<body>This is our socket.html file</body>

</html>

在 socket.html 建立好之後，當你在瀏覽器中輸入 http://localhost:8001/socket.html 這個網址時，就會顯示這個網頁的內容了，不過這個並不是這裡的重點，接下來就要開始加入 Socket.IO 的部份了。  
  
首先在伺服器端的 server.js 中加入 Socket.IO 模組：

var http = require("http");

var url = require('url');

var fs = require('fs');

var io = require('socket.io'); *// 加入 Socket.IO*

var server = http.createServer(function(request, response) {

console.log('Connection');

var path = url.parse(request.url).pathname;

switch (path) {

case '/':

response.writeHead(200, {'Content-Type': 'text/html'});

response.write('Hello, World.');

response.end();

break;

case '/socket.html':

fs.readFile(\_\_dirname + path, function(error, data) {

if (error){

response.writeHead(404);

response.write("opps this doesn't exist - 404");

} else {

response.writeHead(200, {"Content-Type": "text/html"});

response.write(data, "utf8");

}

response.end();

});

break;

default:

response.writeHead(404);

response.write("opps this doesn't exist - 404");

response.end();

break;

}

});

server.listen(8001);

io.listen(server); *// 開啟 Socket.IO 的 listener*

這裡只有更動兩個小地方而已，一個是在上方引入 socket.io 模組，例外在最後一行開啟一個 Socket.IO 的 listener，讓伺服器啟動時就可以準備接收來自於瀏覽器的 WebSocket 連線。  
  
現在當我們重新執行 node server.js 之後，瀏覽 http://localhost:8001/socket.html 這個網址時，在瀏覽器端沒有什麼變化，但是在伺服器的終端機中則會出現

info  - socket.io started

這樣的訊息，這就表示我們的伺服器現在已經可以接收來自於任何地方的 WebSocket 連線了。  
  
接著我們要修改 socket.html 的內容，加入 Socket.IO 的連線功能，讓瀏覽器可以透過 WebSocket 連線到我們剛剛寫好的伺服器上。以下是 socket.html 的內容：

<html>

<head>

<script src="/socket.io/socket.io.js"></script>

</head>

<body>

<script>

var socket = io.connect();

</script>

<div>This is our socket.html file</div>

</body>

</html>

這裡我們在網頁中引入了 Socket.IO 的 JavaScript 檔，並且呼叫 io.connect() 連線至伺服器，這時候如果重新載入這個網頁，你就可以在伺服器的終端機中看到類似下面這幾行資訊：

info  - socket.io started  
debug - served static content /socket.io.js  
debug - client authorized  
info  - handshake authorized 1agc6iSouA2QymxUaZ0D  
debug - setting request GET /socket.io/1/websocket/1agc6iSouA2QymxUaZ0D  
debug - set heartbeat interval for client 1agc6iSouA2QymxUaZ0D  
debug - client authorized for  
debug - websocket writing 1::

現在我們已經成功建立了一個伺服器與瀏覽器之間的 WebSocket 連線了，下面我們繼續示範如何讓伺服器透過這個連線傳送資料給瀏覽器。

**傳送資料至瀏覽器**

Socket.IO 傳送資料的方式跟一般 Node.js 程式所使用的方式差不多，都是以回呼函數的方式來處理，這裡我們使用 on() 函數將特定的事件連接到指定的匿名函數，藉此定義整個資料傳輸過程要如何運作。  
  
現在我們在 server.js 的最後，將 io.listen() 的傳回值儲存起來，並加入一小段程式碼：

var serv\_io = io.listen(server);

serv\_io.sockets.on('connection', function(socket) {

socket.emit('message', {'message': 'hello world'});

});

這裡我們使用 on() 函數將 connection 事件與一個匿名函數連接起來，這樣只要 WebSocket 連線一建立，這個匿名函數就會被呼叫。  
  
這裡的 connection 是一個由 Socket.IO 內建的事件，當瀏覽器端呼叫 io.connection() 之後，就會自動產生這個事件，進而呼叫上面這個匿名函數，而我們也可以自行定義事件，這裡馬上就有一個例子。  
  
在這個匿名函數中，當連線建立之後，我們使用 emit() 函數來傳送資料，這個函數在伺服器與瀏覽器端的用法是一樣的，作用就是將資料傳給對方。  
  
emit() 會產生一個事件，而其事件的名稱是透過第一個參數來定義的（以這個例子來說就是 message，當然你也可以使用其他的名稱），而第二個參數則是指定這個事件所伴隨的資料，而這個資料的格式則是一個 JSON 的物件。  
  
到這裡我們已經設定好在連線建立之後，讓伺服器送出一個訊息，接下來我們還要讓瀏覽器接收這段訊息才行，我們將 socket.html 修改一下：

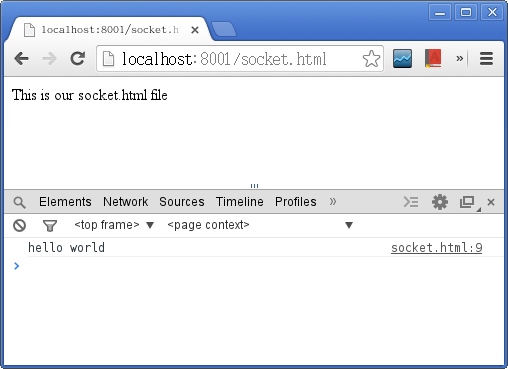
var socket = io.connect();

socket.on('message', function(data){

console.log(data.message);

});

我們在這裡也同樣使用 on() 函數連接一個匿名函數，讓臉器可以接收來自於伺服器的訊息（也就是接收由伺服器所產生的 message 事件），然後呼叫 console.log() 函數將訊息輸出在瀏覽器的 console 中。  
  
現在重新啟動 server.js 與重新整理瀏覽器之後，開啓瀏覽器的 JavaScript 除錯視窗，應該就會發現 hello world 出現在瀏覽器的 console 中了。

[](http://1.bp.blogspot.com/-fFfW96SKD_s/Uxhohrj81PI/AAAAAAAARwk/jW6NYTbNPXU/s1600/nodejs-socket-io-2.png)

然而如果僅僅只是像這樣傳送一個訊息，一般的網頁技術或是 Ajax 也可以輕易做到，而 Socket.IO 所擅長的其實是持續性的資料傳遞，接下來我們將實作一個網頁時鐘，讓伺服器每秒鐘傳遞一個時間的資訊給瀏覽器。  
  
在 JavaScript 中，有一個 setInterval() 函數，它可以讓指定的函數在指定的間隔時間下重複執行，例如若要每秒鐘輸出一行 hello world 字串，則可以這樣寫：

setInterval(function() {

console.log('hello world');

}, 1000);

而在我們的例子中，我們希望伺服器每秒傳送一個時間的訊息給瀏覽器，所以將 connection 事件所連接的匿名函數修改成這樣：

serv\_io.sockets.on('connection', function(socket) {

*// 傳送時間訊息給瀏覽器*

setInterval(function() {

socket.emit('date', {'date': new Date()});

}, 1000);

});

這裡我們每秒使用 JavaScript 的 Date() 函數產生一個 JSON 物件，傳送給瀏覽器。  
  
接著我們要讓瀏覽器可以接收這個 JSON 物件，並且顯示在網頁上，所以將 socket.html 修改成這樣：

<html>

<head>

<script src="/socket.io/socket.io.js"></script>

<script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"></script>

</head>

<body>

<script>

var socket = io.connect();

socket.on('date', function(data) {

$('#date').text(data.date);

});

</script>

<div id="date"></div>

</body>

</html>

這裡我們將接收到的時間資料透過 jQuery 即時放進網頁中，現在將伺服器重新啟動，載入新的網頁之後，你會發現網頁上的時間會每秒更新一次。  
  
接下來就是比較有趣的地方了，原本在 setInterval() 中我們設定每 1000 milisecond（一秒）送出一次時間訊息，你可以把它改成比較小的數字，看看傳送的過程是否正常，你應該會發現即使把它設為每千分之一秒傳送一次，它還是可以很正常的運作，這就是 WebSocket 的優勢所在。

**傳送資料至伺服器**

WebSocket 是一個允許雙向傳輸的協定，所以除了讓伺服器傳送資料至瀏覽器端之外，我們也可以讓瀏覽器上的資料即時傳回伺服器中，而且由於這裡我們是使用 Socket.IO，所以不管是從伺服器傳送至瀏覽器，或是從瀏覽器傳送至伺服器，程式的語法都相同（看到這裡我想你應該會很高興），這也是 Socket.IO 的優點之一。  
  
這裡我們在網頁中加入一個 textarea，並將使用者輸入的文字配合 jQuery 即時傳回伺服器端：

<html>

<head>

<script src="/socket.io/socket.io.js"></script>

<script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js"></script>

</head>

<body>

<script>

var socket = io.connect();

socket.on('date', function(data) {

$('#date').text(data.date);

});

$(document).ready(function(){

$('#text').keypress(function(e){

socket.emit('client\_data', {

'letter': String.fromCharCode(e.charCode)

});

});

});

</script>

<div id="date"></div>

<textarea id="text"></textarea>

</body>

</html>

這裡我們使用 jQuery 的 keypress 事件，這樣的話每當使用者按下一個鍵，他就會即時以 socket.emit() 送出。而 String.fromCharCode(e.charCode) 則是將按下按鍵的 character code 轉為一般的字串，也就是說如果按下 a 按鍵，整個 JSON 物件就會是 {'letter': 'a'}。  
  
接著還要再修改一下 server.js，讓伺服器可以接收來自於瀏覽器的資料：

var serv\_io = io.listen(server);

serv\_io.set('log level', 1); *// 關閉 debug 訊息*

serv\_io.sockets.on('connection', function(socket) {

setInterval(function() {

socket.emit('date', {'date': new Date()});

}, 1000);

*// 接收來自於瀏覽器的資料*

socket.on('client\_data', function(data) {

process.stdout.write(data.letter);

});

});

這裡我們加入一行 io.set('log level', 1);，這樣可以關閉那些 Socket.IO 輸出的除錯（debug）訊息，這樣我們才能看清楚來自於瀏覽器的資料。  
  
由於 console.log() 在輸出訊息時會自動加上換行字元，但是在這裡我們不希望他這麼做，所以改用 process.stdout.write() 來輸出資料。  
  
最後重新啟動伺服器並載入網頁，然後在網頁的 textarea 輸入一些文字。

[](http://4.bp.blogspot.com/-5RA3oA0XSZA/Uxhq6hf5MVI/AAAAAAAARw4/baiNyTNLeNo/s1600/nodejs-socket-io-3.png)

記得同時將伺服器的終端機與網頁都顯示在螢幕上，你會發現當你在網頁上輸入文字時，伺服器的終端機會即時同步地顯示每一個輸入的字母。