

Getting Pushy with Node.js and OpenEdge

Dustin Grau, Software Architect



BRAVEPOINT

5000 Peachtree Ind. Blvd.

Suite 100

Norcross, GA 30071

- Senior developer and consultant at BravePoint, Inc.
 - Founded in 1987 with currently ~125 employees
 - Consulting, training, and placement services
- WebSpeed application developer since 1999
- Implementing JS/AJAX frameworks since 2010
- Lead architect for modernization framework “Application Evolution”

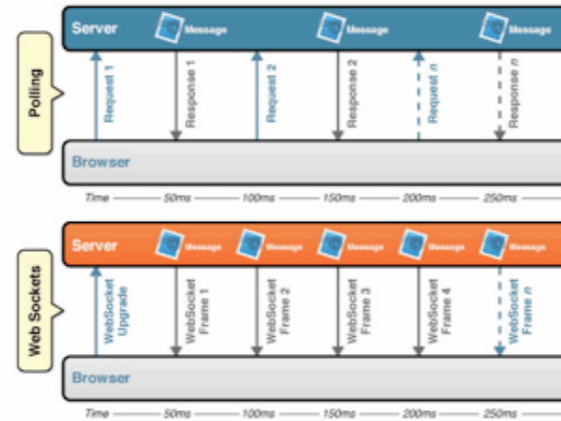
- Understand concepts that are critical to push technologies
- Quick primer on integrating ABL with Node.js and Socket.io
- You should leave here able to do all of this on your own!

- You will be able to download all software shown here today!

- If you request data, you are performing a “pull”
- Polling mechanisms operate via requests
- A true “push” only comes direct from the server
- You need 2-way communication for this to work
- This can be addressed by using WebSockets

- RFC 6455 - The WebSocket Protocol
- Provides 2-way communication to supported browsers
- Begins as a standard HTTP GET request from client
- Requests an “upgrade” to WebSocket protocol
- Server performs handshake and initiates connection
- Client remains connected to server
- Can be provided by Node.js and Socket.io
- This is where the magic happens...

- AJAX vs. WebSockets
- AJAX is expensive (headers, overhead, etc.)
- Polling causes network traffic for useless data
- WebSockets have a small handshake, occurs once
- Lower latency when using WebSockets vs. HTTP
- Means nothing if your browser doesn't support WS



- Progress OpenEdge (11+)
- Node.js (<http://nodejs.org>)
- Socket.io (<http://socket.io>)
- jQuery (<http://jquery.com>)
- A server (AWS, Modulus, etc.)
- A compatible browser
- A compatible device



- Sample Problem
- A Solution
- Demonstration
- Code How-to
- Future Enhancements
- Summary / Q&A

- Client wants immediate notification sent to users
- Needs to access this via a web application
- Wants to use minimal system resources
- Needs to target specific users with info
- We've got something for this...

- Should ideally...
 - Keep a low profile (cpu/memory) with minimal setup
 - Be able to deliver data continuously (no “spin up” time)
 - Handle multiple connections from interested parties
 - Data should update automatically, in near real-time
 - Maintain a level of security among data and subscribers

- Our solution...
 - Uses ABL code to send messages to Node.js
 - Works on any operating system where OpenEdge can be installed
 - Utilizes a separate, central process for all client connections
 - Is accessible by any web browser on any device
 - Uses WebSockets to push data to subscribed clients

- It's extremely lightweight, both for installation and runtime
- It can handle many, many simultaneous connections
- Can provide multiple services over the same port
- Has a package manager to provide additional capabilities
- Install is as simple as "npm install socket.io"
- Allows for event-driven applications using "on" statements
- Socket.io provides a single solution for 2-way communication
- Can use AJAX long polling and other fallback mechanisms

Node.js is essentially JavaScript for the server-side environment
It uses an asynchronous, multi-threaded, non-blocking I/O scheme
Made specifically for HTTP and WebSocket communication
Arbitrary events can be observed

Subscribed to db3b25a674d1819ce21162fb1d3ffad225467603

Updated at 12:08:27

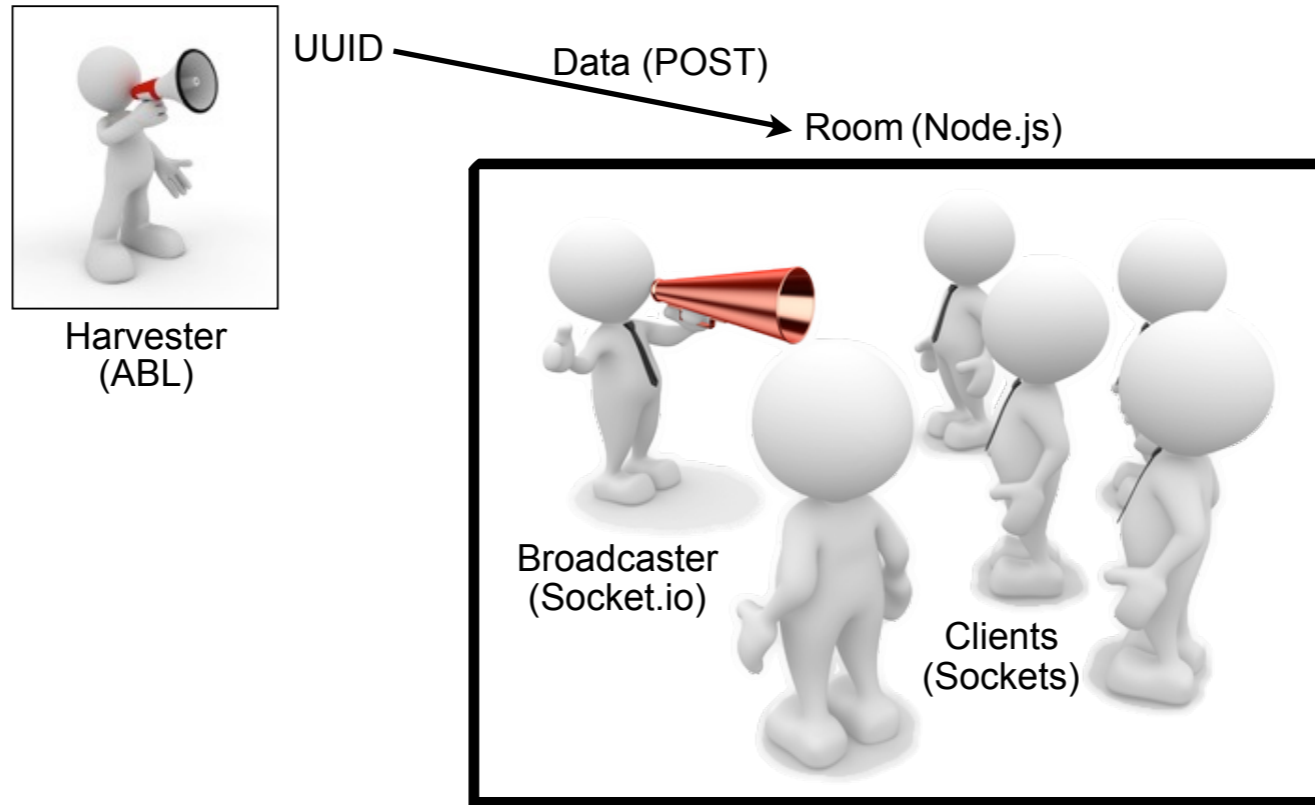
Reset Stats

Table Activity Index Activity **User Activity**

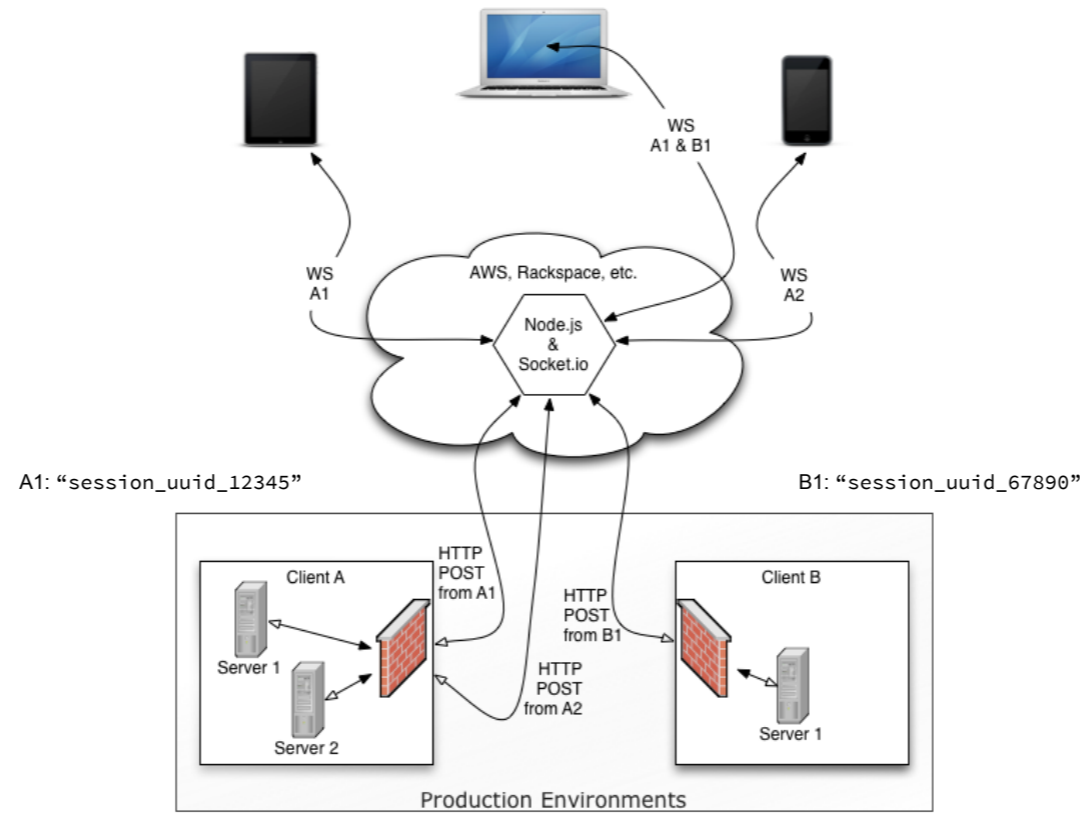
Username	Tenant	PID	DB Access	DB Reads	New Access	New Reads
demo@rei_com	REI	5348	9244	53	444	8
Dustin@windowsid	Default	5096	8003	1	404	0
demo@rei_com	REI	7560	9348	48	102	3
Anonymous	Default	568	11536	21	3	0
SYSTEM	NA	5928	3942	285	0	0
SYSTEM	NA	5800	4	0	0	0
SYSTEM	NA	4504	4	0	0	0
DB_Agent	NA	5268	4565993	138	0	0
SYSTEM	NA	872	4	0	0	0
SYSTEM	NA	4588	4	0	0	0
SYSTEM	NA	5416	13	0	0	0
SYSTEM	NA	5780	13	0	0	0
Anonymous	Default	3468	12178	136	0	0

<http://goo.gl/2KXQ7G>

Our newest version instead packages the data as a dataset in JSON for delivery.
 The UI handles display of the data in a web-based environment.
 Let's look at a live demonstration...



The harvester is the primary daemon, running on the production environment. A harvester sends data to a Node.js server, identifying itself by a UUID. The broadcaster is the Socket.io server, which is organized by UUID "rooms". Clients connect and join a room, where they receive data from harvesters. Only data for the UUID to which they subscribe will be sent to them.



We're not limited to a single room. There could be multiple clients with multiple servers utilizing the Node.js broadcaster service. Clients can subscribe to one or many server broadcasts by opening a new browser tab and entering the desired UUID. For our purposes a UUID is a 40-character value that identifies a running instance of our monitor procedure.

- Progress (Harvester)

- monitor.p
 - SysLoad.cls
 - WebSocket.cls

- Node.js (Broadcaster)

- index.js ←
- Socket.io (Server)
- client.html
 - Socket.io (Client)

```
/**
 * Provide service at a default port.
 */
var listen_port = 1337;

/**
 * Create an HTTP server to handle regular web requests.
 */
var http = require("http"); // HTTP service
var url = require("url"); // URL parser
var fs = require("fs"); // FileSystem
var httpServer =
http.createServer(onHttpRequest).listen(listen_port);

/**
 * Handle socket.io connections.
 */
var io = require("socket.io").listen(httpServer);
io.sockets.on("connection", onSocketConnect);
```

Node.js is a very open-ended server platform. It answers to a port, handling requests according to protocol. In this case we start up an HTTP server on a port, then extend that server to support WebSockets. Let's look at two handler methods for each of these server types.


```
function onHttpRequest(request, response) {
  var parsedUrl = url.parse(request.url);
  var pathName = parsedUrl.pathname || "";
  var pathArray = pathName.split("/"); // Convert to array.
  var pathVal = (pathArray.length > 1) ? pathArray[1] : "";

  switch(request.method){
    case "GET":
      doGet(pathVal, request, response);
      break;
    case "POST":
      doPost(pathVal, request, response);
      break;
    default:
      // Unsupported HTTP method.
      response.writeHead(405, "Method Not Allowed",
        {"Content-Type": "text/plain"});
      response.end();
  }
}
```

The first is the HTTP handler, which will deal with data from monitor.p and initial client requests.
The GET and POST handlers work as expected for those methods.

```
function doGet(pathVal, request, response){
  var filename = null;

  switch(pathVal){
    case "":
    case "client.html":
      // Serve the client HTML file on GET.
      filename = "/client.html";
      break;
    default:
      // File not found for serving.
      response.writeHead(404, "Not Found",
        {"Content-Type": "text/plain"});
      response.end();
  }

  if (filename) {
    fs.readFile(__dirname + filename, "utf8", function(error, content) {
      response.writeHead(200, "OK", {"Content-Type": "text/html"});
      response.end(content);
    });
  }
}
```

```
function doPost(pathVal, request, response){
  var uuid = pathVal; // In this case the path value is a UUID.
  var postData = ""; // Store the body data on POST.
  var count = 1;

  request.on("data", function(chunk){
    postData += chunk;
    if (postData.length > 1e6) {
      postData = ""; // Abort if data appears to be a [malicious] flood.
      response.writeHead(413, {"Content-Type": "text/plain"}).end();
      request.connection.destroy();
    }
    count++;
  }); // on data

  ...
}
```

```
request.on("end", function(){
  var responseBody = {response: "Broadcast Sent: " + uuid};
  if (postData != "") {
    var jsonObj = null;
    try {
      jsonObj = JSON.parse(postData);
    } catch(parseErr) {
      responseBody = {response: "JSON Error: " + parseErr.message};
    }

    // Add the UUID for socket broadcast.
    if (harvesters.indexOf(uuid) < 0) {
      harvesters.push(uuid);
    }

    // Broadcast to clients on socket server, based on UUID.
    if (jsonObj && io.sockets.clients(uuid).length > 0) {
      var room = io.sockets.in(uuid);
      room.emit("broadcast-data", jsonObj.activityData);
    }
  } // postData

  ...
}
```

We continue by checking for the "end" event on the request. This signals when we can begin the output process. First the data we gathered gets parsed into JSON, and we send the activity data to the socket server. The trick here is to see if anyone is in a room, identified by a UUID, and emit a broadcast if subscribers are present.

```
// Prepare response body with optional commands.
if (commands[uuid]) {
  responseBody.commands = commands[uuid];
  delete commands[uuid];
}

// End the response with a message.
var responseJSON = JSON.stringify(responseBody);
var responseHeaders = {
  "Content-Type": "application/json",
  "Access-Control-Allow-Origin": "*",
  "Content-Length": Buffer.byteLength(responseJSON)
};
response.writeHead(200, "OK", responseHeaders);
response.write(responseJSON);
response.end();
}); // on end
}
```

After making the broadcast, we check for any commands that may have been queued for this UUID. Finally, we prepare a response back to the Progress program, sending the body and headers.

```
function onSocketConnect(socket){
  /**
   * Handle requests to listen for broadcast consoles
   */
  socket.on("listen", function(uuid, callback) {
    // If the uuid looks legit, subscribe.
    if (uuid.length == 40) {
      socket.uuid = uuid; // Save UUID on socket.
      socket.join(uuid); // Join a "room" for UUID.

      // Callback to the listener with a successful flag.
      callback(true);
    } else {
      // If the uuid is not available, reject the request to listen.
      callback(false);
    }
  }); // on listen

  ...
}
```

Now the fun part: socket!

We need a handler for the socket server, which accepts the a socket connection.

Here we can define events on the socket, such as "listen" which is used by client connections.

```
...  
  
/**  
 * Handle requests for new commands back to harvester  
 */  
socket.on("send-command", function(uuid, command) {  
  // If the uuid is being broadcast, add to queue.  
  if (harvesters.indexOf(uuid) >= 0) {  
    if (commands[uuid] && commands[uuid] instanceof Array) {  
      // Queued commands pending, add to existing list.  
      if (commands[uuid].indexOf(command) == -1) {  
        // Prevent adding duplicate commands.  
        commands[uuid].push(command);  
      }  
    } else {  
      // No queued commands, create new queue for uuid.  
      commands[uuid] = [command];  
    }  
  }  
}); // on send-command  
}
```

Events can be completely arbitrary, and in this case the server listens for a "send-command" event. In this case, it accepts a list of commands for a given UUID, and queues them for return to Progress.

```
Windows:  
C:\> node index.js
```

```
Linux:  
# node index.js
```

To start a Node.js program, it's as simple as running "node" with the name of a JavaScript file as parameter.

- Progress (Harvester)

- monitor.p
 - SysLoad.cls
 - WebSocket.cls

- Node.js (Broadcaster)

- index.js
 - Socket.io (Server)
- client.html 
 - Socket.io (Client)

```
<body>
  <form id="sub-form">
    <label for="uuid-input">Enter a UUID:</label>
    <input id="uuid-input" maxLength="40" size="50" value="db3b25a674d1819ce21162fb1d3ffad225467603" />
    <input type="submit" id="sub-button" value="Subscribe" />
    <div id="uuid-error">The specified UUID is invalid.</div>
  </form>
  <div id="console">
    <div id="monitor-header">
      <div id="monitor-uuid"></div>
      <div id="monitor-time"></div>
      <div id="monitor-controls">
        <form id="command-form"><input type="button" id="reset-button" value="Reset Stats" /></form>
      </div>
    </div>
    <div id="monitor-data">
      <ul>
        <li><a href="#tab-1">Table Activity</a></li>
        <li><a href="#tab-2">Index Activity</a></li>
        <li><a href="#tab-3">User Activity</a></li>
      </ul>
      <div id="tab-1">
        <table id="activity-table"></table>
      </div>
      <div id="tab-2">
        <table id="index-table"></table>
      </div>
      <div id="tab-3">
        <table id="user-table"></table>
      </div>
    </div>
  </div>
</body>
```

The structure of the HTML document provides structure for the webpage.
Several DIV tags are merely placeholders for data that will be dynamically loaded.

```
<link rel="stylesheet" src="http://normalize-css.googlecode.com/svn/trunk/normalize.css" />
<link rel="stylesheet" src="http://code.jquery.com/ui/1.10.3/themes/smoothness/jquery-ui.css" />
<script src="http://code.jquery.com/jquery-1.10.2.js"></script>
<script src="http://code.jquery.com/ui/1.10.3/jquery-ui.js"></script>
<script src="/socket.io/socket.io.js"></script>
```

A few other files are included, like stylesheets and javascript libraries.
Note that most files are linked to CDN's, as we don't want to serve them up from Node.js
However, note that socket.io.js is actually served relative to the document root.

```
<script>
  $(document).ready(function() {
    // Give focus to the UUID input when the page loads
    $("#uuid-input").focus();

    // Create a connection to the server
    var socket = io.connect(document.URL);

    // Prepare tab interface.
    $("#monitor-data").tabs();

    // Handler for updating screen data.
    $.updateConsole = function(data) { ... }

    // Handler for subscription event.
    $.doSubscribe = function() { ... }

    // Handle submission of the form, try to subscribe to the UUID.
    $("#sub-form").submit(function(ev){
      // Prevent the browser from submitting the form via HTTP
      ev.preventDefault();

      // Attempt to subscribe to the given UUID.
      $.doSubscribe();
    });
  });
</script>
```

Using some typical JQuery code, we set up a method to run when the document finishes loading. One of the important items is the creation of the socket connection via `io.connect()`. This connects to the local Node.js server using `socket.io`, which creates a `WebSocket` on compatible browsers. The first major method is `doSubscribe`, which is called when submitting the form with a UUID present.

```
$.doSubscribe = function(){
    var uuid = $("#uuid-input").val();
    if (uuid) {
        socket.emit("listen", uuid, function(successful) {
            if (successful) {
                // Hide the subscription form and show the main console.
                $("#sub-form").hide();
                $("#console").show();
                $("").addClass("system-message").text("Subscribed to "
                    + uuid).appendTo("#monitor-uuid");

                // Handle incoming broadcasts (sends "data" as a parameter).
                socket.on("broadcast-data", $.updateConsole);


                // Handle sending of commands.
                $("#reset-button").click(function(ev) {
                    // Queue command for the next broadcast from harvester.
                    socket.emit("send-command", uuid, "reset");
                    alert("Statistics will be reset after the next update.");
                });
            } else {
                // If the request to subscribe was rejected, show an error message.
                $("#uuid-error").show();
            }
        });
    }
}
```

This method emits the "listen" event on the Node.js/Socket.io server.
Upon the callback function running with a successful response, the client begins watching for the "broadcast-data" event.
In the event the reset button is clicked, a command is queued via the "send-command" event.

```
$.updateConsole = function(data){
  // Split data into tables.
  var monitorData = null;
  if (typeof(data) == "string") {
    try {
      // Always parse in a try/catch block!
      monitorData = JSON.parse(data);
    } catch(parseErr) {
      // Fail silently.
    }
  } else if (typeof(data) == "object") {
    monitorData = data;
  }
  monitorData = monitorData || {};
  var activity = monitorData.returnAct || [];
  var indexAct = monitorData.returnIdxAct || [];
  var userAct = monitorData.returnUsrAct || [];

  $("#activity-table").empty();
  $("#index-table").empty();
  $("#user-table").empty();
  ...
}
```

This method is the callback for the "broadcast-data" event. It takes the data returned (as JSON) and parses it into an object. The object is then split into the tables (this was originally a dataset), and the HTML tables are populated dynamically.

- Progress (Harvester)
 - monitor.p 
 - SysLoad.cls
 - WebSocket.cls
- Node.js (Broadcaster)
 - index.js
 - Socket.io (Server)
 - client.html
 - Socket.io (Client)

This solution consists of two primary parts: the harvester and the broadcaster.
monitor.p is the only program that must run on the Progress side
index.js is the main program on the Node.js side,
which also serves up the client.html file for clients

start.sh

```
#!/bin/bash
export PROPATH=./;$1
_progres -b -p monitor.p -pf startup.pf > logs/monitor.log
```

startup.pf

```
# Database to Monitor

-db Sports2kMT
-H localhost
-N TCP
-S 8650

-rereadnolock
-T ./temp
-TB 31
-TM 32
-rand 2
-mmax 8000
```



```
&GLOBAL-DEFINE THROW ON ERROR UNDO, THROW
  &GLOBAL-DEFINE SERVER_IP "127.0.0.1"
  &GLOBAL-DEFINE SERVER_PORT 1337
  &GLOBAL-DEFINE UUID "db3b25a674d1819ce21162fb1d3ffad225467603"
  &GLOBAL-DEFINE PROCESS_WAIT 10
  &GLOBAL-DEFINE MAX_ROWS 30

ROUTINE-LEVEL ON ERROR UNDO, THROW.
USING com.bravepoint.*.

DEFINE VARIABLE oSysLoad AS SysLoad NO-UNDO.
ASSIGN oSysLoad = NEW SysLoad( {&SERVER_IP},
                               {&SERVER_PORT},
                               {&UUID},
                               {&PROCESS_WAIT},
                               {&MAX_ROWS} ).

MESSAGE SUBSTITUTE("&1 &2] Starting monitor...",
                  STRING(TODAY, "99/99/9999"), STRING(TIME, "HH:MM:SS")).
oSysLoad:startMonitor().

FINALLY:
  DELETE OBJECT oSysLoad NO-ERROR.
END FINALLY.
```

The monitor starts a new instance of SysLoad against the connected database.
This sends data (in JSON format) to the specified server and port, using a UUID.

```
METHOD PUBLIC VOID startMonitor ():
...
MAINBLK:
REPEAT:
    waitFor().
    buildSample().

    IF DATASET activityData:WRITE-JSON("LONGCHAR", cRequest, FALSE,
        "UTF-8", FALSE, TRUE) THEN DO:
        jsonRequest:Add(INPUT "activityData", INPUT cRequest).
        jsonRequest:Write(INPUT-OUTPUT cRequest, INPUT TRUE, INPUT "UTF-8").
        jsonRequest:Remove(INPUT "activityData").
        oWebSocket:sendData( INPUT SUBSTITUTE("http://&1:&2/&3",
            cServerAddr, iServerPort, cMonitorUUID),
            INPUT cRequest,
            OUTPUT iStatus,
            OUTPUT cMimeType,
            OUTPUT cResponse ).

        ...
    END. /* WRITE-JSON */

    ...
END.
END METHOD. /* startMonitor */
```

SysLoad:startMonitor provides 2 services:

- 1) Generation of statistics on a running database.
- 2) Sending of data to Node.js via the WebSocket class.
- 3) Just does an HTTP POST using a Progress socket.

Harvester	Broadcaster	Client
<code>oSysLoad:startMonitor()</code>	<code>httpServer = http.createServer</code> <code>io.listen(httpServer)</code>	<code>socket = io.connect(document.URL)</code>
	<code>socket.on("listen", ...)</code>	<code>socket.emit("listen", ...)</code>
<code>oWebSocket:sendData(...)</code>	<code>room = io.sockets.in(uuid)</code> <code>room.emit("broadcast-data", ...)</code>	<code>socket.on("broadcast-data", ...)</code>
[POST Response]	<code>socket.on("send-command", ...)</code>	<code>socket.emit("send-command", ...)</code>

After this, show demo again to reinforce what we've done.

Subscribed to db3b25a674d1819ce21162fb1d3ffad225467603

Updated at 12:08:27

Reset Stats

Table Activity Index Activity User Activity

Username	Tenant	PID	DB Access	DB Reads	New Access	New Reads
demo@rei_com	REI	5348	9244	53	444	8
Dustin@windowsid	Default	5096	8003	1	404	0
demo@rei_com	REI	7560	9348	48	102	3
Anonymous	Default	568	11536	21	3	0
SYSTEM	NA	5928	3942	285	0	0
SYSTEM	NA	5800	4	0	0	0
SYSTEM	NA	4504	4	0	0	0
DB_Agent	NA	5268	4565993	138	0	0
SYSTEM	NA	872	4	0	0	0
SYSTEM	NA	4588	4	0	0	0
SYSTEM	NA	5416	13	0	0	0
SYSTEM	NA	5780	13	0	0	0
Anonymous	Default	3468	12178	136	0	0

<http://goo.gl/2KXQ7G>

- Use ABL to send messages to Node.js via POST requests
- You can run Node.js locally or externally (hosted platform)
- A separate server for client connections bridges the gap
- Organize connections into manageable notification groups
- Node.js is lightweight and serves all connections (HTTP & WS)
- Socket.io provides realtime data and handles fallback gracefully
- Certainly not limited to just the scenario presented here
- Code is available for you to try immediately!
- So, any questions?

- Get out your phones or laptops, everybody!
- Join the hotel network “RegencyGuestWifi”
- Open your browser and go to the URL below:
 - <http://goo.gl/e9zUd5>
- Sit back and wait...

Thank You!

BRAVEPOINT

Dustin Grau

dgrau@bravepoint.com



<http://goo.gl/Xoikn9>