Introduction to the node.js Asynchronous Programming Model

DEV162

Exercises / Solutions

Thomas Jung / SAP Labs, LLC

Contents

[Exercise 1 – Hello World 3](#_Toc425158848)

[Exercise 1 – Solution 4](#_Toc425158849)

[Exercise 1.1: Console Hello World 4](#_Toc425158850)

[Exercise 1.2: Web Hello World 8](#_Toc425158851)

[Exercise 2 –Reuseable modules 10](#_Toc425158852)

[Exercise 2 – Solution 11](#_Toc425158853)

[Exercise 2.1: Modules and NPM 11](#_Toc425158854)

[Exercise 2.2: Express Module and Package.json 13](#_Toc425158855)

[Exercise 2.3: Express and Static HTML content 16](#_Toc425158856)

[Exercise 2.4: Local Modules 17](#_Toc425158857)

[Exercise 3 –Asynchronous / non-blocking 19](#_Toc425158858)

[Exercise 3 – Solution 20](#_Toc425158859)

[Exercise 3.1: Basic Asynchronous 20](#_Toc425158860)

[Exercise 3.2: Non Blocking I/O 21](#_Toc425158861)

[Exercise 3.3: Non Blocking HTTP Requests 23](#_Toc425158862)

[Exercise 3.4: Non Blocking Database Requests 24](#_Toc425158863)

[Exercise 4 –SAP XS Specific Capabilities 27](#_Toc425158864)

[Exercise 4 – Solution 28](#_Toc425158865)

[Exercise 4.1: Text Bundles 28](#_Toc425158866)

[Exercise 4.2: HANA Database Access 32](#_Toc425158867)

[Exercise 4.3: XSJS compatibility 34](#_Toc425158868)

[Exercise 5 –Web Sockets 37](#_Toc425158869)

[Exercise 5 – Solution 38](#_Toc425158870)

[Exercise 5.1: Web Sockets 38](#_Toc425158871)

Exercise 1 – Hello World

Objective

In this first exercise, you will learn the basics of the most simple of Node.js applications. We will focus on how to start an application and how to display output.

Exercise Description

* Console Hello World
* Web Hello World

Exercise 1 – Solution

## Exercise 1.1: Console Hello World

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. On your desktop, there should be a short cut to the Student (local) folder. Click this link to open Windows Explorer at this location. |  |
| 1. Windows Explorer should open to D:\Files\Session. Inside this folder is a series of sub-folders for each hands-on workshop at TechEd. Find folder DEV162. This is where we will place all of our work for this hands-on session. |  |
| 1. You will notice in the DEV162 folder you have several things. First Node.js and Python are both installed locally. The apps folder is where you will place all of your coding. There is also a shortcut called Node.js command prompt. Click it now to open a DOS prompt with the proper setup for running Node.js commands. |  |
| 1. The command window should open with all the proper environment variables already set. You can keep this window open or reopen it via the link in the root of the DEV162 folder whenever you need to issue node.js commands. |  |
| 1. Return to Windows Explorer and navigate to the apps folder within DEV162. There are many samples and example implementations already in this folder. This is also where you will create your applications. |  |
| 1. Create a new folder named Exercise1. |  |
| 1. Create a new text document within this folder |  |
| 1. Rename the file hello.js |  |
| 1. You will receive a warning because you are changing the file extension from txt to js. This is fine and you should just hit Yes to accept the change. |  |
| 1. Right mouse click on the hello.js file and choose Edit with Notepad ++. |  |
| 1. We want to issue the text HelloWorld to the Node.js console (DOS command line). You can use the API console.log to issue text to the console output. Save your file and close the editor. | console.log("Hello World"); |
| 1. Return to the Node.js command prompt you opened earlier. Change to the D drive. |  |
| 1. Now change directories to go up to the DEV162 folder |  |
| 1. Change directories again to go into apps\Exercise1. |  |
| 1. We can invoke the node.js runtime by issuing the command node. The parameter after the node command tells it what JavaScript file to execute.   Therefore issue: node hello.js to run your first exercise. |  |
| 1. You should see the text you sent into the console.log returned to the command prompt. |  |
| 1. Congratulations! You just wrote your first Node.js application. |  |

## Exercise 1.2: Web Hello World

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. Returning to the Windows Explore, copy the hello.js from the first part of the exercise. Change the name of the copied file to helloWeb.js. |  |
| 1. The previous example was very simple and sometimes you want to issue interactive texts to the console. However more likely you want to issue messages via a web interface. Node.js has a built-in web server; therefore you can directly interact with the HTTP Request or Response objects. In this exercise create a small service that returns the Hello World text in a web browser. |  |
| 1. Node.js uses the syntax **require** to load a dependent library file. Require the built-in **http** library. Use the **createServer** API to start an HTTP server and then listen on port 3000 of your localhost (IP 127.0.0.1). The **createServer** API passes in the request and response objects as parameters. Use **res.end** to write the body of the response object. | var http = require('http');  http.createServer(function (req, res) {  res.end('Hello World\n');  }).listen(3000, '127.0.0.1');  console.log('Server running at http://127.0.0.1:3000/'); |
| 1. From the node.js command prompt execute the helloWeb.js you just created. Unlike the earlier example, it doesn’t immediately return control to the DOS prompt. Node continues to run because it’s listening for HTTP requests. |  |
| 1. Open a web browser and navigate to your node.js application which is running on <http://localhost:3000>. You should see the text of the message you passed back in the response object. |  |
| 1. The HTTP server for your service will continue to run until you manually terminate it. Return to the Node.js command prompt and press CTRL+C to terminate the service listener. |  |

Exercise 2 –Reuseable modules

Objective

In this exercise, you will learn about how to create and use reusable code in the form of modules. You will use both the npm tool and packages.json to define dependencies to these modules and make the installation of them quite easy. You will use one of the most popular modules – **express**; which helps with the setup of a web server and the handling of the request and response object. You will use **express** to handle dynamic and static content in the same application by using routes.

Exercise Description

* Modules and NPM
* Express Module and package.json
* Express and static HTML content
* Local Modules

Exercise 2 – Solution

## Exercise 2.1: Modules and NPM

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. Return to the D:\Files\Session\DEV162\apps folder in Windows Explorer. Create a new folder named Exercise2 for all the development objects in this exercise. |  |
| 1. In exercise 1.2 we created a hello world that runs from the web, but you had to manually open the web browser. Wouldn’t it be nice if the application opened the browser for you? Luckily there are many open source libraries available for Node.js and they are very easy to use and install. We will use one such library to extend the previous exercise to open a web browser for you.  Begin by copying the helloWeb.js file from the Exercise1 folder to Exercise2. |  |
| 1. Add the following line of code at the beginning of this file. This **require** statement gives us access to the **opn** library. Unlike the http library we used earlier, this isn’t built into node.js and has to be installed in a later step. | var opn = require('opn'); |
| 1. Add this line of code to the end of the file. This will trigger the opening of the default web browser to the web page where our application is listening. | opn('http://127.0.0.1:3000/'); |
| 1. From the node.js command prompt we can install the **opn** library using the **npm** tool. First make sure you are in the Exercise2 directory. Then issue the command **npm install opn** to install the **opn** library.   It will take a few seconds to fetch the library from the public repository on the internet.   Of course **npm** had to be configured with the URL of the repository and the HTTP proxy settings. However we did that configuration in advance on your desktop. For more information on **npm** configuration see this link: https://docs.npmjs.com/cli/config |  |
| 1. If you now run your helloWeb.js, you will see that a web browser opens automatically for you.   Don’t forget to return the command prompt and use Ctrl+C to terminate the service when you are finished with this exercise. |  |

## Exercise 2.2: Express Module and Package.json

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. One of the most common tasks in node.js is acting as a web server and handling http requests/responses. **express** is a module that wraps the low level **http** library and provides many additional services. Therefore it is widely used.   In this exercise we will adapt our helloWeb to use **express** instead of **http** library.   Begin by copying your helloWeb.js to helloWeb2.js in our Exercise 2 folder. |  |
| 1. Replace the require statement for the **http** library with one for the **express** library. | var express = require('express'); |
| 1. We also want to get a little more sophisticated with our HTTP port assignment. Instead of hard coding an HTTP port, use the built-in object called **process** to query the **PORT** which node.js is running under. If none is found fall back to port 3000. This is useful in the XS environment because the controller framework in XS will assign and pass in a distinct port to each application. | var PORT = process.env.PORT || 3000; |
| 1. Replace the rest of the code in your application with this code. It will create an instance of **express** and setup a route. Routes are code handlers for certain HTTP URL paths. In our case we will now only respond to GET requests to the /hello path.   Save your code. | var app = express();  //Hello Router  app.route('/hello')  .get(function(req, res) {  res.send('Hello World');  })  // Start the server  var server = app.listen(PORT, function() {  console.log('Listening on http://localhost:'+ PORT +'/hello');  opn('http://localhost:'+ PORT + '/hello');  }); |
| 1. As we get more and more libraries in our node.js applications, we don’t want to have to rely upon someone always manually using the **npm install** command to include them all.   Instead we can create a package that lists all the dependencies and their particular versions. We have a template package for you to copy over.   Open the Student (Share) from the desktop icon |  |
| 1. Navigate to DEV162\solutions\Exercise2 on the Student share. Copy the file package.json file from the share into your local Exercise2 folder. |  |
| 1. Look at the package.json file in the editor. You will see the dependencies section which lists all required libraries and their versions. |  |
| 1. Go to the node.js command prompt and issue the command **npm install**. This will read the package.json and install all libraries listed in the dependencies section.  Express is much larger library; so many more dependencies were installed as well. |  |
| 1. You can now run your application from the Node.js command prompt. It should open as before with the Hello World message. |  |
| 1. Try changing the URL and remove the /hello. You should receive an error message because we have no Route for that URL.   Don’t forget to return the command prompt and use Ctrl+C to terminate the service when you are finished with this exercise. |  |

## Exercise 2.3: Express and Static HTML content

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. One of the additional features of **express** is that it makes it easy to serve out static HTML content alongside your dynamic services. In this exercise we will add a static resource route for the root of our application.  Copy the helloWeb2.js file to helloWeb3.js. |  |
| 1. Add a line of code before the hello route to redirect requests from the root of the URL to static content in the **html** sub-directory. | //Home Router  app.use('/', express.static(\_\_dirname + '/html')); |
| 1. Open the Student (Share) from the desktop icon  Navigate to DEV16\solutions\Exercise2 on the Student share. Copy the folder html from the share into your local Exercise2 folder. |  |
| 1. You can now run your application from the Node.js command prompt. It should open as before with the Hello World message. |  |
| 1. Try changing the URL and remove the /hello. You should now receive the XS Engine welcome screen instead of the error message you got earlier.    Don’t forget to return the command prompt and use Ctrl+C to terminate the service when you are finished with this exercise. |  |

## Exercise 2.4: Local Modules

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. Modules don’t all have to come from central repositories. They can also be a way of modularizing your own code (similar to XSJSLIB files). Let’s now add another route but have the handler for it be contained in a module.  Copy the helloWeb3.js file to helloWeb4.js. |  |
| 1. Toward the beginning of your file add a require statement that points to a library named **myModule** which we will create locally in just a few steps. | var myModule = require('./myModule'); |
| 1. Add another route handler for the URL path **/module** that calls the **helloModule** function in the **myModule** library. | //Module Router  app.route('/module')  .get(function(req, res) {  res.send(myModule.helloModule());  }) |
| 1. Open the Student (Share) from the desktop icon  Navigate to DEV16\solutions\Exercise2 on the Student share. Copy the file myModule.js from the share into your local Exercise2 folder. |  |
| 1. Notice that the myModule.js uses the syntax **module.exports**. Any function declarations within this block are exposed to the outside. Other, “normal” functions are then effectively private to this this library and not accessible outside. |  |
| 1. You can now run your application from the Node.js command prompt. It should open as before with the Hello World message. |  |
| 1. Try changing the URL and alter the /hello to /module. You should now receive the new message which was coded in the reusable local library.    Don’t forget to return the command prompt and use Ctrl+C to terminate the service when you are finished with this exercise. |  |

Exercise 3 –Asynchronous / non-blocking

Objective

In this exercise, you will learn about the fundaments of the asynchronous nature of nodel.js. We will so see how this asynchronous capability allows for non-blocking input and output. This technique is one of the basic things that makes node.js development different from other JavaScript development and also creates one of the reasons for its growing popularity. We will see how these techniques are applied to common operations like HTTP web service calls or even SAP HANA database access.

Exercise Description

* Basic asynchronous processing
* Non-blocking I/O
* Non-blocking HTTP requests
* Non-blocking database requests

Exercise 3 – Solution

## Exercise 3.1: Basic Asynchronous

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. Return to the D:\Files\Session\DEV162\apps folder in Windows Explorer. Create a new folder named Exercise3 for all the development objects in this exercise. |  |
| 1. We want first to look a very simple example that shows the asynchronous nature of node.js. Begin by creating a file named async.js. |  |
| 1. In this code we will output a start message to the console, then set a timer which will issue a message after 3 seconds. Finally we will issue an ending message to the console. | console.log('Start');  setTimeout(function(){  console.log('Wait Timer Over');  }, 3000);  console.log('End'); |
| 1. What do you expect this code will output? From many other programming languages we would expect sequential processing and therefore the End output wouldn’t come until after the timer expired. However part of the power of node.js is asynchronous non-blocking execution of many core elements. In your node.js command prompt, change to Exercise3 folder. Run the async.js from the command prompt to see the results. |  |

## Exercise 3.2: Non Blocking I/O

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. Perhaps a timer seemed like an obvious asynchronous operation. However this asynchronous nature of node.js is often used when programs must wait on input or output. When asynchronous processing is applied to these operations, we can keep from blocking execution of other logic while we wait on things like file access, http requests or even database query execution.   Let’s first look at the difference between synchronous and asynchronous file operations.   Create two text files in your Exercise3 folder named file.txt and file2.txt. Place a little text in each file. The actual content isn’t that important. You can also copy these text files from the solutions folder on the Student share if you wish. |  |
| 1. Now create a file named fileSync.js. Using the fs library and the function **readFileSync**, read each of the text files. Output the content of each file to the console. After each read operation output a console message. | var fs = require('fs');  var text = fs.readFileSync('file.txt','utf8');  console.log(text);    console.log("After First Read\n");    text = fs.readFileSync('file2.txt','utf8');  console.log(text);    console.log("After Second Read\n"); |
| 1. Test your fileSync.js from the node.js command prompt.   As you might expect, everything is output in exactly the same order as the lines of code were listed in the application because all operations were synchronous. Program execution didn’t continue until each read operation had completely finished. |  |
| 1. Create an additional file named fileAsync.js or copy from the fileSync.js in the previous step. Adjust the logic to use the **fs.readFile** function. Notice that the **console.log(text)** now is embedded as an in-line callback function. It doesn’t get executed until the read operation is complete, but the rest of the program flow continues and isn’t blocked by the file operation. | var fs = require('fs');  fs.readFile('file.txt','utf8', function(error, text){  console.log(text);  });  console.log("After First Read\n");    fs.readFile('file2.txt','utf8', function(error, text){  console.log(text);  });  console.log("After Second Read\n"); |
| 1. Now run fileAsync.js from the node.js command prompt. The output of this exercise gives us very different results. Both after comments are output before either of the file contents. Also if the first file had been significantly larger than the second, it’s possible that the second might have finished and output first. .   This has powerful implications to how we code applications. |  |

## Exercise 3.3: Non Blocking HTTP Requests

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. Similar to file operations, HTTP requests are another area where our programs must often wait on an external response. In this exercise let’s see how node.js also makes calling external HTTP services non-blocking.   Create a file named httpClient.js in your Exercise3 folder. |  |
| 1. The **http** library we used in earlier exercises can also be used to make HTTP requests. Use the **get** function of the **http** library to call to <http://www.loc.gov/pictures/search/?fo=json&q=SAP>. This will call the US Library of Congress Image Search (a REST API which requires no authentication or API Key to keep the exercise simple). Issue a console message before and after the HTTP request. You can also copy this code from the Solutions folder on the Student share if you wish. | var http = require('http')  console.log("Before HTTP Call\n");  http.get(  {path: "http://www.loc.gov/pictures/search/?fo=json&q=SAP",  host: "proxy.fair.sap.corp",  port: "8080",  headers: {  host: "www.loc.gov"  }},  function (response) {  response.setEncoding('utf8');  response.on('data', function(data){console.log(data.substring(0,100))});  response.on('error', console.error);  });  console.log("After HTTP Call\n"); |
| 1. Test your httpClient.js from the node.js command prompt.   Similar to the earlier file exercise, the after http call console message is output before the response from the HTTP request. |  |

## Exercise 3.4: Non Blocking Database Requests

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. Perhaps most interesting to us is that this non-blocking concept can also be extended to database access. This allows us to issue multiple requests to the underlying HANA database in parallel and without stopping the processing flow of the JavaScript application logic. |  |
| 1. In Exercise 4 you will learn more about making database access to HANA. For this exercise we’ve already coded the database requests in a reusable module for you, so you can concentrate on the asynchronous flow.   Therefore from the Solutions/Exercise3 folder of the Students share, please copy the node\_modules folder and the database.js file into your local Exercise3 folder. |  |
| 1. Create a new file named databaseAsync.js. Issue a message to the console, then call two functions (**callHANA1** and **callHANA2**), then issue another message to the console. This will execute two different queries in the HANA database. | var hana = require('./database');  console.log('Before Database Call');  hana.callHANA1(console.log);  hana.callHANA2(console.log);  console.log("After Database Call"); |
| 1. Test your databaseAsync.js from the node.js command prompt.   As you are hopefully learning to expect, the messages you issued after the database requests are actually output first. Only then are the database query results returned. There is also no guarantee that query 1 will finish before query 2.   Don’t forget to return the command prompt and use Ctrl+C to terminate the service when you are finished with this exercise. |  |
| 1. But what if you want more control over the flow of program execution. Maybe you want several database operations to happen in parallel, but then some logic to execute only after all queries are complete. This is one of things the **async** library in node.js can make easier.   Copy the databaseAsync.js to databaseAsync2.js. Adjust the logic to use the **async.parallel** function. This allows some of the commands to execute in parallel as before, but then have a sync point once all operations are complete to allow further processing. We will output one final message after everything is done. | var hana = require('./database');  var async = require("async");  async.parallel([  function(cb){console.log('Before Database Call'); cb()},  function(cb){hana.callHANA1(cb, console.log); },  function(cb){hana.callHANA2(cb, console.log); },  function(cb){console.log("After Database Call"); cb();}  ], function(err){  setTimeout(function() {  console.log("---Everything's Really Done Now. Go Home!---");  process.exit();  }, 100);  }); |
| 1. Test your databaseAsync2.js from the node.js command prompt.  The execution is similar to before, but now we have the final message after all queries are complete. Notice that we don’t have to manually kill the service with Ctrl+C this time either. Because we have a sync point after all parallel execution is complete, we can exit the process safely.   Note: We did have to use a timer with a delay of 1/10 of a second otherwise the process would close before the last query console log was done being output making the results look odd. |  |

Exercise 4 –SAP XS Specific Capabilities

Objective

In this exercise, you explore the SAP HANA XS specific capabilities that are being added to node.js development. We will see how to create language translatable text strings and HANA database queries from node.js, we will also use the XSJS compatibility library to run existing XSJS artifacts without any changes.

Exercise Description

* Text bundles
* SAP HANA database access from node.js
* XSJS compatibility

Exercise 4 – Solution

## Exercise 4.1: Text Bundles

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. Return to the D:\Files\Session\DEV162\apps folder in Windows Explorer. Create a new folder named Exercise4 for all the development objects in this exercise. |  |
| 1. One of the libraries that SAP supplies for XS Node.js development is to help with the processing of text bundles. This allows you to output text strings in multiple languages.   From the Solutions/Exercise4 folder of the Students share, please copy the node\_modules folder and the files textBundleTemplate.js, messages.properties, messages\_de.properties, and messages\_ja.properties into your local Exercise4 folder.   messages.properties is your default message. The \_de and \_ja files have the same message but translated to German and Japanese respectively.   **Please note** this exercise has many dependent modules in the node\_modules folder and its normal that it will take 2 minutes or so to copy over. |  |
| 1. Rename the textBundleTemplate.js file to textBundle.js. It has a shell of an implementation with a helper function to get the locale from the HTTP request already implemented for you.   Using what you’ve already learned, implement an HTTP handler for the root URL using **express**. In the processing of this handler use the **TextBundle** library. When creating a new **TextBundle** instance the input are **path** (value messages to point to your message.properties file) and **locale** which should call **getLocale** passing in the **req** object. Then write into the **res** object with the **TextBundle** **getText** function (passing in the text ID of greeting and two parameters of **os.hostname(**) and **os.type().**   Try to write this code on your own but if you get stuck refer to the solution on the Student share. |  |
| 1. Switch to the Exercise4 folder and test your textBundle.js from the node.js command prompt.   The web browser should open and display the default English text string. |  |
| 1. In order to test the translated strings, go into Internet Explorer /Internet Options. Choose the Languages button. Then click Add. |  |
| 1. Add German and Japanese. |  |
| 1. Use the Move up button to raise German [de] to the top of the list. |  |
| 1. Refresh the web browser and you should now see the German text. |  |
| 1. Repeat the process raising Japanese [ja] to the top of the list and refresh the web page. |  |
| 1. Please return the browser settings to English at the top so as to not disrupt later courses that will use this room.   Also don’t forget to return the command prompt and use Ctrl+C to terminate the service when you are finished with this exercise. |  |

## Exercise 4.2: HANA Database Access

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. In this exercise we will look at how to use the HANA database access library to send queries to the database.   From the Solutions/Exercise4 folder of the Students share, please copy the files default-services.json and partnerListTemplate.js into your local Exercise4 folder. |  |
| 1. Rename the partnerListTemplate.js file to partnerList.js. It is actually the complete implementation of a service that calls the query 'select \* from "sap.hana.democontent.epmNext.data::MD.BusinessPartner" where PARTNERROLE = ? in the underlying HANA database.   The function **readTables** contains all the most interesting code. We create the HANA client and pass in connection details. These details are read from the default-services.json file. In the final version of XS with node.js integrated; these settings can also be configured centrally in a service broker instead.   The actual database code is very similar to JDBC or XSJS coding. There is a connection, prepared statement and query execution. The result set which is return is a JSON object (just like XSJS $.hdb interface).   The major difference is the structure of callbacks for the steps described above due to the asynchronous nature of node.js. |  |
| 1. Test your partnerList.js from the node.js command prompt.   The web browser should open. Unfortunately the default web browser is Internet Explorer and it doesn’t display JSON output. It will try and download the output. You are welcome to download the file and view it locally.   Or you can open Chrome to <http://localhost:3000/partners> and display the JSON content directly. Either way you will see a list of all business partners stored in this HANA database.  Don’t forget to return to the command prompt and use Ctrl+C to terminate the service when you are finished with this exercise. |  |
| 1. (OPTIONAL): As an optional part of this exercise adjust your partnerList.js to use what you have learned about the **async** library previously.   This library contains a function called **waterfall**. It allows you to rewrite such nested callbacks as the database connect/prepare/query/disconnect in a way that is more human readable.   You pass an array of functions for each operation. The callbacks are then linked in a chain instead of nested. Here is a screenshot of the first two operations in the waterfall chain to get you started. The complete implementation is also in the Solutions folder on the Student share.  . |  |

## Exercise 4.3: XSJS compatibility

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. One of the most exciting features of node.js on XS is the XSJS backward compatibility. We have the ability to run most XSJS applications without any changes directly from within node.js.   From the Solutions/Exercise4 folder of the Students share, please copy the folder src into your local Exercise4 folder. |  |
| 1. This folder has a series of XSJS, XSJSLIB, and XSHTTPDEST files. We’ve taken these directly from a HANA system without making any real changes to them.   The only change we made was to shorten the package paths which was purely to make this exercise easier to study and not a technical requirement. |  |
| 1. Back in your Exercise4 folder; create a new file called xsjs.js. This will be the node.js starting point that initializes the xsjs library and gives us an access point to then call through to the specific xsjs artifacts. |  |
| 1. Add the following logic. It brings in the necessary base modules and SAP specific modules that we need. Then we start the XSJS service handler directing it to look in the src folder for our XSJS content. We also set index.xsjs as our default target if nothing is specified by the user. The rest of the code is similar to earlier exercises where we open the web browser for testing. | 'use strict';  var os = require('os');  var path = require('path');  var xsjs = require('xsjs');  var xsenv = require('xsenv');  var opn = require('opn');  var port = process.env.PORT || 3000;  var options = xsjs.extend({  rootDir: path.join(\_\_dirname, 'src'),  redirectUrl: '/index.xsjs',  port: port  }, xsenv.getServices());  xsjs(options).listen(port);  console.log('Using HANA on %s:%d', options.hana.host, options.hana.port);  console.log('Server running at http://' + os.hostname() + ':' + port);  opn('http://' + os.hostname() + ':' + port); |
| 1. Test your xsjs.js from the node.js command prompt.   The web browser should open. You will be prompted for authorization to the HANA database.   The user id should be DEV162\_<group number>. Your group number will be given to you by the session instructor or displayed upon a card on your desk. For example if the group number is “01”, then your user id would be DEV162\_001.   Enter the password: Welcome15 |  |
| 1. If you authentication successfully, you should see the XS welcome screen. |  |
| 1. Navigate to /hello.xsjs and you should see the following output. |  |
| 1. Go to hello.xsjs in the src folder and make some change to the output text. Save the file. |  |
| 1. Stop the running xsjs.js service with Ctrl+C and then restart it to pick up the change you just made. Refresh the web browser and you should see your new text. |  |
| 1. Feel free to experiment with the other XSJS examples in your src folder.   Don’t forget to return the command prompt and use Ctrl+C to terminate the service when you are finished with this exercise. |  |

Exercise 5 –Web Sockets

Objective

Our final exercise will demonstrate the ease at which you can tap into the powerful web sockets capabilities of node.js. We will use web sockets to build a simple chat application. Any message sent from the SAPUI5 client side application will be propagated by the server to all listening clients.

Exercise Description

* Web Sockets chat application

Exercise 5 – Solution

## Exercise 5.1: Web Sockets

|  |  |
| --- | --- |
| Explanation | **Screenshot** |
| 1. Return to the D:\Files\Session\DEV162\apps folder in Windows Explorer. Create a new folder named Exercise5 for all the development objects in this exercise.. |  |
| 1. One of the often requested features of XS has been Web Sockets support. Luckily this is something that node.js comes with and the asynchronous programming is particularly well suited to the very idea of Web Sockets.   In this exercise we will create a small chat application using Web Sockets.   From the Solutions/Exercise5 folder of the Students share, please copy the node\_modules and html folders into your local Exercise5 folder.   . |  |
| 1. The HTML folder contains an SAPUI5 application to display a chat text box and input field to send a new message to the chat participants. Although not central to the topic of this workshop, you are welcome to have a look at the implementation of the client side.   SAPUI5 makes web sockets very easy from the client side as there is a library which deals with most of the details. |  |
| 1. We are more interested in the server side, node.js implementation part, however.  For that create a new file named chatServer.js in the root of your Exercise5 folder. |  |
| 1. Using what you’ve already learned, add code to the chatServer.js that will use **express** to serve the static content from the html directory on the **process.env.PORT** or 3000.   The library for Web Sockets which we are going to use is **ws**. Require it and create a new instance of the **WebSocketServe**r on port 3080 (that’s what our UI part is expecting).  This is then a recommended implementation for the remainder of the Web Sockets functionality to both receive and send messages. | wss.broadcast = function (data) {  for (var i in this.clients)  this.clients[i].send(data);  console.log('sent: %s', data);  };  wss.on('connection', function (ws) {  ws.on('message', function (message) {  console.log('received: %s', message);  wss.broadcast(message);  });  ws.send(JSON.stringify({  user: 'XS',  text: 'Hello from Node.js XS Server'  }));  }); |
| 1. The last part is to adjust the normal **opn** command. Since our server is running on each desktop we can’t chat with each other. We can however open two web browsers and chat with ourselves. Therefore issue two **opn** commands, force the use of Chrome instead of IE and use **the –new-window** command so each instances opens in its own window instead of two tabs. | opn('http://localhost:'+ httpPort, {app: ['chrome', '--new-window']});  opn('http://localhost:'+ httpPort, {app: ['chrome', '--new-window']}); |
| 1. If you get stuck, the complete implementation is of course available in the Solutions folder on the Student share. |  |
| 1. Change to the Exercise5 folder and test your chatServer.js from the node.js command prompt.   Two web browsers should open. |  |
| 1. Anything you type into either window is sent to the server and then pushed out to all listeners. If you want to test further open more than two browser windows. |  |
| 1. Don’t forget to return the command prompt and use Ctrl+C to terminate the service when you are finished with this exercise. |  |



© 2015 SAP SE or an SAP affiliate company. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or for any purpose without the express permission of SAP SE or an SAP affiliate company. SAP and other SAP products and services mentioned herein as well as their respective logos are trademarks or registered trademarks of SAP SE (or an SAP affiliate company) in Germany and other countries.   
Please see [http://www.sap.com/corporate-en/legal/copyright/index.epx#trademark](http://www.sap.com/corporate-en/legal/copyright/index.epx%23trademark) for additional trademark information and notices.