



Introduction to Node.js

HBD260

Exercises / Solutions
Thomas Jung, SAP Labs, LLC.



TABLE OF CONTENTS

BEFORE YOU START	3
Getting Help	3
Source Code Solutions	3
EXERCISE 1.....	4
Exercise 1.1: Node.js Module - Hello World.....	4
Exercise 1.2: Clone a Repository from Git.....	11
Exercise 1.3: Alternative: Import Project from File System	14
EXERCISE 2.....	17
Exercise 2.1: XSJS and XSODATA	17
Exercise 2.2: Debugging XSJS or Node.js.....	25
Exercise 2.3: Exploring JavaScript Language Features.....	27
Exercise 2.4: Creating a Simple OData Service.....	32
Exercise 2.5: Creating an OData Service with an Entity Relationship	36
Exercise 2.6: Creating an OData Service with Create Operation and XSJS Exit	38
EXERCISE 3.....	41
Exercise 3.1: Modules and Express.....	42
Exercise 3.2: HANA Database Access from Node.js	45
Exercise 3.3: Asynchronous Non-Blocking I/O	56
Exercise 3.4: Exploring JavaScript Language Features.....	65
Exercise 3.5: Text Bundles	71
Exercise 3.6: Open Source Modules.....	75
Exercise 3.7: Web Sockets	86

BEFORE YOU START

System Host: wdfibmt0749.wdf.sap.corp

System Instance Number: 00

System User ID: HBD260

All Passwords: Welcome17

XSA Organization: XS

XSA Development Space: DEV

Getting Help

If you need addition help resources beyond this document, we would suggest the following content:

- The Online Help at <https://help.sap.com/viewer/4505d0bdaf4948449b7f7379d24d0f0d/2.0.01/en-US>

Source Code Solutions

All source code solutions and templates for all exercises in this document can be found in the following webpage.

<https://github.com/l809764/TechEd2017.HBD260/tree/snippets/>

In some cases, it might be a little easier to copy and paste the coding instead of typing it all manually. If copying/pasting, I would suggest that you make sure to understand what is being copied/pasted before moving on.

Open the browser and enter the following URL to access the solutions web page (or you can use the bookmark shortcut we have created for you in Google Chrome). You can access the source code for each exercise by clicking on the appropriate exercise in the navigation bar.

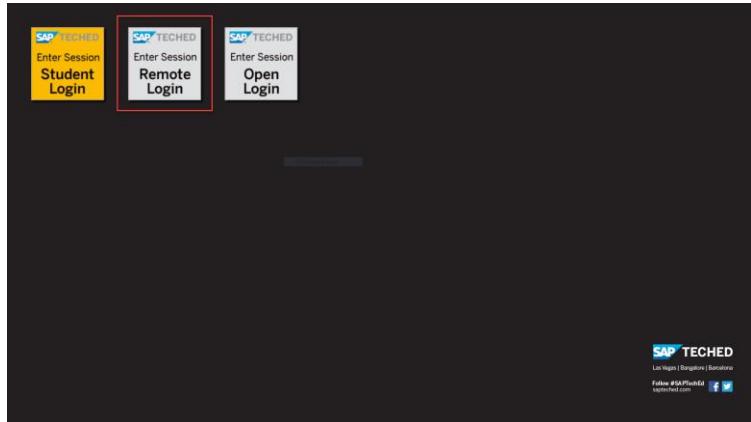
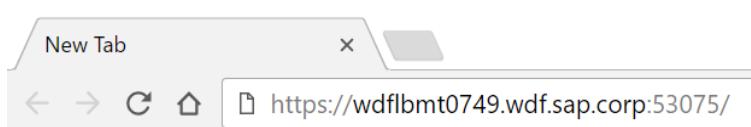
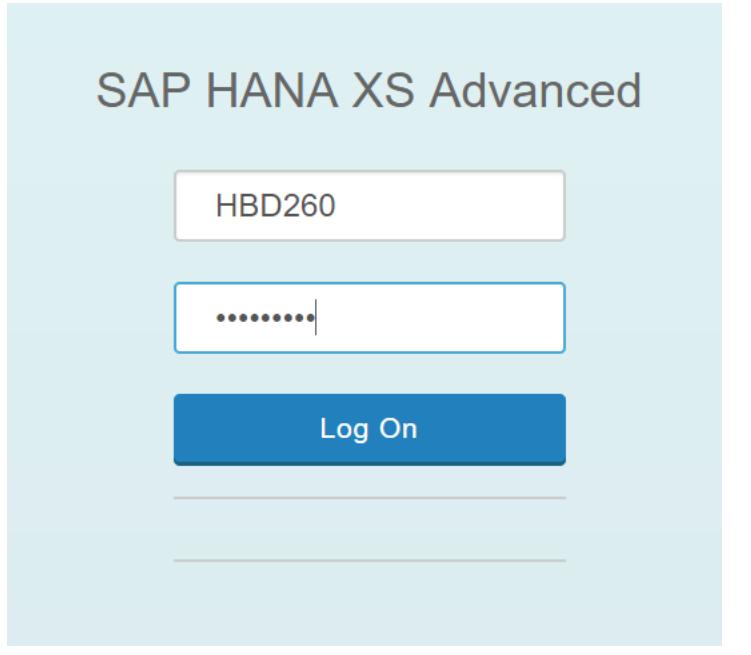
<https://github.com/l809764/TechEd2017.HBD260/tree/snippets/>



EXERCISE 1

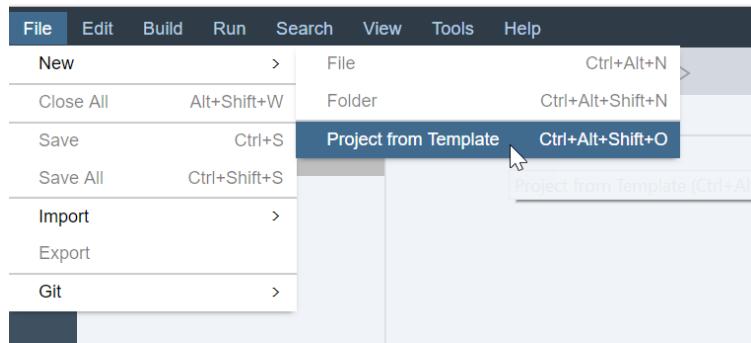
In this first exercise, we will connect to the SAP Web IDE for SAP HANA, run the new project wizard, and then create a Node.js module. At the end of this exercise you will be able to connect to your server via web browser and see a Hello World message from your Node.js service.

Exercise 1.1: Node.js Module - Hello World

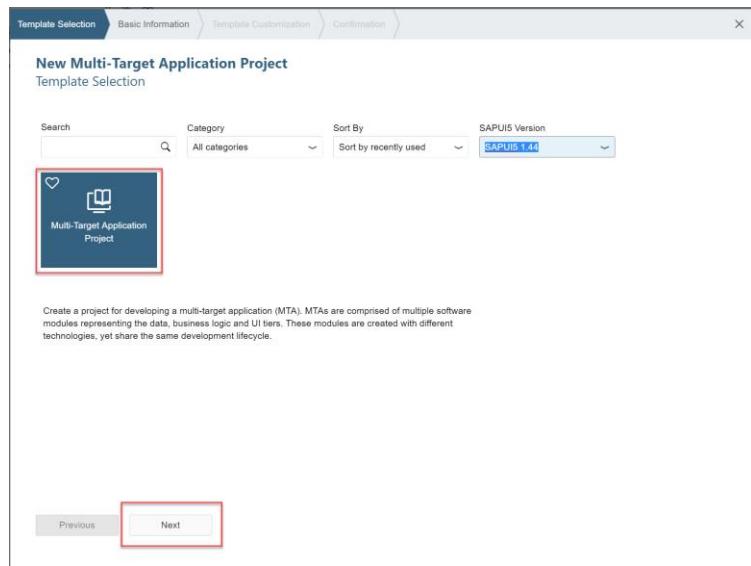
Explanation	Screenshot
<p>1. Please click on the Remote Login tile. You will be asked for username and password – please enter: .\student with password: Welcome17</p>	
<p>2. Launch the Google Chrome browser, and enter the following URL. https://wdflbmt0749.wdf.sap.corp:53075/</p>	
<p>3. User: HBD260 Password: Welcome17</p>	



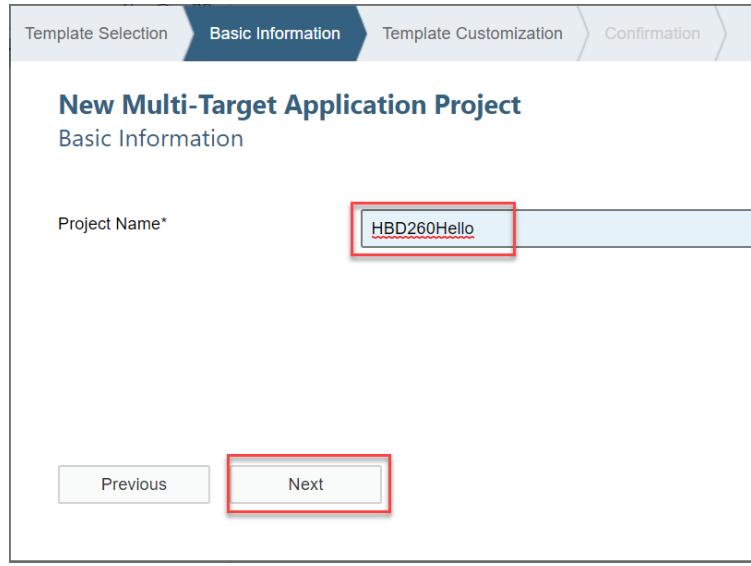
4. We will begin by creating a new project from template. Use **File->New->Project from Template**



5. Choose Multi-Target Application Project and then press **Next**.



6. Enter the project name HBD260Hello. Press **Next**.





7. You can optionally enter a description for your new application. Set the Space to DEV then press **Next**.

Template Selection Basic Information **Template Customization** Confirmation

New Multi-Target Application Project
Template Customization

MTA Details

Application ID*	HBD260Hello
Application Version*	0.0.1
Description	
Space*	DEV

Previous **Next** Finish

8. Press **Finish** to complete the wizard and generate your new project.

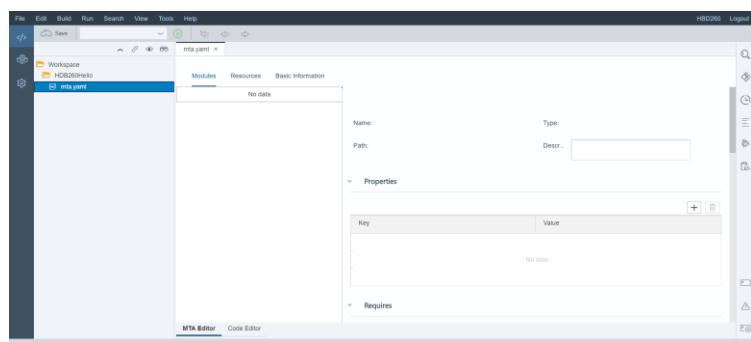
Template Selection Basic Information Template Customization **Confirmation**

New Multi-Target Application Project
Confirmation

Click Finish. A new project named HDB260Hello will be created in your workspace.

Previous **Finish**

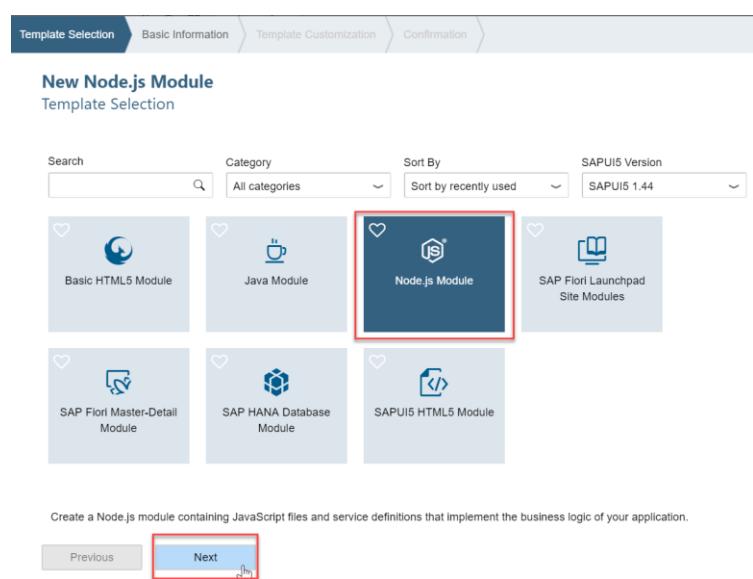
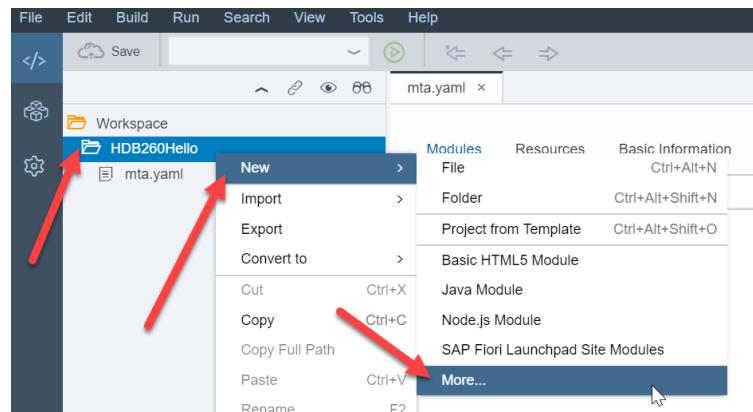
9. Your empty project has been created.



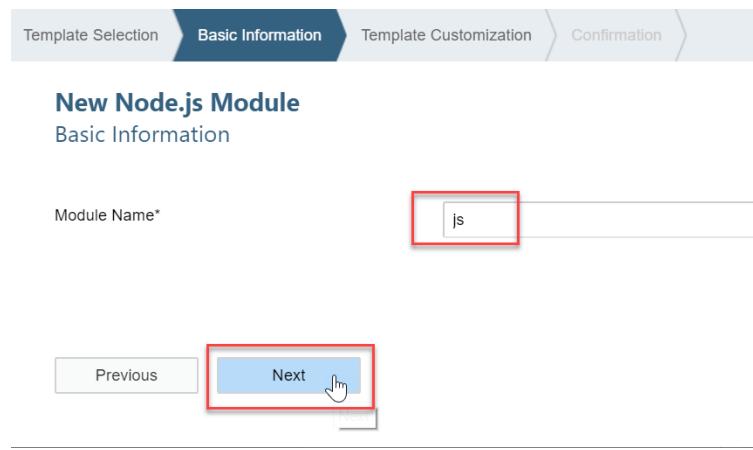
10. Next we need to create the Node.js module.

Begin by selecting your project and then choosing **New -> More**

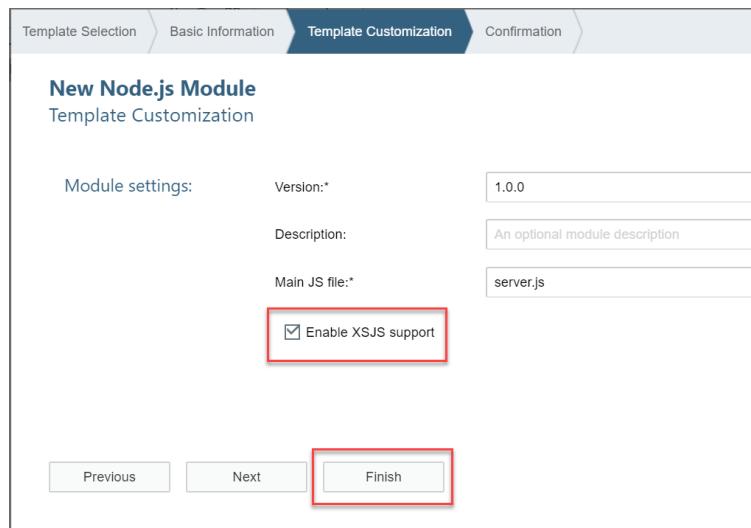
Then choose **Node.js Module** from the Template Selection and then choose **Next**.



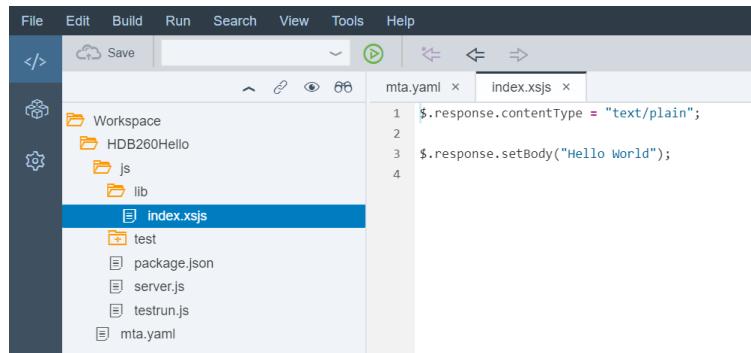
11. Name the module **js**. Press **Next**.



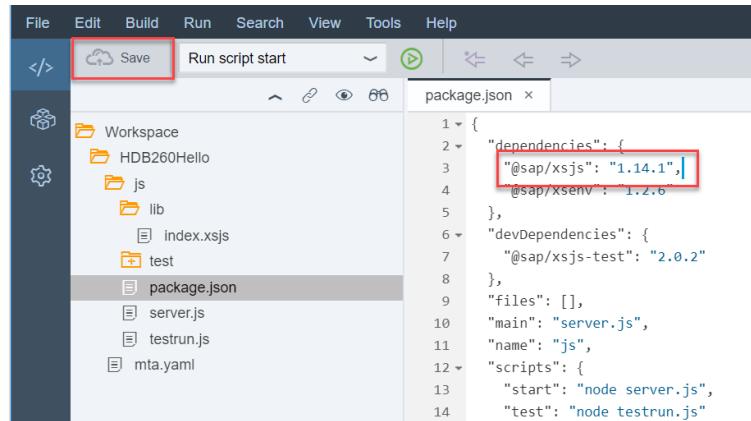
12. Select **Enable XSJS support** and then press **Finish**.



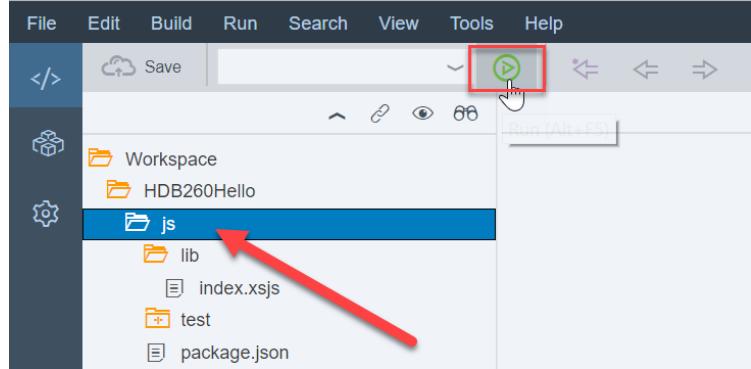
13. The Node.js module wizard created a folder structure starting with **js**. Expand **js** and the **lib** folder and double click on the **index.xsjs** file. This is our hello world service which was generated for us by the wizard.



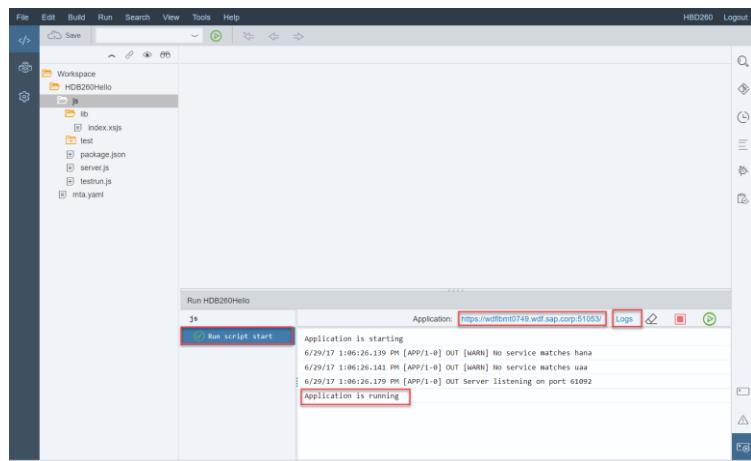
14. One manual update we need to make is the **package.json** file in the **/js** folder. Change the version of **@sap/xsjs** from 1.13.1 to 1.14.1. Save the file after the edit.



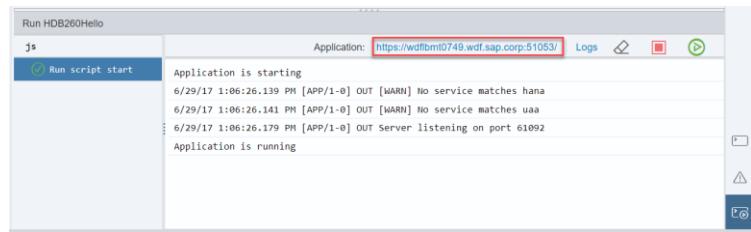
15. Our initial development is done and we are ready to deploy our application onto the XS Advanced server. Highlight the **js** folder and press **Run**. This will perform a build, then deploy the service onto the server. If successful it will open a new browser tab to the default page of this Node.js service.



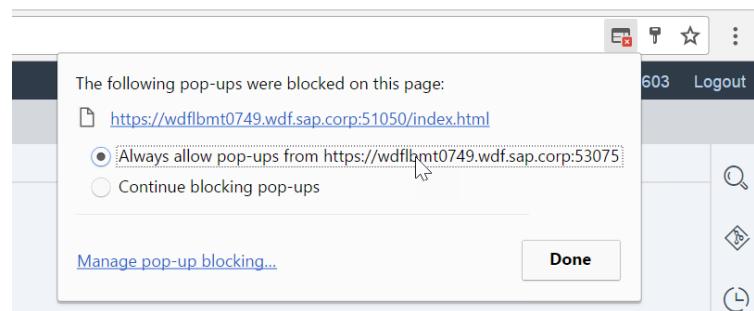
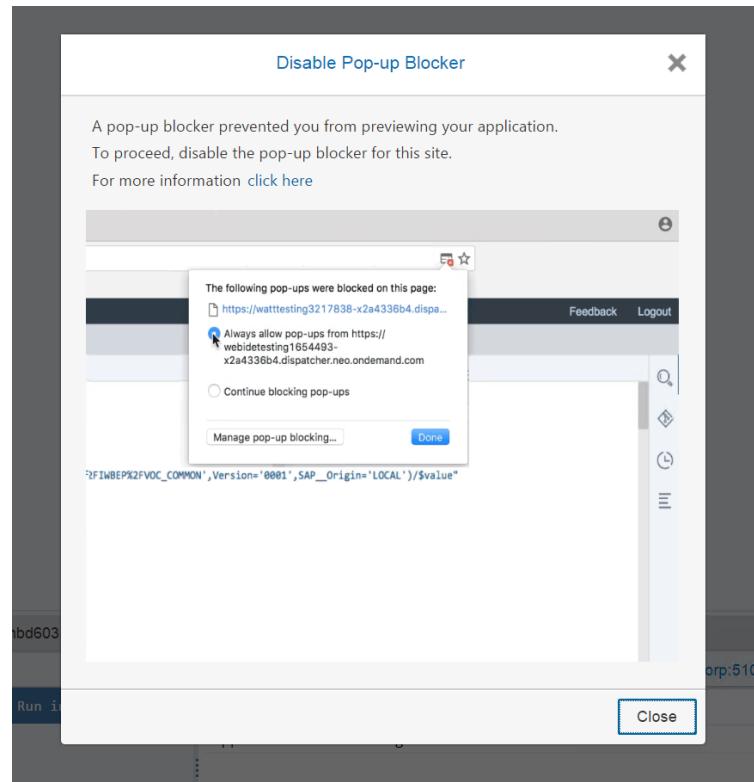
16. It can take a minute or two for the first build/deploy/run operation to complete. Upon completion, you should see that the service status has changed to Running and there is a hyper link to the application URL and the logs.



17. Click the hyperlink for the application URL in the Run window to open it in a new browser tab. Please note the port number for the URL shown in the screenshots might be different than what you see in your system since these ports are handed out at runtime.

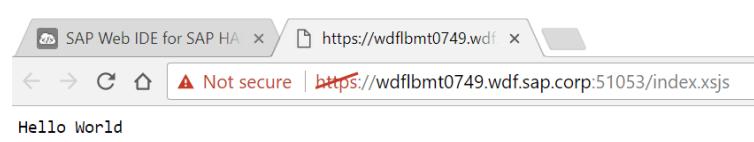


18. You might receive a message about the browser pop-up blocker. If you do receive this message press **Close**. Then click the pop-up configuration button on the right side of the browser address bar. Choose the **Always allow pop-ups from ...** option.



19. Depending upon the amount of time since you began the exercise you might be asked to re-authentication. After successful authentication (if needed), you should see your **index.xsjs** with the Hello World message.

Congratulations! You just wrote your first XS Advanced / Node.js application.

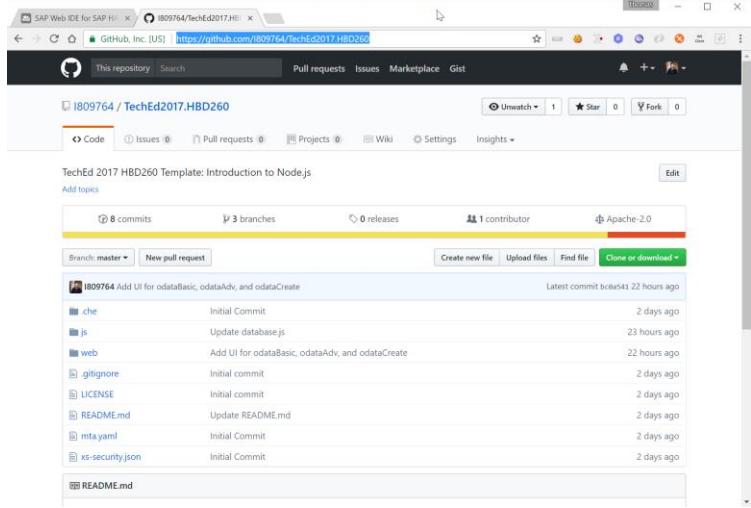




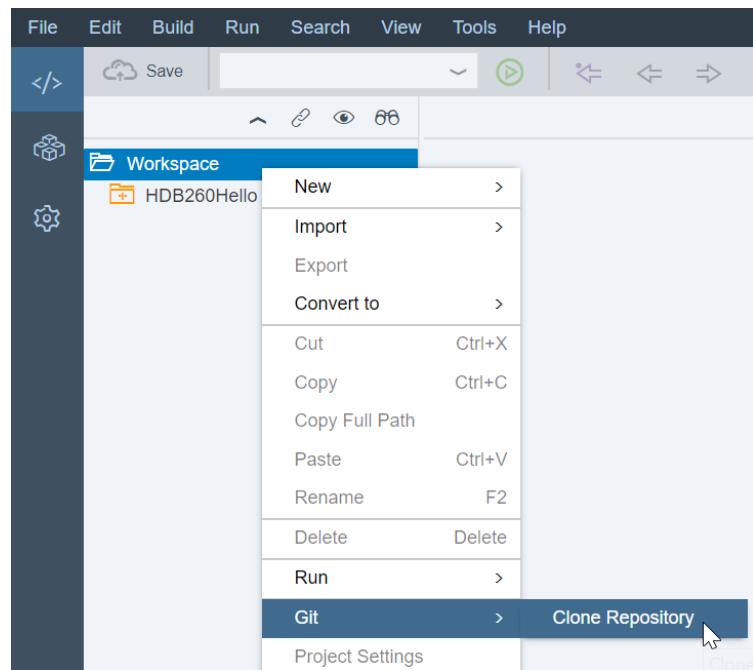
Exercise 1.2: Clone a Repository from Git

With XSA/HDI based development we no longer use the HANA database as the design time repository. We use a central Git to store all design time artifacts. Therefore, when you launch the Web IDE for SAP HANA, you only see the projects which have been pulled from Git or which you have created locally in your workspace. There is no repository browser. In this exercise, we will see how we can clone a project from Git to bring it into our local SAP Web IDE for SAP HANA workspace. You can alternatively import the project from the file system which will be shown in Exercise 1.3.

By cloning a skeleton project, we can setup all the basic configuration and add some test SAPUI5 applications allowing the remainder of the workshop to focus on JavaScript and Node.js coding.

Explanation	Screenshot																														
<p>1. In a new browser tab, please navigate to this URL: https://github.com/l809764/TechEd2017.HBD260</p> <p>This is the skeleton project for this course which we will use to get started with the remainder of the exercises.</p>	 <p>The screenshot shows a GitHub repository page for 'TechEd2017.HBD260'. The repository has 8 commits, 3 branches, 0 releases, and 1 contributor. The 'Clone or download' button is highlighted. The commits listed are:</p> <table border="1"><thead><tr><th>Author</th><th>Commit Message</th><th>Time Ago</th></tr></thead><tbody><tr><td>che</td><td>Initial Commit</td><td>2 days ago</td></tr><tr><td>js</td><td>Update database.js</td><td>23 hours ago</td></tr><tr><td>web</td><td>Add UI for odataBasic, odataAdv, and odataCreate</td><td>22 hours ago</td></tr><tr><td>.gitignore</td><td>Initial commit</td><td>2 days ago</td></tr><tr><td>LICENSE</td><td>Initial commit</td><td>2 days ago</td></tr><tr><td>README.md</td><td>Update README.md</td><td>2 days ago</td></tr><tr><td>mta.yaml</td><td>Initial Commit</td><td>2 days ago</td></tr><tr><td>xs-security.json</td><td>Initial Commit</td><td>2 days ago</td></tr><tr><td>README.md</td><td>(empty)</td><td>(empty)</td></tr></tbody></table>	Author	Commit Message	Time Ago	che	Initial Commit	2 days ago	js	Update database.js	23 hours ago	web	Add UI for odataBasic, odataAdv, and odataCreate	22 hours ago	.gitignore	Initial commit	2 days ago	LICENSE	Initial commit	2 days ago	README.md	Update README.md	2 days ago	mta.yaml	Initial Commit	2 days ago	xs-security.json	Initial Commit	2 days ago	README.md	(empty)	(empty)
Author	Commit Message	Time Ago																													
che	Initial Commit	2 days ago																													
js	Update database.js	23 hours ago																													
web	Add UI for odataBasic, odataAdv, and odataCreate	22 hours ago																													
.gitignore	Initial commit	2 days ago																													
LICENSE	Initial commit	2 days ago																													
README.md	Update README.md	2 days ago																													
mta.yaml	Initial Commit	2 days ago																													
xs-security.json	Initial Commit	2 days ago																													
README.md	(empty)	(empty)																													

2. Return to the SAP Web IDE for SAP HANA. Right mouse click on the Workspace and choose Git->Clone Repository.

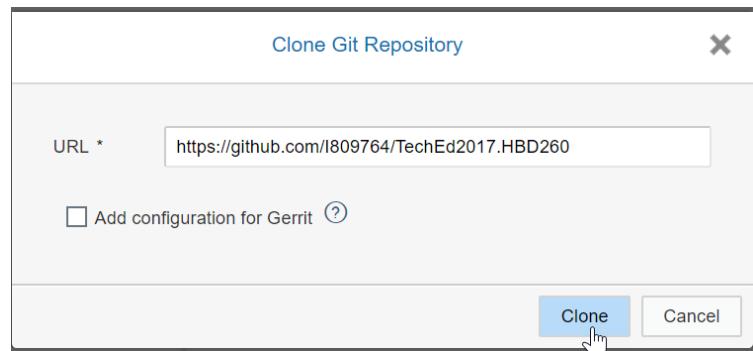


3. In the Clone Repository dialog, please use the following values.

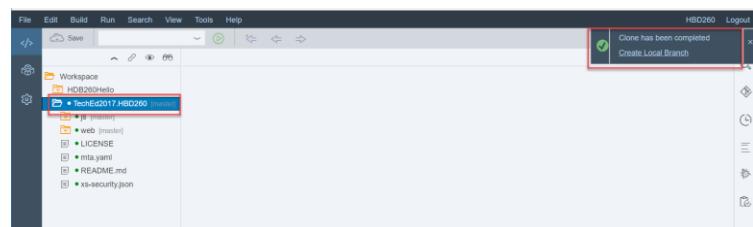
URL:

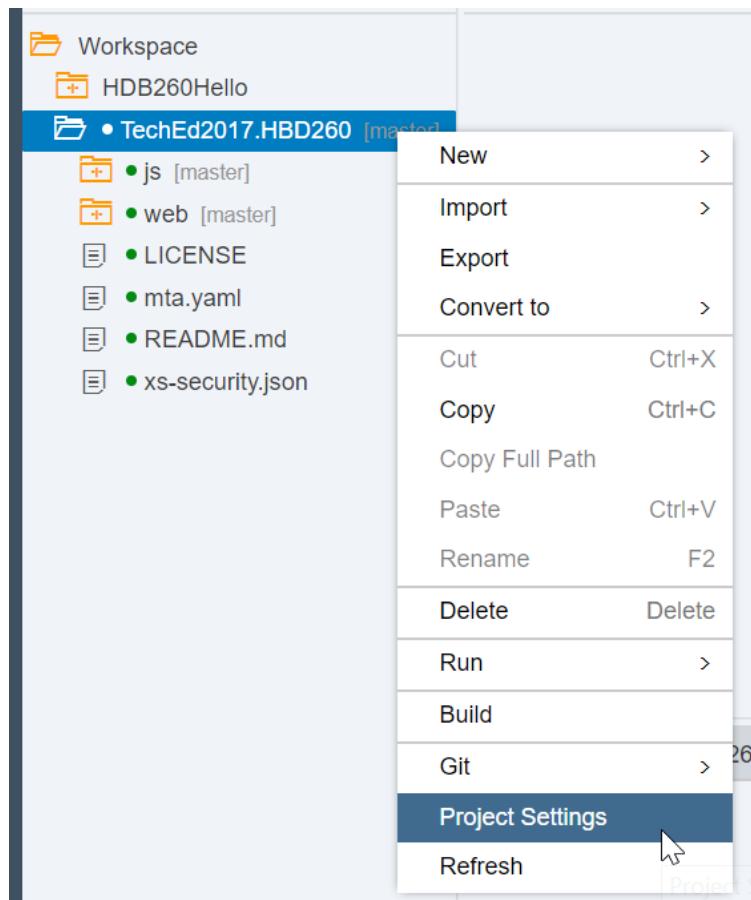
<https://github.com/l809764/TechEd2017.HBD260>

Press **Clone**.

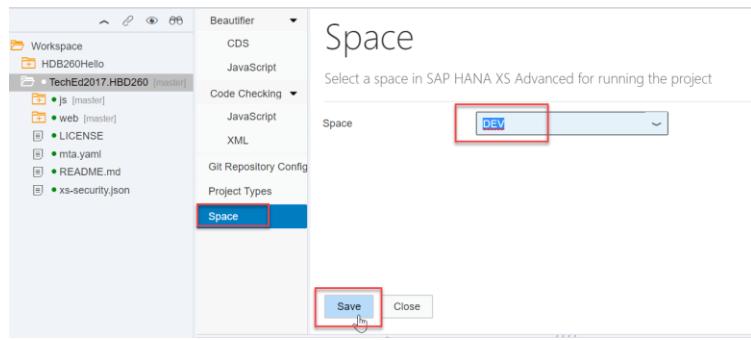


4. If successful, the TechEd2017.HBD260 project will now appear in your workspace and is connected to the git repository.





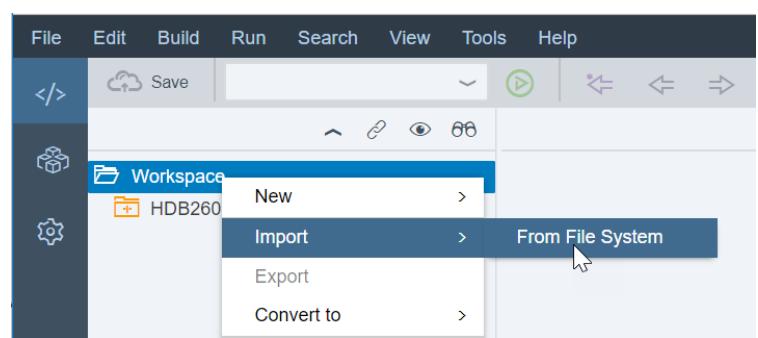
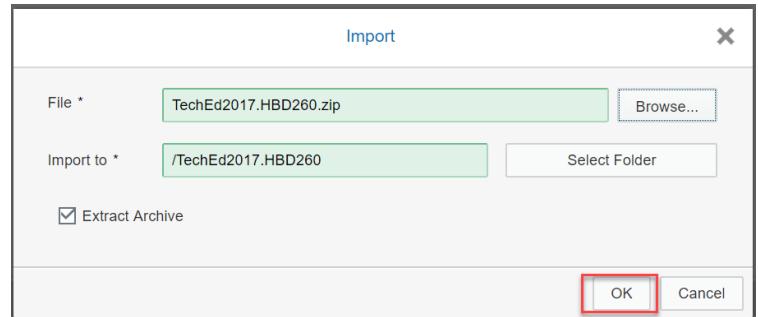
5. Right mouse click on your project and choose **Project Settings**.



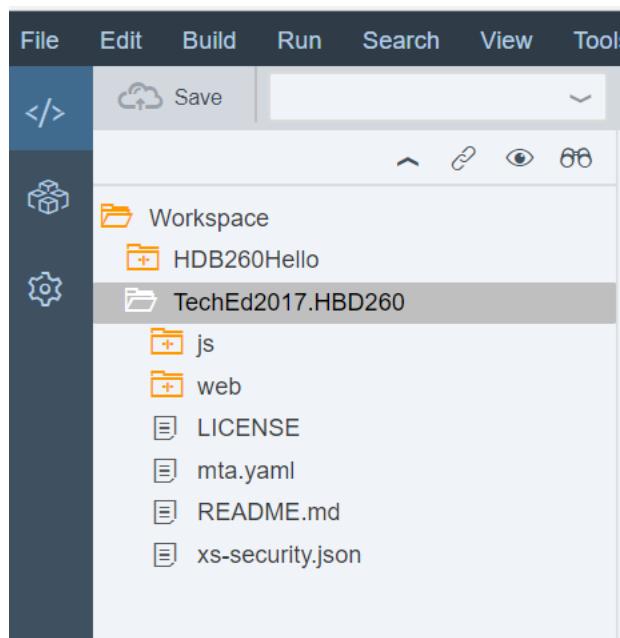
6. Choose the Space section and then choose DEV as the Development Space. Press **Save**.
If you have reached this step you can move ahead to Exercise 2. If you were not able to complete it for whatever reason, please move onto Exercise 1.3.



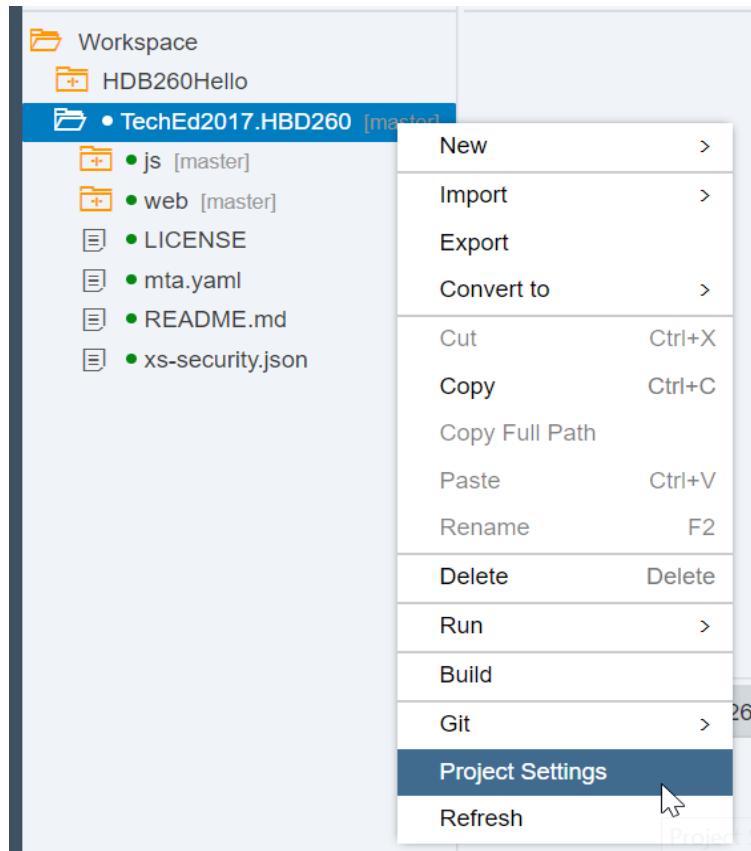
Exercise 1.3: Alternative: Import Project from File System

Explanation	Screenshot
<p>1. Download a zip file with the complete project contents from the following location: Z:\HBD260\</p>	
<p>2. Return to the SAP Web IDE for SAP HANA. Right mouse click on the Workspace and choose Import -> From File System.</p>	
<p>3. Click the browse button and from the File Open dialog choose the location where you download the file in step 1 of this exercise. Select the file and click Open. Press the OK button to import the template project.</p>	

-
4. If successful, the TechEd2017.HBD260 project will now appear in your workspace.

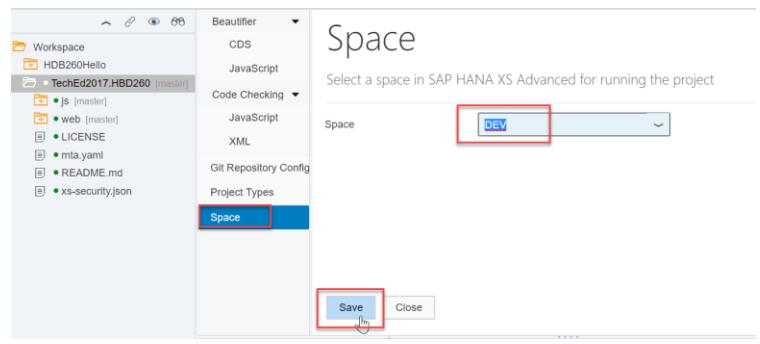


-
5. Right mouse click on your project and choose





-
6. Choose the Space section and then choose DEV as the Development Space. Press **Save**.



EXERCISE 2

For this exercise, we will now build the XSJS and XSODATA services used to expose our data model to the user interface. Although XS Advanced runs on Node.js, SAP has added modules to Node.js to provide XSJS and XSODATA backward compatibility. Therefore, you can use the same programming model and much of the same APIs from XS, classic even within this new environment.

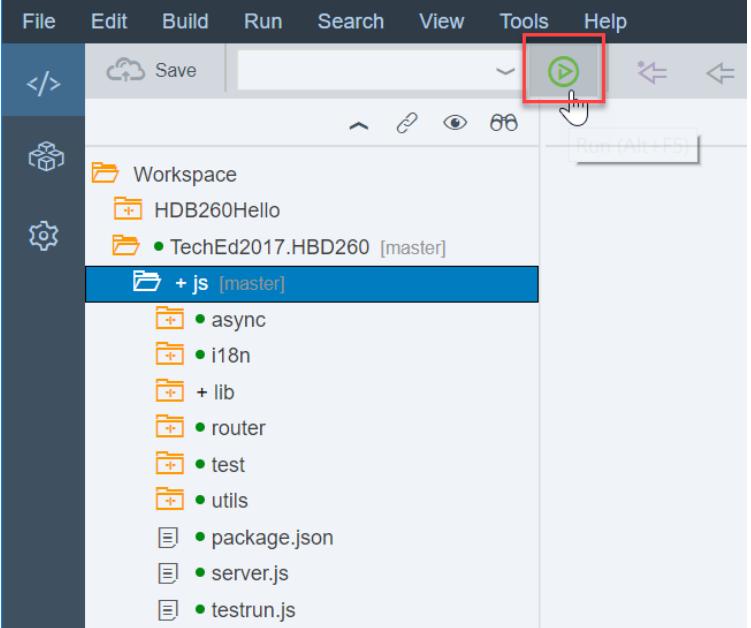
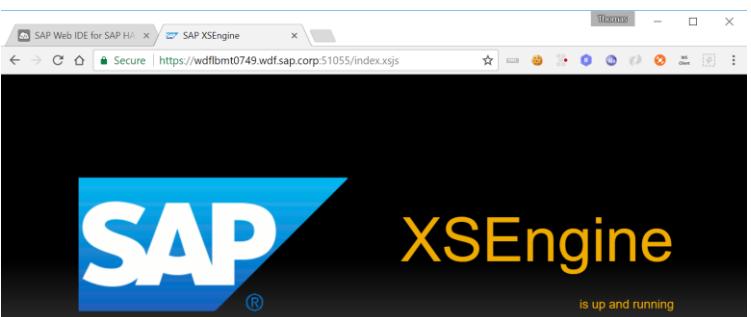
Exercise 2.1: XSJS and XSODATA

Explanation	Screenshot
<p>1. In the js/lib folder, create a sub-folder called xsodata. Create a file named purchaseOrder.xsodata. Here is the source code for this file.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/I809764/TechEd2017.HBD260/blob/snippets/ex2/ex2_1</p> <p>Here we expose both the Header and Item tables from our HDI container as separate entities and build a navigation association between the two.</p> <p>Note: If you receive a syntax error on Line 2, you can ignore this. It is a bogus error because our target tables do not use Namespaces.</p>	<pre>service { "PurchaseOrder.Header" as "POHeader" navigates ("Items" as "POItem"); "PurchaseOrder.Item" as "POItem"; association "Items" principal "POHeader"("PURCHASEORDERID") multiplicity "1" dependent "POItem"("PURCHASEORDERID") multiplicity "*"; }</pre>
<p>2. In the lib folder, create a sub-folder called xsjs. Create a file named hdb.xsjs. Here is the source code for this file.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/I809764/TechEd2017.HBD260/blob/snippets/ex2/ex2_2</p> <p>This logic reads data from our item table and sends it to the frontend as JSON.</p>	<pre>/*eslint no-console: 0, no-unused-vars: 0, dot-notation: 0*/ "use strict"; var conn = \$.hdb.getConnection(); var query = "SELECT FROM PurchaseOrder.Item { " + " POHeader.PURCHASEORDERID as \ \"PurchaseOrderId\", " + " PRODUCT as \"ProductID\", " + " GROSSAMOUNT as \"Amount\" " + " } "; var rs = conn.executeQuery(query); \$.response.setBody(JSON.stringify(rs)); \$.response.contentType = "application/json"; \$.response.status = \$.net.http.OK;</pre>

<p>3. Create a second file named exercisesMaster.xsjs. Here is the source code for this file.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex2/ex2_3</p> <p>It sends the current user and language back to the client for filling in the header of the UI. This shows how the \$ apis still work and have been adapted to the new Node.js environment.</p>	<pre>/*eslint no-console: 0, no-unused-vars: 0, dot-notation: 0*/ "use strict"; function fillSessionInfo(){ var body = ""; body = JSON.stringify({ "session" : [{"UserName": \$.session.getUsername(), "Language": \$.session.language}] }); \$.response.contentType = "application/json"; \$.response.setBody(body); \$.response.status = \$.net.http.OK; } var aCmd = \$.request.parameters.get("cmd"); switch (aCmd) { case "getSessionInfo": fillSessionInfo(); break; default: \$.response.status = \$.net.http.INTERNAL_SERVER_ERROR; \$.response.setBody("Invalid Request Method"); }</pre>
<p>4. Create a forth file named procedures.xsjs. This example shows you how to call a stored procedure from XSJS.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex2/ex2_4</p>	<pre>/*eslint no-console: 0, no-unused-vars: 0, dot-notation: 0, no-use-before-define: 0*/ "use strict"; /** * @function JSON as returned by hdb */ function hdbDirectTest(){ var results = _selection(); //Pass output to response \$.response.status = \$.net.http.OK; \$.response.contentType = "application/json"; \$.response.setBody(JSON.stringify(results)); } /** * @function Flattened JSON structure */ function hdbFlattenedTest(){ outputJSON(_selection().EX_TOP_3_EMP_PO_COMBINED_CNT); } /** * @function load/call the procedure */</pre>

	<pre> function _selection(){ var connection = \$.hdb.getConnection(); var getPOHeaderData = connection.loadProcedure("get_po_header_data"); var results = getPOHeaderData(); return results; } /** * @function Puts a JSON object into the Response Object * @param {object} jsonOut - JSON Object */ function outputJSON(jsonOut){ var out = []; for(var i=0; i<jsonOut.length;i++){ out.push(jsonOut[i]); } \$response.status = \$.net.http.OK; \$response.contentType = "application/json"; \$response.setBody(JSON.stringify(out)); } </pre> <pre> var aCmd = \$.request.parameters.get("cmd"); switch (aCmd) { case "direct": hdbDirectTest(); break; case "flattened": hdbFlattenedTest(); break; default: hdbDirectTest(); break; } </pre>
<p>5. Create a 5th file called os.xsjs. This example shows you how you can call to Node.js from XSJS using the \$.require API. We will learn more about Node.js modules, like os, in subsequent exercises.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017-HBD260/blob/blob/snippets/ex2/ex2_5</p>	<pre> /*eslint no-console: 0, no-unused-vars: 0*/ "use strict"; var os = \$.require("os"); var output = {}; output.tmpdir = os.tmpdir(); output.endianness = os.endianness(); output.hostname = os.hostname(); output.type = os.type(); output.platform = os.platform(); output.arch = os.arch(); output.release = os.release(); output.uptime = os.uptime(); output.loadavg = os.loadavg(); </pre>

	<pre> output.totalmem = os.totalmem(); output.freemem = os.freemem(); output.cpus = os.cpus(); output.networkInfraces = os.networkInterfaces(); \$.response.status = \$.net.http.OK; \$.response.contentType = "application/json"; \$.response.setBody(JSON.stringify(output)); </pre>
6. Create a 6 th file called whoAmI.xsjs . This example shows how we now use a different user at the DB and XS layers, but also how the XS user (Application User) can be accessed from within the DB by reading the Session Context. Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017-HBD260/blob/snippets/ex2/ex2_6	<pre> var connection = \$.hdb.getConnection(); var query = "SELECT CURRENT_USER FROM \"DUMMY\";"; var rs = connection.executeQuery(query); var currentUser = rs[0].CURRENT_USER; query = "SELECT SESSION_CONTEXT('APPLICATIONUSER') \"APPLICATION_USER\" FROM \"DUMMY\";"; rs = connection.executeQuery(query); var applicationUser = rs[0].APPLICATION_USER; var greeting = "XS Layer Session User: " + \$.session.getUsername() + "
Database Current User: " + currentUser + "
 Database Application User: " + applicationUser + "
Welcome to HANA "; \$.response.contentType = "text/html; charset=utf-8"; \$.response.setBody(greeting); </pre>
7. Create a 7 th file called whoAmI_SQLCC.xsjs . This example is very like the previous example but shows how we can still use the concept of SQLCC from XS Classic to force a different Database User. Here we are forcing our database connection to use the SQLCC named SQLCC_CONFIG. We no longer setup the SQLCC_CONFIG as a file, however. It is instead configured in the bootstrap of XSJS startup to point to an XSA User Provided Service for actual connection parameters. Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017-HBD260/blob/snippets/ex2/ex2_7	<pre> var connection = \$.hdb.getConnection({"sqlcc": "xsjs.sqlcc_config", "pool": true }); var query = "SELECT CURRENT_USER FROM \"DUMMY\";"; var rs = connection.executeQuery(query); var currentUser = rs[0].CURRENT_USER; query = "SELECT SESSION_CONTEXT('APPLICATIONUSER') \"APPLICATION_USER\" FROM \"DUMMY\";"; rs = connection.executeQuery(query); var applicationUser = rs[0].APPLICATION_USER; var greeting = "XS Layer Session User: " + \$.session.getUsername() + "
Database Current User: " + currentUser + "
 Database Application User: " + applicationUser + "
Welcome to HANA "; </pre>

	<pre><code>\$.response.contentType = "text/html; charset=utf-8"; \$.response.setBody(greeting);</code></pre>
8. Save any unsaved/open files. We can now run the js module. Like in Exercise 1, the first run of a new module will take a minute or two.	
9. You should see that the build and deploy was successful. The initial build can take a few minutes.	
10. So now run the web module. All our authentication and access will run through this module and act as a reverse proxy to the Node.js module.	
11. A new tab will open and you might be asked for authentication again (HBD260/Welcome17). It will load a small Hello World SAPUI5 page. We can change the url to our xsjs service /index.xsjs in the browser. You will see that our xsjs service is accessible via the HTML5 module runtime. The HTML5 module functions as a proxy and performs the routing to the other service internally.	

12./xsjs/hdb.xsjs reads data from our Purchase Order table and exports it as a JSON string (formatted on the client side by a browser plug-in).

```

[{"PurchaseOrderId": "500000000", "ProductId": "HT-1000", "Amount": "1137.64"}, {"PurchaseOrderId": "500000000", "ProductId": "HT-1091", "Amount": "61.88"}, {"PurchaseOrderId": "500000000", "ProductId": "HT-6100", "Amount": "1116.22"}, {"PurchaseOrderId": "500000000", "ProductId": "HT-1001", "Amount": "2275.28"}, {"PurchaseOrderId": "500000000", "ProductId": "HT-1092", "Amount": "92.82"}]
  
```

13. /xsjs/exercisesMaster.xsjs shows the typical approach to have multiple REST actions in one XSJS. Change the URL to **/xsjs/exercisesMaster.xsjs?cmd=getSessionInfo**. You should see your User Name returned from the XSJS session layer.

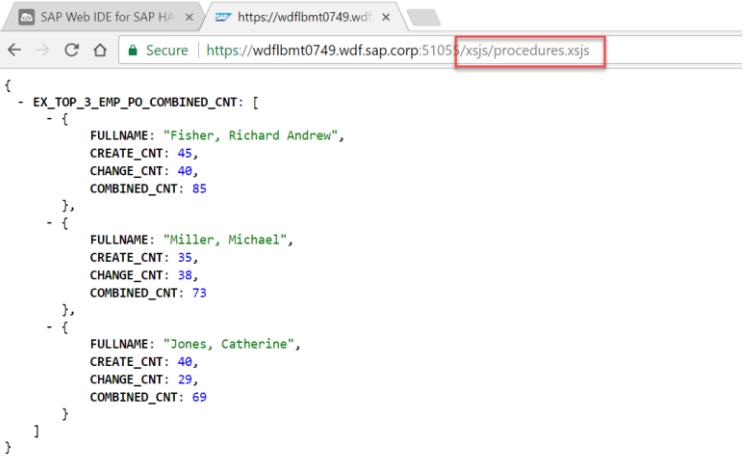
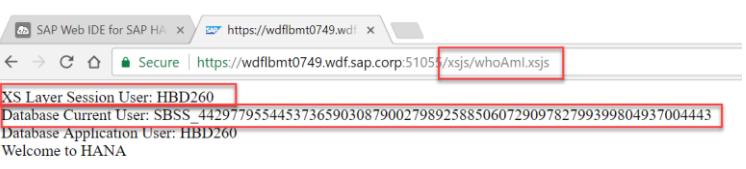
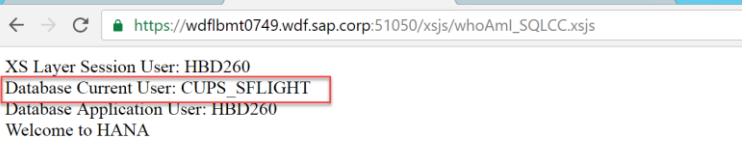
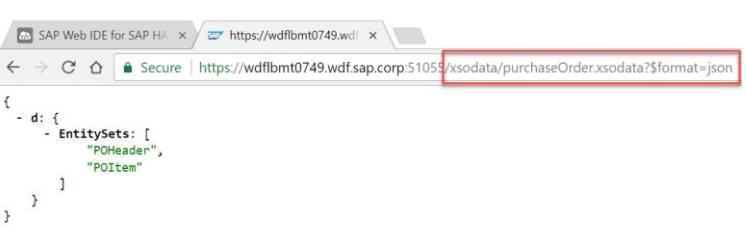
```

{
  "session": [
    {
      "UserName": "HBD260",
      "Language": ""
    }
  ]
}
  
```

14. /xsjs/os.xsjs demonstrates the new **\$.require** feature to access Node.js modules.

```

{
  "tmpdir: "/hana/shared/TDS/xs/app_working/wdfibmt0749/executionroot/b421ce35-b5fe-4be1-8574-c94d732dcf1d/app-tmp",
  endianness: "LE",
  hostname: "wdfibmt0749",
  type: "Linux",
  platform: "Linux",
  arch: "x64",
  release: "4.4.59-92.17-default",
  uptime: 45259,
  loadavg: [0.45740464375, 1.16740464375, 1.73607421875],
  totalmem: 33738297344,
  freemem: 498573312,
  cpus: [
    {
      model: "Intel(R) Xeon(R) CPU E7- 8870 @ 2.40GHz",
      speed: 2394,
      times: [
        {
          user: 43414000,
          nice: 0,
          sys: 22603500,
          idle: 371008100,
          irq: 0
        }
      ],
      {
        model: "Intel(R) Xeon(R) CPU E7- 8870 @ 2.40GHz",
        speed: 2394,
        times: [
          {
            user: 42165100,
            nice: 0
          }
        ]
      }
    }
  ]
}
  
```

<p>15. /xsjs/procedures.xsjs shows the results from calling the stored procedure, get_po_header_data.</p>	 <pre>{ "EX_TOP_3_EMP_PO_COMBINED_CNT": [{ "FULLNAME": "Fisher, Richard Andrew", "CREATE_CNT": 45, "CHANGE_CNT": 48, "COMBINED_CNT": 85 }, { "FULLNAME": "Miller, Michael", "CREATE_CNT": 35, "CHANGE_CNT": 38, "COMBINED_CNT": 73 }, { "FULLNAME": "Jones, Catherine", "CREATE_CNT": 48, "CHANGE_CNT": 29, "COMBINED_CNT": 69 }] }</pre>
<p>16. /xsjs/whoAmI.xsjs demonstrates the separation between application user and database user as well as how the application user can be accessed from both the XSJS and Database layers.</p>	 <p>XS Layer Session User: HBD260 Database Current User: SBSS_44297795544537365903087900279892588506072909782799399804937004443 Database Application User: HBD260 Welcome to HANA</p>
<p>17. And /xsjs/whoAmI_SQLCC.xsjs demonstrates how we can use User Provided Service to provide the same functioning as SQLCC and force a particular DB User (CUPS_SFLIGHT).</p>	 <p>XS Layer Session User: HBD260 Database Current User: CUPS_SFLIGHT Database Application User: HBD260 Welcome to HANA</p>
<p>18./xsodata/purchaseOrder.xsodata?\$format=json gives you access to a full OData service for the Purchase Order header and item tables we created in the previous exercise.</p>	 <pre>{ "d": { "EntitySets": ["POHeader", "POItem"] } }</pre>

19. We can also see the metadata about the OData interface we created by changing the URL to `/xsodata/purchaseOrder.xsodata/$metadata`

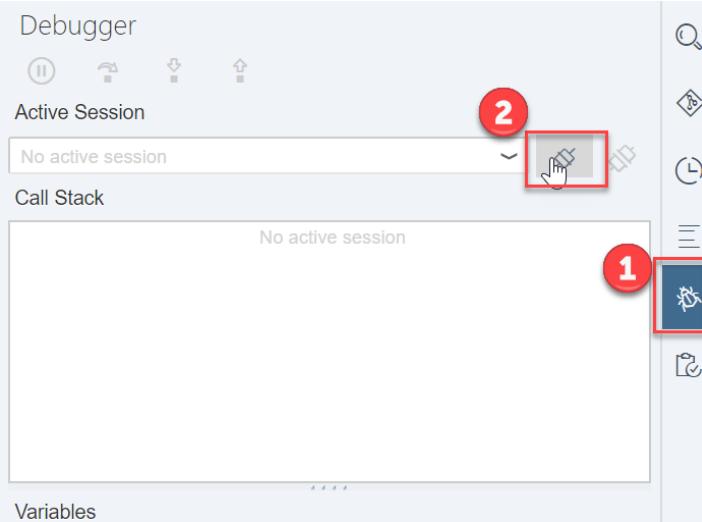
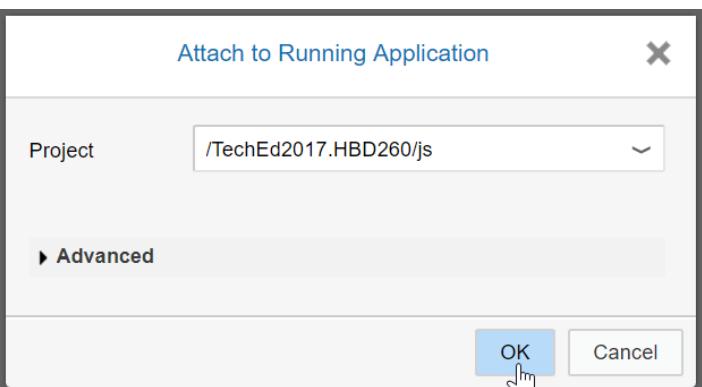
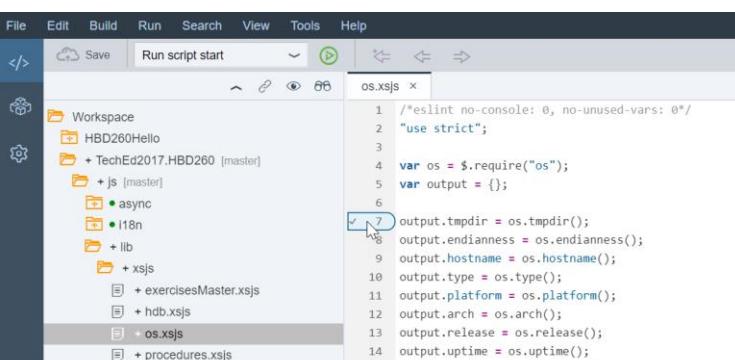
```
Click entity for XPath. Double-click to collapse/expand. Enter XPath or XML  
string, then click XPath1. Press F2 for results or to XML Treeify, respectively. Source Options  
https://wdflb0m0749.wdf.sap.corp:51055/xodata/purchaseOrder.xodata/$metadata  
  
<Entity Type="PurchaseOrder">  
  <Key>  
    <PropertyRef Name="PURCHASEORDERID"/>  
  </Key>  
  <Property Name="PURCHASEORDERID" Type="Edm.Int32" Nullable="false"/>  
  <Property Name="HISTORY.CREATEDBY" Type="Edm.String" MaxLength="10"/>  
  <Property Name="HISTORY.CREATEDAT" Type="Edm.DateTime"/>  
  <Property Name="HISTORY.CHANGEDBY" Type="Edm.String" MaxLength="10"/>  
  <Property Name="HISTORY.CHANGEMAT" Type="Edm.DateTime"/>  
  <Property Name="PARTNERID" Type="Edm.String" MaxLength="10"/>  
  <Property Name="PARTNER" Type="Edm.String" MaxLength="10"/>  
  <Property Name="CURRENCY" Type="Edm.String" MaxLength="3"/>  
  <Property Name="GROSSAMOUNT" Type="Edm.Decimal" Precision="15" Scale="2"/>  
  <Property Name="NETAMOUNT" Type="Edm.Decimal" Precision="15" Scale="2"/>  
  <Property Name="TAXAMOUNT" Type="Edm.Decimal" Precision="15" Scale="2"/>  
  <Property Name="LIFECYCLESTATUS" Type="Edm.String" MaxLength="1"/>  
  <Property Name="APPROVALSTATUS" Type="Edm.String" MaxLength="1"/>  
  <Property Name="CONFIRMSSTATUS" Type="Edm.String" MaxLength="1"/>  
  <Property Name="ORDERINGSTATUS" Type="Edm.String" MaxLength="1"/>  
  <Property Name="INVOICINGSTATUS" Type="Edm.String" MaxLength="1"/>  
  <Navigation PropertyName="POItem" Relationship="default.ItemsType" FromRole="POHeaderPrincipal" ToRole="POItemDependent"/>  
</EntityType>  
<EntityType Name="POItemType">  
  <Key>  
    <PropertyRef Name="POHeader.PURCHASEORDERID"/>  
    <PropertyRef Name="PRODUCT"/>  
  </Key>
```

20. Change the URL again to point to the **POHeader** entity and we will see the data details.
`/xsodata/purchaseOrder.xsodata/POHeader?$format=json`

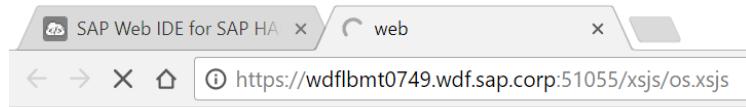
```
{ "d": { "results": [ { "metadata": { "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/purchaseOrder.xsodata/POHeader(50000000)", "type": "default.POHeaderType" }, "PURCHASEORDERID": "50000000", "HISTORY.CREATEDBY": null, "HISTORY.CREATEDAT": null, "HISTORY.CHANGEDBY": null, "HISTORY.CHANGEDAT": null, "NOTEID": "9000000001", "PARTNER": "6100000000", "CURRENCY": "EUR", "GROSSAMOUNT": "13224.47", "NETAMOUNT": "11113.00", "TAXAMOUNT": "2111.47", "LIFECYCLESTATUS": "N", "APPROVALSTATUS": "I", "CONFIRMSTATUS": "I", "ORDERINGSTATUS": "I", "INVOICINGSTATUS": "I", "POItem": { "deferred": { "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/purchaseOrder.xsodata/POHeader(50000000)/POItem(5000000001)" } }, "metadata": { "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/purchaseOrder.xsodata/POHeader(50000000)", "type": "default.POHeaderType" }, "PURCHASEORDERID": "50000001", "HISTORY.CREATEDBY": null, "HISTORY.CREATEDAT": null, "HISTORY.CHANGEDBY": null, "HISTORY.CHANGEDAT": null, "NOTEID": "9000000001", "PARTNER": "6100000000", "CURRENCY": "EUR", "GROSSAMOUNT": "13224.47", "NETAMOUNT": "11113.00", "TAXAMOUNT": "2111.47", "LIFECYCLESTATUS": "N", "APPROVALSTATUS": "I", "CONFIRMSTATUS": "I", "ORDERINGSTATUS": "I", "INVOICINGSTATUS": "I", "POItem": { "deferred": { "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/purchaseOrder.xsodata/POHeader(50000000)/POItem(5000000002)" } } ] } }}
```

- Finally, we can debug XSODATA requests if we add the URL parameter **sap-ds-debug** with either the value json or html. We can use this to see internal request, response, processing time, or even the generated SQL statements.

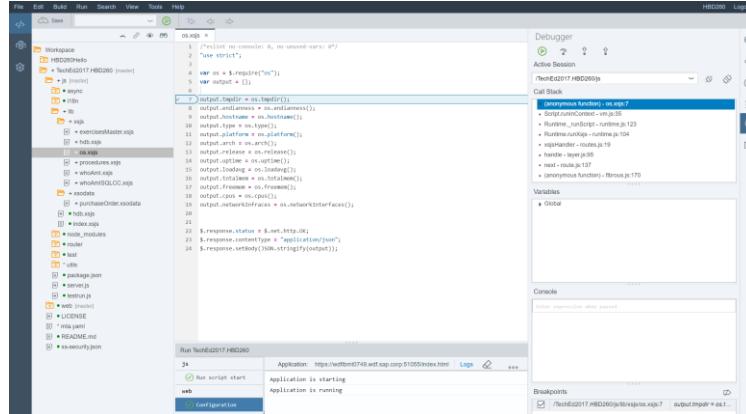
Exercise 2.2: Debugging XSJS or Node.js

Explanation	Screenshot
<p>1. Return to the SAP Web IDE for SAP HANA. Click the debug link in the left side of the SAP Web IDE for SAP HANA. Then click the Attach Debugger button.</p>	
<p>2. Confirm the module you want to debug and then press OK.</p>	
<p>3. Click in the left side of the editor of the os.xsjs file to set a breakpoint on line 7.</p>	 <pre data-bbox="715 1362 1450 1722"> 1 /*eslint no-console: 0, no-unused-vars: 0*/ 2 "use strict"; 3 4 var os = \$ require("os"); 5 var output = {}; 6 7 output.tmpdir = os.tmpdir(); 8 output.endianess = os.endianess(); 9 output.hostname = os.hostname(); 10 output.type = os.type(); 11 output.platform = os.platform(); 12 output.arch = os.arch(); 13 output.release = os.release(); 14 output.uptime = os.uptime(); 15 output.loadavg = os.loadavg(); </pre>

4. Switch the browser window to the where the web module is running and change the url path to **/xsjs/os.xsjs**

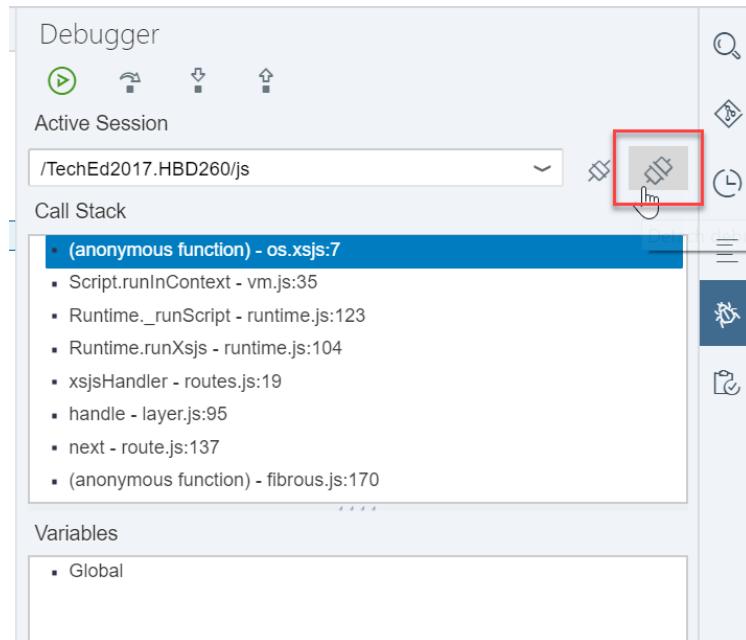


5. Switch back to the Web IDE tab in the browser. The debugger should attach and open in the editor.

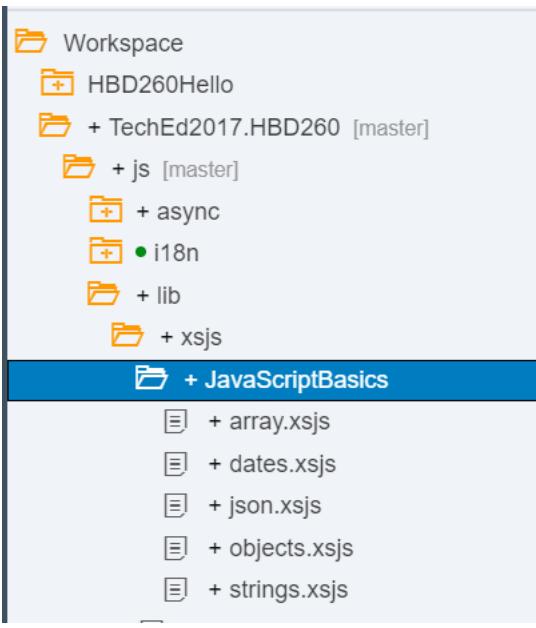
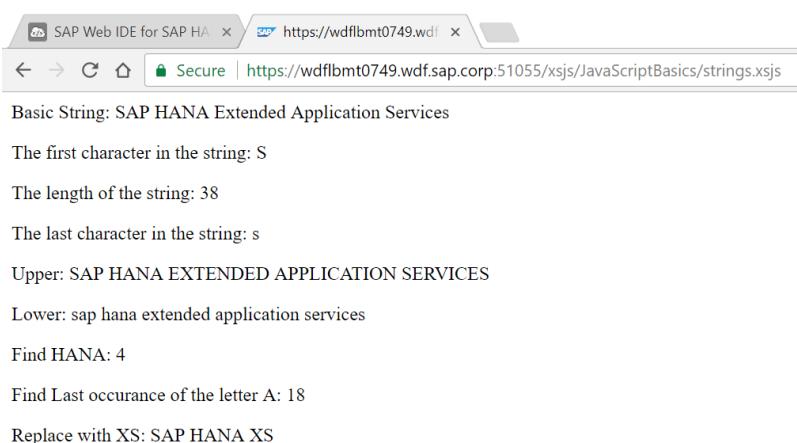


6. You can now spend a moment exploring the debugger. Step over some statements. Try adding the output variable to the Console.

When you are finished, you can detach the debugger by clicking the Detach debugger button.

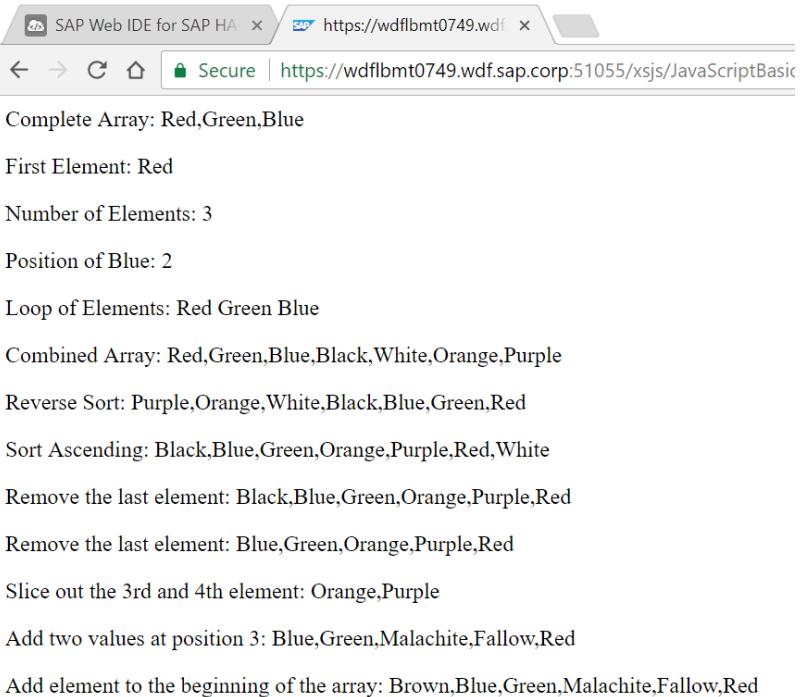


Exercise 2.3: Exploring JavaScript Language Features

Explanation	Screenshot
<p>1) This exercise will be a bit different from all others. Instead of following along step-by-step or working toward a very specific final product; this exercise will instead allow you to explore various JavaScript language features.</p> <p>There is no strict right or wrong solution to many of these exercise parts. However, you can view sample solutions from: https://github.com/l809764/TechEd2017.HBD260/tree/solution/js/lib/xsjs/JavaScriptBasics</p> <p>Feel free to create similar xsjs files to explore these language features. However, the main point of this exercise is to focus on general JavaScript language constructs and not anything XSJS or HANA specific.</p>	 <pre> Workspace HBD260Hello + TechEd2017.HBD260 [master] + js [master] + async + i18n + lib + xsjs + JavaScriptBasics + array.xsjs + dates.xsjs + json.xsjs + objects.xsjs + strings.xsjs </pre>
<p>2) First we will learn about strings in JavaScript.</p> <p>Create a string with some sample text in it (hint: <code>=</code>).</p> <p>Access the first character of the string. (hint: offsets are zero based and specified via <code>[]</code>)</p> <p>Access the last character in the string (hint: <code>slice</code>).</p> <p>Output the length of the string (hint: <code>length</code>).</p> <p>Convert the string to all upper and all lower case (hint: <code>toUpperCase</code>).</p> <p>Find the position of a text part of the string (hint: <code>indexOf</code>).</p> <p>Find the last occurrence of a character in the string.</p> <p>Perform a find and replace operation on the string (hint: <code>lastIndexOf</code>).</p>	 <pre> Basic String: SAP HANA Extended Application Services The first character in the string: S The length of the string: 38 The last character in the string: s Upper: SAP HANA EXTENDED APPLICATION SERVICES Lower: sap hana extended application services Find HANA: 4 Find Last occurrence of the letter A: 18 Replace with XS: SAP HANA XS </pre>



<p>Sample implementation: strings.xsjs</p> <p>3) Create a Data object with the current date (hint: new Date()).</p> <p>Convert the data to various other formats (UTC, Local Data, Local Time, ISO, JSON (hint: to*String with various modifies UTC, Date, LocalDate, LocaleTime, etc)).</p> <p>Access the sub-parts of the date: Year, Month, Day, Hours, Minutes, and Seconds (hint: get* with modifiers for FullYear, Month, Day, Date, Time, Hours, Minutes, Seconds).</p> <p>Perform math on a date, adding 30 days to the current date (hint: setDate and simple math).</p> <p>Sample implementation: dates.xsjs</p>	<pre> Now: Mon Jul 03 2017 16:36:12 GMT+0200 (CEST) Now UTC: Mon, 03 Jul 2017 14:36:12 GMT Now Date String: Mon Jul 03 2017 Now Locale Date String: 7/3/2017 Now Locale Time String: 4:36:12 PM Now Locale String: 7/3/2017, 4:36:12 PM Now ISO String: 2017-07-03T14:36:12.366Z Now JSON String: 2017-07-03T14:36:12.366Z Now Year: 2017 Now Month: 6 Now Day of Week: 1 Now Day of Month: 3 Now number of milliseconds since midnight Jan 1, 1970: 1499092572366 Now Hours: 16 Now Minutes: 36 Now Seconds: 12 30 days from now: Wed Aug 02 2017 16:36:12 GMT+0200 (CEST) </pre>
--	--



SAP Web IDE for SAP HA https://wdflbmt0749.wdf

Secure | https://wdflbmt0749.wdf.sap.corp:51055/xsjs/JavaScriptBasic

```

Complete Array: Red,Green,Blue
First Element: Red
Number of Elements: 3
Position of Blue: 2
Loop of Elements: Red Green Blue
Combined Array: Red,Green,Blue,Black,White,Orange,Purple
Reverse Sort: Purple,Orange,White,Black,Blue,Green,Red
Sort Ascending: Black,Blue,Green,Orange,Purple,Red,White
Remove the last element: Black,Blue,Green,Orange,Purple,Red
Remove the last element: Blue,Green,Orange,Purple,Red
Slice out the 3rd and 4th element: Orange,Purple
Add two values at position 3: Blue,Green,Malachite,Fallow,Red
Add element to the beginning of the array: Brown,Blue,Green,Malachite,Fallow,Red

```

- 4) Create two arrays: one with the colors Red, Green, and Blue and one with Black, White, Orange, and Purple (hint: ["one", "two", "three"]).
- Output the first array as a string (hint: `toString`).
- Access the first element by index (hint: zero based).
- Get the number of elements in the first array (hint: `length`).
- Get the index of Blue in the first array (hint: `indexOf`).
- Loop over the first array and output each element (hint: `for(var i; i < length; i++)`).
- Combine the two arrays (hint: `concat`).
- Reverse Sort and then Sort Ascending (hint: `reverse, sort`).
- Remove the last and then first element (hint: `pop, shift`).
- Slice out the 3rd and 4th element (hint: `slice`).
- Add two new colors (Malachite and Fallow) at the 3rd position (hint: `splice`).
- Add Brown to the start of the Array (hint: `unshift`).
- Sample implementation: `array.xsjs`



- 5) Select 10 records from **PO.Header** and convert the results to JSON (hint: `$.hdb`).

Loop over the elements in the JSON object and create a new **DISCOUNTAMOUNT** attribute by discounting the **GROSSAMOUNT** by 10% (hint: `for (var i; i < length; i++)`).

Convert the JSON object to a string and output it to the frontend (hint: `JSON.stringify`).

Sample implementation: `json.xsjs`

```
[{"PURCHASEORDERID": "300000000", "HISTORY.CREATEDBY.EMPLOYEEID": 20, "HISTORY.CREATEDAT": "2015-01-01", "HISTORY.CHANGEDBY.EMPLOYEEID": 17, "HISTORY.CHANGEDAT": "2015-01-01", "NOTEID": "9000000001", "PARTNER.PARTNERID": 100000000, "CURRENCY": "EUR", "GROSSAMOUNT": "13224.47", "NETAMOUNT": "11113.00", "TAXAMOUNT": "2111.47", "LIFECYCLESTATUS": "N", "APPROVALSTATUS": "I", "CONFIRMSTATUS": "I", "ORDERINGSTATUS": "I", "INVOICINGSTATUS": "I", "DISCOUNTAMOUNT": 11902.023}, {"PURCHASEORDERID": "300000001", "HISTORY.CREATEDBY.EMPLOYEEID": 18, "HISTORY.CREATEDAT": "2015-01-02", "HISTORY.CHANGEDBY.EMPLOYEEID": 1, "HISTORY.CHANGEDAT": "2015-01-02", "NOTEID": "9000000001", "PARTNER.PARTNERID": 100000002, "CURRENCY": "EUR", "GROSSAMOUNT": "12493.73", "NETAMOUNT": "10498.94", "TAXAMOUNT": "1994.79", "LIFECYCLESTATUS": "N", "APPROVALSTATUS": "I", "CONFIRMSTATUS": "I", "ORDERINGSTATUS": "I", "INVOICINGSTATUS": "I", "DISCOUNTAMOUNT": 11244.357}]
```

SAP Web IDE for SAP HA https://wdflbmt0749.wdf.sap.corp:51055/xsjs/JavaScriptBasics/

Object Literals

Red
Blue
Green

Favorite Color

References

Value 1: New Value
Value 2: First Value
Value 3: New Value
Value 4: New Value

Object Constructor

Purchase Order: 300000000 Gross Amount: 13224.47 Discount Amount: 11902.023
Purchase Order: 300000001 Gross Amount: 12493.73 Discount Amount: 11244.357

- 6) Create an object literal called colors. Give it three properties – red with the value #FF0000, green with the value #00FF00, and blue with the value #0000FF. Give the object a function named favoriteColor. On Monday, it should return blue, all other days it returns red (hint: var colors = {}).

Access the colors in the object including the Favorite color (hint: object.<variable>).

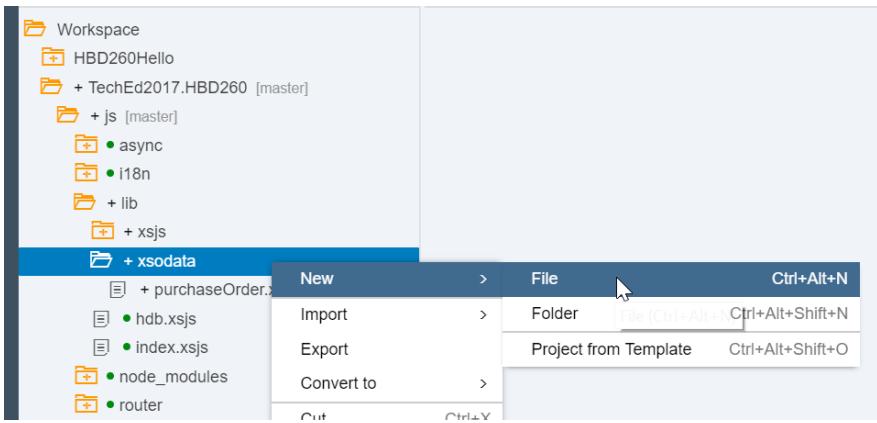
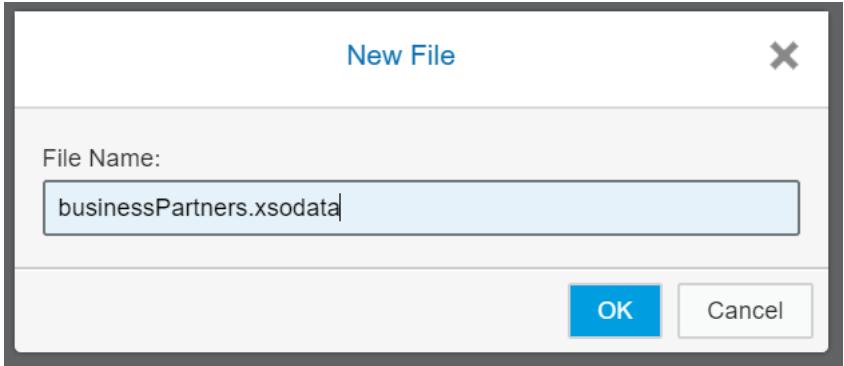
Prove that objects are assign by reference and not assign by value (hint: just set a value to a regular variable and an object and see the difference).

Create a purchase order object for **PO.Header** which returns PO Id, gross amount, and a calculated discount amount (10% off gross amount) (hint: function with logic from last step – json.xsjs).

Create two instances of this object – one for PO 300000000 and one for 300000001

Sample implementation:
objects.xsjs

Exercise 2.4: Creating a Simple OData Service

Explanation	Screenshot
1) Right mouse click on the js/lib/xsodata folder and choose New->File .	
2) Enter the name as businessPartners.xsodata and click "OK".	
3) We want to define an OData service to expose the business partner table. The syntax of the XSODATA service is relative easy for this use case. We need only define the name of the HANA Table we will base the service from (MD.BusinessPartner) and the name of the OData entity (BusinessPartners).	 <pre> 1 service { 2 "MD.BusinessPartner" 3 as "BusinessPartners"; 4 } </pre>

- 4) Save the file. Run the Node.js module. Then return to the running html5 module. Change the url path to **/xsodata/businessPartners.xsodata/?\$format=json** to test this new service.

The resulting document describes the service entities. We only had the one entity named **BusinessPartners**.

```
{
  "d": {
    "EntitySets": [
      "BusinessPartners"
    ]
  }
}
```

- 5) You can now adjust the URL slightly and add the **/\$metadata** parameter to the end of it.

For Example:
/xsodata/businessPartners.xsodata/\$metadata

You can see the field details for all the attributes of the OData service.

```
<edmx:Edmx Version="1.0" xmlns:edmx="http://schemas.microsoft.com/ado/2007/06/edmx">
  <edmx:DataServices m:DataServiceVersion="2.0" xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices" xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices" xmlns:m="http://schemas.microsoft.com/ado/2008/09/edm">
    <Schema Namespace="default" xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices" xmlns:m="http://schemas.microsoft.com/ado/2008/09/edm">
      <EntityType Name="BusinessPartnersType">
        <Key>
          <PropertyRef Name="PARTNERID"/>
        </Key>
        <Property Name="PARTNERID" Type="Edm.Int32" Nullable="false"/>
        <Property Name="PARTNERROLE" Type="Edm.String" MaxLength="3"/>
        <Property Name="EMAILADDRESS" Type="Edm.String" MaxLength="255"/>
        <Property Name="PHONENUMBER" Type="Edm.String" MaxLength="30"/>
        <Property Name="FAXNUMBER" Type="Edm.String" MaxLength="30"/>
        <Property Name="WEBADDRESS" Type="Edm.String" MaxLength="1024"/>
        <Property Name="ADDRESSES.ADDRESSID" Type="Edm.Int32"/>
        <Property Name="COMPANYNAME" Type="Edm.String" MaxLength="80"/>
        <Property Name="LEGALFORM" Type="Edm.String" MaxLength="10"/>
        <Property Name="HISTORY.CREATEDBY.EMPLOYEEID" Type="Edm.Int32"/>
        <Property Name="HISTORY.CREATEDDAT" Type="Edm.DateTime"/>
        <Property Name="HISTORY.CHANGEDBY.EMPLOYEEID" Type="Edm.Int32"/>
        <Property Name="HISTORY.CHANGEDDAT" Type="Edm.DateTime"/>
        <Property Name="CURRENCY" Type="Edm.String" MaxLength="5"/>
      </EntityType>
      <EntityContainer Name="v2" m:IsDefaultEntityContainer="true">
        <EntityType Name="BusinessPartners" EntityType="default.BusinessPartnersType"/>
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

- 6) In order to view the data of the entity, you would append **BusinessPartners** to the end of the URL:

For Example:

/xsodata/businessPartners.xsodata/ BusinessPartners? \$format=json

You are now able to see the data from the businessPartner table.



```
{
  "d": {
    "results": [
      {
        "__metadata": {
          "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/businessPartners.xsodata/ BusinessPartners(100000000)",  

          "type": "default.BusinessPartnersType"
        },  

        "PARTNERID": "100000000",  

        "PARTNERROLE": "1",  

        "EMAILADDRESS": "karl.mueller@sap.com",  

        "PHONENUMBER": "622734567",  

        "FAXNUMBER": null,  

        "WEBADDRESS": "http://www.sap.com",  

        "ADDRESSES.ADDRESSID": "1000000034",  

        "COMPANYNAME": "SAP",  

        "LEGALFORM": "AG",  

        "HISTORY.CREATEDBY.EMPLOYEEID": 33,  

        "HISTORY.CREATEDAT": "/Date(1349222400000)/",  

        "HISTORY.CHANGEDBY.EMPLOYEEID": 33,  

        "HISTORY.CHANGEDAT": "/Date(1349222400000)/",  

        "CURRENCY": "EUR"
      },
      {
        "__metadata": {
          "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/businessPartners.xsodata/ BusinessPartners(100000001)",  

          "type": "default.BusinessPartnersType"
        },  

        "PARTNERID": "100000001",  

        "PARTNERROLE": "2",  

        "EMAILADDRESS": "dagmar.schulze@beckerberlin.de",  

        "PHONENUMBER": "3888538",  

        "FAXNUMBER": null,  

        "WEBADDRESS": "http://www.beckerberlin.de",  

        "ADDRESSES.ADDRESSID": "1000000035",  

        "COMPANYNAME": "Becker Berlin",  

        "LEGALFORM": "GmbH",  

        "HISTORY.CREATEDBY.EMPLOYEEID": 33,  

        "HISTORY.CREATEDAT": "/Date(1349222400000)/",  

        "HISTORY.CHANGEDBY.EMPLOYEEID": 33,  

        "HISTORY.CHANGEDAT": "/Date(1349222400000)/",  

        "CURRENCY": "EUR"
      }
    ]
  }
}
```

- 7) You can also experiment with standard OData URL parameters like \$top, \$skip, or \$filter. These options are interpreted and handled by the OData service of XS for you. You get complex service handling without any coding. For example, the following URL would return only three business partner records and would skip the first five records. Such parameters are helpful when implementing server side scrolling, filtering, or sorting in table UI elements.

For Example:

/xsodata/businessPartners.xsodata/ BusinessPartners? \$top=3&\$skip=5&\$format=json



```
{
  "d": {
    "results": [
      {
        "__metadata": {
          "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/businessPartners.xsodata/ BusinessPartners(100000005)",  

          "type": "default.BusinessPartnersType"
        },  

        "PARTNERID": "100000005",  

        "PARTNERROLE": "1",  

        "EMAILADDRESS": "bart.koenig@tecum-ag.de",  

        "PHONENUMBER": "2511415",  

        "FAXNUMBER": null,  

        "WEBADDRESS": "http://www.tecum-ag.de",  

        "ADDRESSES.ADDRESSID": "1000000039",  

        "COMPANYNAME": "TECUM",  

        "LEGALFORM": "AG",  

        "HISTORY.CREATEDBY.EMPLOYEEID": 33,  

        "HISTORY.CREATEDAT": "/Date(1349222400000)/",  

        "HISTORY.CHANGEDBY.EMPLOYEEID": 33,  

        "HISTORY.CHANGEDAT": "/Date(1349222400000)/",  

        "CURRENCY": "EUR"
      }
    ]
  }
}
```

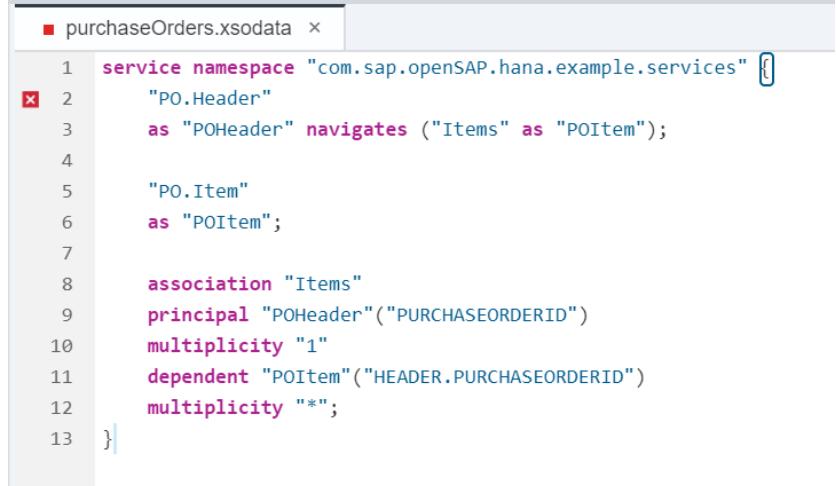
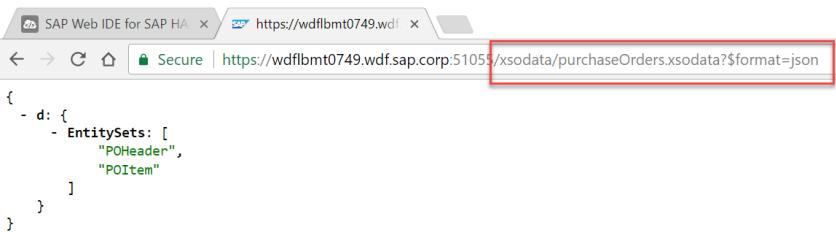


- 8) In our web module, we also already have an SAPUI5 application ready to consume and test this OData service. Change the URL to **/odataBasic/** and you should see this same business partner data in a nice table control.

The screenshot shows a SAP Web IDE interface with a browser window displaying a table titled "Business Partner List (45)". The table has columns: PARTNERID, PAR..., EMAILADDRESS, and PHONENUMBER. The data includes various email addresses and phone numbers for business partners, such as karl.mueller@sap.com, dagmar.schulze@beckerberlin.de, maria.brown@delbont.com, saskia.sommer@talpa-hannover.de, bob.buyer@panorama-studios.biz, bart.koenig@tecum-ag.de, yokonakamura@aia-fit.com, sophie.ribey@laurent-paris.com, victor.sanchez@avantel.com, jorge.velez@telecomunicacionesstar.com, and franklin.jones@pear-computing.com. The browser address bar shows the URL https://wdflbmt0749.wdf.sap.corp:5102/odataBasic/.

	PARTNERID	PAR...	EMAILADDRESS	PHONENUMBER
□	100,000,000	1	karl.mueller@sap.com	622734567
□	100,000,001	2	dagmar.schulze@beckerberlin.de	3088530
□	100,000,002	1	maria.brown@delbont.com	3023352668
□	100,000,003	1	saskia.sommer@talpa-hannover.de	511403266
□	100,000,004	1	bob.buyer@panorama-studios.biz	2244668800
□	100,000,005	1	bart.koenig@tecum-ag.de	2511415
□	100,000,006	1	yoko.nakamura@aia-fit.com	9078563412
□	100,000,007	1	sophie.ribey@laurent-paris.com	149744423
□	100,000,008	1	victor.sanchez@avantel.com	7257716
□	100,000,009	1	jorge.velez@telecomunicacionesstar.com	1133557799
□	100,000,010	2	franklin.jones@pear-computing.com	6789012345

Exercise 2.5: Creating an OData Service with an Entity Relationship

Explanation	Screenshot
<p>1) The previous step of this exercise was very simplistic because it only exposed one database table as a single entity. Often you need to also represent relationships between multiple entities. For example, we might want to build an OData service which has both the Purchase Order Header and Items. For this we would build a one-to-many relationship between the two entities.</p>	
<p>2) Returning to the editor, you should now create a new OData service named purchaseOrders.xsodata and extend it to include the PO.Header and PO.Item tables. Next create a navigation 1:many association. The new content of the definition file should look like this:</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address</p> <p>https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex2/ex2_8</p>	 <pre> 1 service namespace "com.sap.openSAP.hana.example.services" [] 2 "PO.Header" 3 as "POHeader" navigates ("Items" as "POItem"); 4 5 "PO.Item" 6 as "POItem"; 7 8 association "Items" 9 principal "POHeader"("PURCHASEORDERID") 10 multiplicity "1" 11 dependent "POItem"("HEADER.PURCHASEORDERID") 12 multiplicity "*"; 13 } </pre>
<p>3) Save, run and test using the same steps as in the previous section of this exercise.</p> <p>Notice that the base service definition now has two entities</p>	 <pre>{ - d: { - EntitySets: ["POHeader", "POItem"] } }</pre>

- 4) The Header data now has a hyperlink relationship to the item entity

```
{
  "d": {
    "results": [
      {
        "_metadata": {
          "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/purchaseOrders.xsodata/POHeader(300000000)/",
          "type": "com.sap.openSAP.hana.example.services.POHeaderType"
        },
        PURCHASEORDERID: 300000000,
        HISTORY.CREATEDBY_EMPLOYEEID: 20,
        HISTORY.CREATEDAT: "/Date(1420870400000)/",
        HISTORY.CHANGEDBY_EMPLOYEEID: 17,
        HISTORY.CHANGEDAT: "/Date(1420070400000)/",
        NOTEID: "9000000001",
        PARTNER_PARTNERID: 100000000,
        CURRENCY: "EUR",
        GROSSAMOUNT: "13224.47",
        NETAMOUNT: "11113.68",
        TAXAMOUNT: "2111.47",
        LIFECYCLESTATUS: "N",
        APPROVALSTATUS: "I",
        CONFIRMSSTATUS: "I",
        ORDERINGSTATUS: "I",
        INVOICINGSTATUS: "I",
        POItem: {
          "_deferred": [
            {
              "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/purchaseOrders.xsodata/POHeader(300000000)/POItem"
            }
          ]
        }
      },
      {
        "_metadata": {
          "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/purchaseOrders.xsodata/POHeader(300000001)"
        }
      }
    ]
  }
}
```

- 5) Associations can be an excellent way to load child elements on demand; however, there is also an option to expand the children details in place so that all levels can be retrieved with one request.

Test the service again using the same steps as in the previous section of this exercise.

This time add
&\$expand=POItem to the end of the URL.

You will then see that all the items are embedded within each header record.

```
{
  "d": {
    "results": [
      {
        "_metadata": {
          "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/purchaseOrders.xsodata/POHeader(300000000)",
          "type": "com.sap.openSAP.hana.example.services.POHeaderType"
        },
        PURCHASEORDERID: 300000000,
        HISTORY.CREATEDBY_EMPLOYEEID: 20,
        HISTORY.CREATEDAT: "/Date(1420870400000)/",
        HISTORY.CHANGEDBY_EMPLOYEEID: 17,
        HISTORY.CHANGEDAT: "/Date(1420070400000)/",
        NOTEID: "900000001",
        PARTNER_PARTNERID: 100000000,
        CURRENCY: "EUR",
        GROSSAMOUNT: "13224.47",
        NETAMOUNT: "11113.68",
        TAXAMOUNT: "2111.47",
        LIFECYCLESTATUS: "N",
        APPROVALSTATUS: "I",
        CONFIRMSSTATUS: "I",
        ORDERINGSTATUS: "I",
        INVOICINGSTATUS: "I",
        POItem: {
          "results": [
            {
              "_metadata": {
                "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/purchaseOrders.xsodata/POItem(HEDER_PURCHASEORDERID=300000000_PRODUCT_PRODUCTID='HT-1000')",
                "type": "com.sap.openSAP.hana.example.services.POItemType"
              },
              HEADER_PURCHASEORDERID: 300000000,
              PRODUCT_PRODUCTID: "HT-1000",
              NOTEID: null,
              CURRENCY: "EUR",
              GROSSAMOUNT: "1137.64",
              NETAMOUNT: "989.00",
              TAXAMOUNT: "146.64",
              QUANTITY: "1.000",
              QUANTITYUNIT: "EA",
              DELIVERYDATE: "/Date(1430956800000)/"
            }
          ]
        }
      }
    ]
  }
}
```

- 6) In our web module, we also already have an SAPUI5 application ready to consume and test this OData service and show us both entities. Change the URL to **/odataAdv/** and you should see tables with details from both entities in the service.

PURCHASEORDERID	HISTORY.CREATEDBY_EMPLOYEEID
300,000,000	20
300,000,001	18
300,000,002	20
300,000,003	15
300,000,004	11

HEADER_PURCHASEORDERID	PRODUCT.PRODUCTID	NOTEID	CURRE...	GROSSAMOUNT
300,000,008	HT-1000		EUR	1,137.64
300,000,001	HT-1091		EUR	67.88
300,000,001	HT-1100		USD	213.98
300,000,001	HT-2026		USD	39.69
300,000,002	HT-1010		EUR	2,370.91

Exercise 2.6: Creating an OData Service with Create Operation and XSJS Exit

Explanation	Screenshot
<p>1) Odata services aren't limited to read-only operations. They can create, update and delete records as well. We also have exit mechanisms in order to add custom code and validations to these write operations.</p> <p>Create another OData service named poCreate.xsodata for PO.POView in the /js/lib/xsodata folder.</p> <p>This time, also link the create operation to the Server Side JavaScript Library (XSJSLIB) xsjs:poExits.xsjslib and the function po_create. This will be the exit code that performs the insert of the new record. We will also link to xsjs:poExits.xsjslib and the function po_create_before_exit to validate the incoming data before we pass to the actual insertion function.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/I809764/TechEd2017.HBD260/blob/snippets/ex2/ex2_9</p>	<pre>service namespace "com.sap.openSAP.hana.example.services" { "PO.POView" as "purchaseDetails" key ("PURCHASEORDERID") create using "xsjs:poExits.xsjslib::po_create" events(before "xsjs:poExits.xsjslib::po_create_before_exit"); }</pre>
<p>2) In the js/lib/xsjs folder create the file poExits.xsjslib. Here is the code for this file.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/I809764/TechEd2017.HBD260/blob/snippets/ex2/ex2_10</p>	<pre>function po_create_before_exit(param) { \$.trace.error("Start of Exit"); var after = param.afterTableName; var pStmt = null; var poid = ""; try { pStmt = param.connection .prepareStatement("select \"purchaseOrderSeqId\".NEXTVAL from \"DUMMY\""); var rs = pStmt.executeQuery(); while (rs.next()) { poid = rs.getInteger(1); } \$.trace.error(poid); pStmt.close(); }</pre>

```

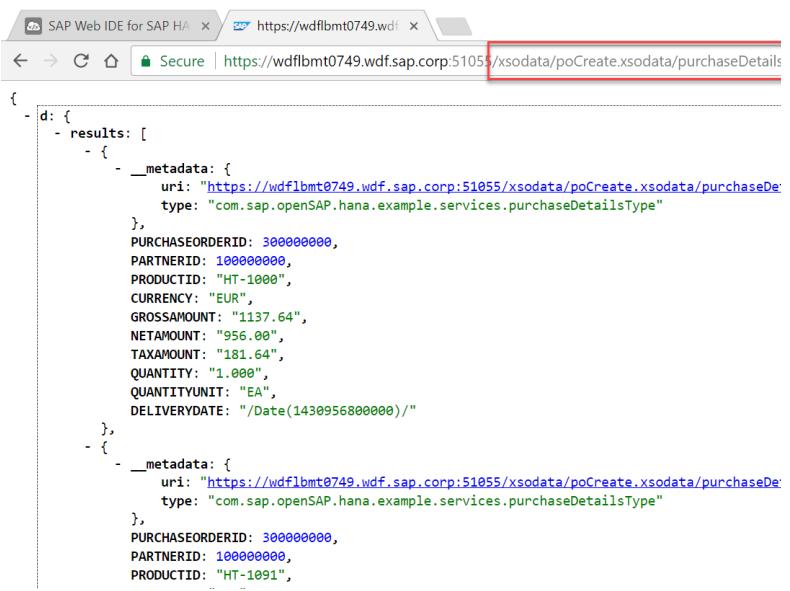
pStmt = param.connection.prepareStatement("update\""+ +
after + "\"set PURCHASEORDERID = ?");
pStmt.setInteger(1, poid);
pStmt.execute();
pStmt.close();
} catch (e) {
    $.trace.error(e.message);
    pStmt.close();
}

}

function po_create(param){
    $.trace.error("Insert");
}

```

- 3) Save and run the js and then the web module. Change the URL to
**/xsodata/poCreate.xsodata/purchas
eDetails/?\$format=json**



```

{
  "d": {
    "results": [
      {
        "_metadata": {
          "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/poCreate.xsodata/purchaseDetails/?$format=json&index=0",
          "type": "com.sap.openSAP.hana.example.services.purchaseDetailsType"
        },
        "PURCHASEORDERID": 30000000,
        "PARTNERID": 10000000,
        "PRODUCTID": "HT-1000",
        "CURRENCY": "EUR",
        "GROSSAMOUNT": "1137.64",
        "NETAMOUNT": "956.00",
        "TAXAMOUNT": "181.64",
        "QUANTITY": "1.00",
        "QUANTITYUNIT": "EA",
        "DELIVERYDATE": "/Date(1430956800000)/"
      },
      {
        "_metadata": {
          "uri": "https://wdflbmt0749.wdf.sap.corp:51055/xsodata/poCreate.xsodata/purchaseDetails/?$format=json&index=1",
          "type": "com.sap.openSAP.hana.example.services.purchaseDetailsType"
        },
        "PURCHASEORDERID": 30000000,
        "PARTNERID": 10000000,
        "PRODUCTID": "HT-1091",
        ...
      }
    ]
  }
}

```



- 4) In our web module, we also already have an SAPUI5 application ready to consume and test this OData service which is especially useful to test Create or Update operations which are difficult to simulate from a browser alone. Change the URL to **/odataCreate/** and you hit the create button to invoke the validation and create exits in the XSODATA service.

The screenshot shows a SAPUI5 application within the SAP Web IDE. The title bar indicates the URL is <https://wdflbm0749.wdf.sap.corp:5190/odataCreate/>. The main content is titled "ENTITY CREATE WITH EXIT". Below it, there's a "Multi-Entity Service Selection" section with "Service Path" and "Header Entity Name" fields. A "Call Service" button is highlighted with a red box. To the right, a table titled "PO Header Data (2,000)" lists purchase order details. At the bottom right of the screen, a message box displays "Create successful" with another red box around it.

EXERCISE 3

Objective

In exercise 2 we worked in a Node.js module, but didn't do much Node.js specific programming. We only were using Node.js to run XSJS and XSODATA services. The support for XSJS and XSODATA is an important feature for XS Advanced. It not only allows backward compatible support for much of your existing development; but it provides a simplified programming model as an alternative to the non-block I/O event driven programming model normally used by Node.js.

But we certainly aren't limited to only the functionality provided by XSJS and XSODATA. We have access to the full programming model of Node.js as well. In this exercise, we will learn how to extend our existing Node.js module in the SAP Web IDE for SAP HANA.

You will learn about how to create and use reusable code in the form of node.js modules. You will use packages.json to define dependencies to these modules which make the installation of them quite easy. You will use one of the most popular modules – **express**; which helps with the setup the handling of the request and response object. You will use **express** to handle multiple HTTP handlers in the same service by using routes.

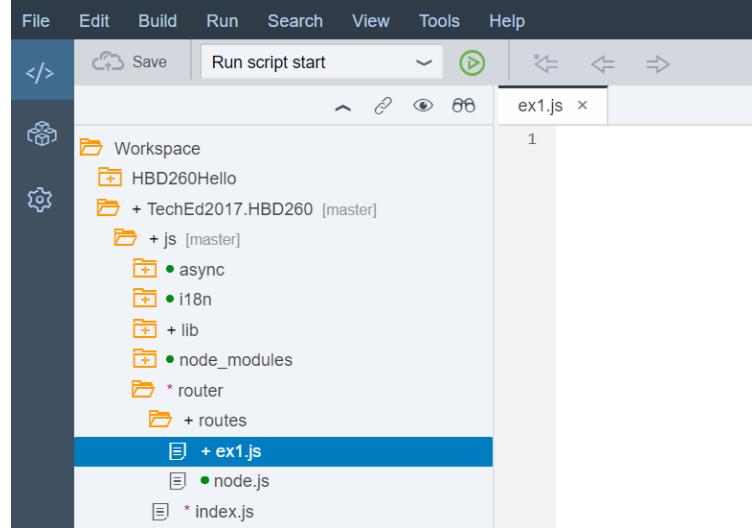
You will learn about the fundamentals of the asynchronous nature of node.js. We will see how this asynchronous capability allows for non-blocking input and output. This technique is one of the basic things that makes Node.js development different from other JavaScript development and creates one of the 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. We will also see how to create language translatable text strings and HANA database queries from Node.js.

Our final part of this 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

- Modules
- Express Module and package.json
- Local Modules
- SAP HANA database access from node.js
- Basic asynchronous processing
- Non-blocking I/O
- Non-blocking HTTP requests
- Non-blocking database requests
- JavaScript basics
- Text bundles
- Web Sockets chat application

Exercise 3.1: Modules and Express

Explanation	Screenshot
<p>1. We want to create a good Node.js project structure that can accommodate many different Express route requests. To follow some Express best practices, we created a folder in js called router with an index.js file in it within the template project you cloned in Exercise 2. This is the root route handler.</p> <p>Add a second route for the path /node/ex1 to this file. The handler for this route will be called ex1.</p>	 <pre>*index.js x 1 "use strict"; 2 3 module.exports = function(app, server) { 4 app.use("/node", require("./routes/node")()); 5 app.use("/node/ex1", require("./routes/ex1")()); 6 };</pre>
<p>2. Notice that in the previous additions we added a reference to a module called ./routes/ex1. This is a Node.js module which will be local to our project.</p> <p>Modules don't all have to come from central repositories. They can also be a way of modularizing your own code (like XJSLIB files). That is exactly what we will do here. Rather than filling up our server.js with application specific logic, we will break that out into its own Node.js module named ex1.js.</p>	
<p>3. Inside the folder js/router/routes create a file called ex1.js.</p>	 <p>The screenshot shows a code editor interface with a toolbar at the top. The left sidebar displays a file tree for a workspace named 'HBD260Hello' under 'TechEd2017.HBD260 [master]'. The tree includes folders for 'js' (containing 'async', 'i18n', 'lib', 'node_modules', 'router', and 'routes'), and files for 'index.js' and 'node.js'. A new file 'ex1.js' is shown in the routes folder, indicated by a blue selection bar underneath the file name.</p>

4. Add the following code to your ex1.js file. This will add a simple GET handler that returns a “Hello World” message when requested.

Notice express in this module has a root route handler. But this will relative to where it is attached in router/index.js. Therefore, this logic will still respond relative to the /node route handler base we specified earlier.

Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/1809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_1

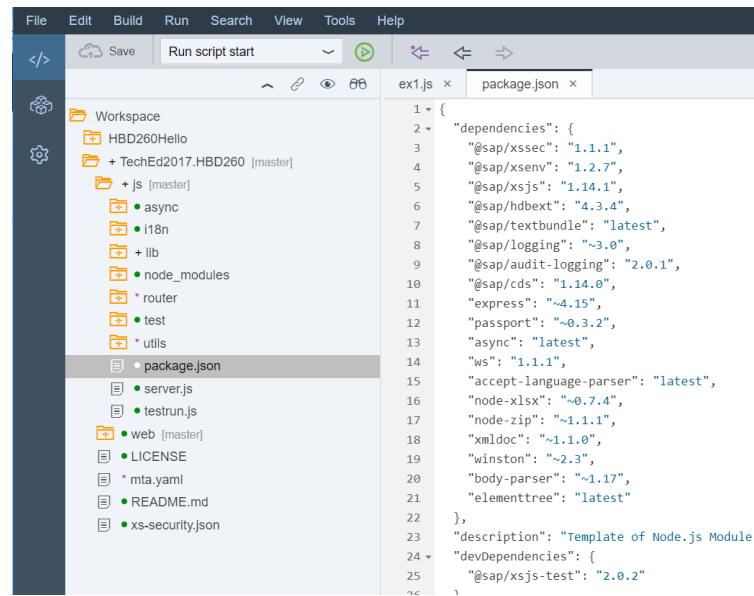
5. Look at the **js/package.json** file in the editor. You will see the dependencies section which lists all required libraries and their versions.

We manually added the express Node.js module to our project, so we also need to extend the dependencies section here. The skeleton project we cloned/imported in Exercise 1 already had all necessary dependencies, but we want everyone to be aware of how the system knows to pull in these external and open source modules.

```
/*eslint no-console: 0, no-unused-vars: 0, no-shadow: 0, new-cap: 0*/
"use strict";
var express = require("express");

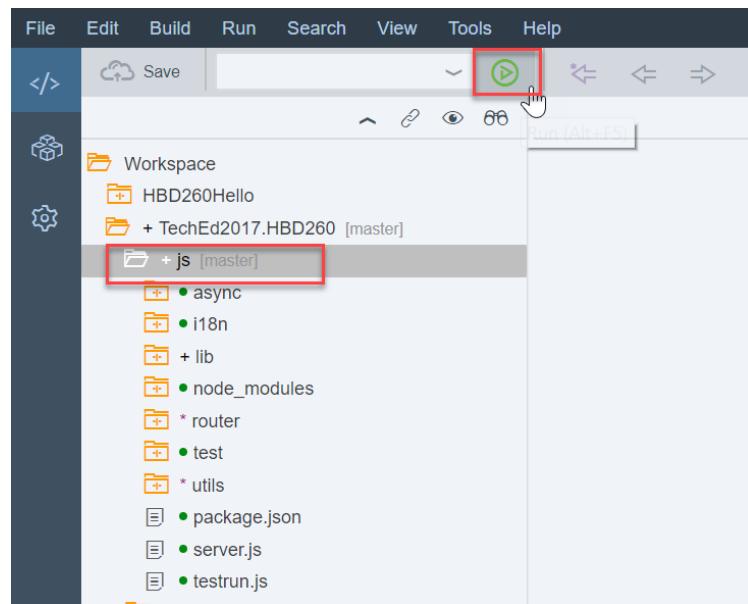
module.exports = function() {
  var app = express.Router();

  //Hello Router
  app.get("/", function(req, res) {
    res.send("Hello World Node.js");
  });
  return app;
};
```



```
1  {
2    "dependencies": {
3      "@sap/xssec": "1.1.1",
4      "@sap/xsenv": "1.2.7",
5      "@sap/xsjs": "1.14.1",
6      "@sap/hdbext": "4.3.4",
7      "@sap/textbundle": "latest",
8      "@sap/logging": "~3.0",
9      "@sap/audit-logging": "2.0.1",
10     "@sap/cds": "1.14.0",
11     "express": "~4.15",
12     "passport": "~0.3.2",
13     "async": "latest",
14     "ws": "1.1.1",
15     "accept-language-parser": "latest",
16     "node-xlsx": "~0.7.4",
17     "node-zip": "1.1.1",
18     "xmldoc": "1.1.0",
19     "winston": "2.3",
20     "body-parser": "~1.17",
21     "elementtree": "latest"
22   },
23   "description": "Template of Node.js Module"
24   "devDependencies": {
25     "@sap/xsjs-test": "2.0.2"
26   }
}
```

6. We can now run the **js** module.



7. You should see that the application was updated successfully is running again.

```

Run TechEd2017.HBD260
js  Run script start
Application is starting
7/3/17 4:24:12,009 PM [APP/2-0] OUT
7/3/17 4:24:12,009 PM [APP/2-0] OUT > js@1.6.0 start /usr/sap/hana/T01xs/app_working/wdflbmt0749/executionroot/904f9312-215b-40be-af1f-
41457413389/Aero
7/3/17 4:24:12,009 PM [APP/2-0] OUT
7/3/17 4:24:12,009 PM [APP/2-0] OUT > node server.js
7/3/17 4:24:12,009 PM [APP/2-0] OUT
7/3/17 4:24:14,066 PM [APP/2-0] OUT HTTP Server: 59235
Application is running
  
```

8. If you closed the tab in the browser for the web module, you can reopen from the link in the run console.

9. Now change the path in the browser to **/node/ex1**. You should see Hello World Node.js this is being returned from the new route handler and Node.js module we just implemented in this exercise.

Exercise 3.2: HANA Database Access from Node.js

Explanation	Screenshot
<p>1. In this exercise, we will look at how to use the HANA database access library to send queries to the database.</p> <p>Return to the js module and using what you learned in the previous 5.1 exercise create a new route called ex2. Let's extend the logic here with a URL handler for /node/ex2 which will issue a database SELECT statement against the DUMMY table and return the current database user.</p>	
<p>2. Now add a new route handler for /ex2 in the file ex2.js that gets the database connection/client from the express request object (req.db). Then create a prepared statement for the SELECT of SESSION_USER from dummy. Execute the statement and send the results as JSON in the response object.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_2</p>	<pre> /*eslint no-console: 0, no-unused-vars: 0, no-shadow: 0, new-cap: 0*/ "use strict"; var express = require("express"); module.exports = function() { var app = express.Router(); //Simple Database Select - In-line Callbacks app.get("/", function(req, res) { var client = req.db; client.prepare("select SESSION_USER from \"DUMMY\" ", function(err, statement) { if (err) { res.type("text/plain").status(500).send("ERROR: " + err.toString()); return; } statement.exec([], function(err, results) { if (err) { res.type("text/plain").status(500).send("ERROR: " + err.toString()); return; } else { var result = JSON.stringify({ Objects: results }); res.end(result); } }); }); }); } </pre>

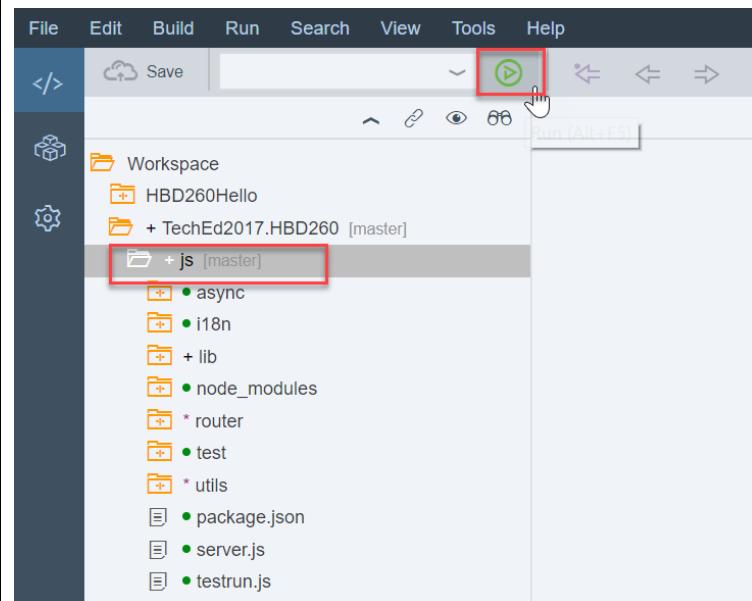
```

        res.type("application/json").status(200).send(result);
    }
});

return app;
}

```

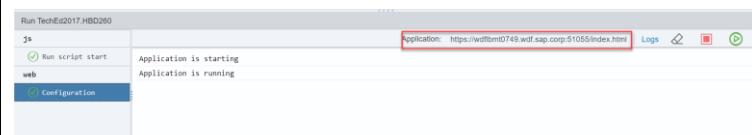
3. We can now run the **js** module.



4. You should see that the application was updated successfully is running again.



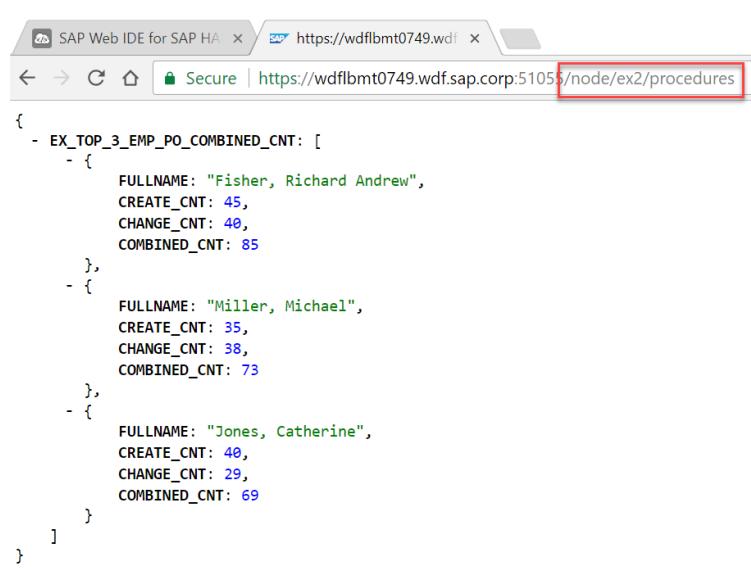
5. Click on the web folder and the run window should change to the details of the running HTML5 module. Since its already running and we didn't make any changes to it, you can just click on the Application link in the top right corner of the window to reopen it in your web browser if you closed it from the previous step.



<p>6. Now change the path in the browser to /node/ex2. You should see the output of your successful SELECT statement.</p>	 <pre>{ "Objects": [{ "SESSION_USER": "SBSS_44297795544537365903087900279892588506072909782799399804937004443" }] }</pre>
<p>7. You might have noticed that the default Node.js programming approach is to use callbacks/event handlers for each operation. This is because even the different parts of a database request (connection, prepared statement, execution, etc) are all non-blocking operations.</p> <p>While this provides considerable parallelization and performance opportunities; it can also make the code more difficult to read. As the next part of this exercise you can return to ex2.js and add a second route handler, /waterfall, that performs the same select but uses the async module.</p> <p>This module doesn't change the runtime aspects of the code, but organizes the callback functions in an easier to read array instead of inlining them within each other.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_3</p> <p>Then run the js module and test again at /node/ex2/waterfall</p>	<pre>var async = require("async"); //Simple Database Select - Async Waterfall app.get("/waterfall", function(req, res) { var client = req.db; async.waterfall([function prepare(callback) { client.prepare("select SESSION_USER from \"DUMMY\" ", function(err, statement) { err, statement); callback(null, statement); }); }, function execute(err, statement, callback) { statement.exec([], function(execErr, results) { results); callback(null, execErr, results); }); }, function response(err, results, callback) { if (err) { res.type("text/plain").status(500).send("ERROR: " + err.toString()); return; } else { var result = JSON.stringify({ results }); res.type("application/json").status(200).send(result); } }]); });</pre>

	});
<p>8. Next we want to see how to call Stored Procedures from Node.js. The logic needed is different from selects as you call the loadProcedure function and then pass a specific interface structure to call the procedure.</p> <p>As the next part of this exercise you can return to ex2.js and add a third route handler, /procedures, that performs calls the stored procedure, get_po_header_data.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_4</p>	<pre>//Simple Database Call Stored Procedure app.get("/procedures", function(req, res) { var client = req.db; var hdbext = require("@sap/hdbext"); //((client, Schema, Procedure, callback) hdbext.loadProcedure(client, null, "get_po_header_data", function(err, sp) { if (err) { res.type("text/plain").status(500).send("ERROR: " + err.toString()); return; } //((Input Parameters, callback(errors, Output Scalar Parameters, [Output Table Parameters]) sp({}, function(err, parameters, results) { if (err) { res.type("text/plain").status(500).send("ERROR: " + err.toString()); } var result = JSON.stringify({ EX_TOP_3_EMP_PO_COMBINED_CNT: results }); res.type("application/json").status(200).send(result); }); }); });</pre>

9. Then run the **js** module and test again at **/node/ex2/procedures**



```
{
  - EX_TOP_3_EMP_PO_COMBINED_CNT: [
    - {
      FULLNAME: "Fisher, Richard Andrew",
      CREATE_CNT: 45,
      CHANGE_CNT: 40,
      COMBINED_CNT: 85
    },
    - {
      FULLNAME: "Miller, Michael",
      CREATE_CNT: 35,
      CHANGE_CNT: 38,
      COMBINED_CNT: 73
    },
    - {
      FULLNAME: "Jones, Catherine",
      CREATE_CNT: 40,
      CHANGE_CNT: 29,
      COMBINED_CNT: 69
    }
  ]
}
```

10. Let's add another path that calls a stored procedure but this time we have input parameters on the procedure.

As the next part of this exercise you can return to **ex2.js** and add a fourth route handler, **/procedures2**, that calls the stored procedure, **get_bp_addresses_by_role**. Notice how we add `:partnerRole?` To the express route definition to process this as a URL parameter.

Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_5

```
//Database Call Stored Procedure With Inputs
app.get("/procedures2/:partnerRole?", function(req,
res) {
  var client = req.db;
  var hdbext = require("@sap/hdbext");
  var partnerRole = req.params.partnerRole;
  var inputParams = "";
  if (typeof partnerRole === "undefined" ||
  partnerRole === null) {
    inputParams = {};
  } else {
    inputParams = {
      IM_PARTNERROLE:
        partnerRole
    };
  }
  //((client, Schema, Procedure, callback)
  hdbext.loadProcedure(client, null,
  "get_bp_addresses_by_role", function(err, sp) {
    if (err) {
      res.type("text/plain").status(500).send("ERROR: " +
      err.toString());
      return;
    }
    //((Input Parameters, callback(errors,
    Output Scalar Parameters, [Output Table Parameters])
    sp(inputParams, function(err,
    parameters, results) {
      if (err) {
```

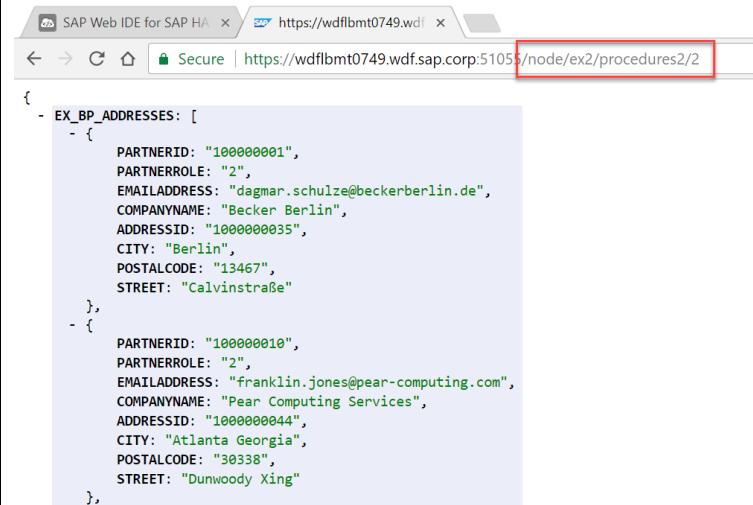
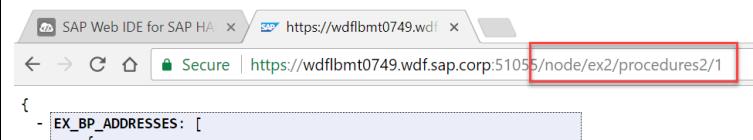
```

        res.type("text/plain").status(500).send("ERROR: " +
err.toString());
    }
    var result = JSON.stringify({
        EX_BP_ADDRESSES: results
    });

    res.type("application/json").status(200).send(result);
});
});

```

11. Then run the **js** module and test again at **/node/ex2/procedures2/1** and **/node/ex2/procedures2/2** to test both partner types – 1=customers and 2=suppliers



12. But we can also take advantage of the parallelization capabilities of Node.js and call two stored procedures in parallel with two separate requests to the database.

As the next part of this exercise you can return to **ex2.js** and add a fifth route handler, **/proceduresParallel**, that calls both two previous stored procedures in parallel requests.

Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_6

```
//Call 2 Database Stored Procedures in Parallel
app.get("/proceduresParallel", function(req, res) {
  var client = req.db;
  var hdbext = require("@sap/hdbext");
  var inputParams = {
    IM_PARTNERROLE: "1"
  };
  var result = {};
  async.parallel([
    function(cb) {
      hdbext.loadProcedure(client,
null, "get_po_header_data", function(err, sp) {
        if (err) {
          cb(err);
          return;
        }
        //Input Parameters,
        callback(errors, Output Scalar Parameters, [Output Table
        Parameters])
        sp(inputParams,
        function(err, parameters, results) {
          result.EX_TOP_3_EMP_PO_COMBINED_CNT =
          results;
          cb();
        });
      });
    },
    function(cb) {
      //client, Schema, Procedure,
      callback)
      hdbext.loadProcedure(client,
null, "get_bp_addresses_by_role", function(err, sp) {
        if (err) {
          cb(err);
          return;
        }
        //Input Parameters,
        callback(errors, Output Scalar Parameters, [Output Table
        Parameters])
        sp(inputParams,
        function(err, parameters, results) {
          result.EX_BP_ADDRESSES = results;
          cb();
        });
      });
    }
  ], function(err) {
    if (err) {
```

```

        res.type("text/plain").status(500).send("ERROR: " +
err);
    } else {
        res.type("application/json").status(200).send(JSON.stri-
ngify(result));
    }
});

});

```

13. Then run the **js** module and test again at **/node/ex2/proceduresParallel**

```

{
  - EX_TOP_3_EMP_PO_COMBINED_CNT: [
    - {
      FULLNAME: "Fisher, Richard Andrew",
      CREATE_CNT: 45,
      CHANGE_CNT: 40,
      COMBINED_CNT: 85
    },
    - {
      FULLNAME: "Miller, Michael",
      CREATE_CNT: 35,
      CHANGE_CNT: 38,
      COMBINED_CNT: 73
    },
    - {
      FULLNAME: "Jones, Catherine",
      CREATE_CNT: 40,
      CHANGE_CNT: 29,
      COMBINED_CNT: 69
    }
  ],
  - EX_BP_ADDRESSES: [
    - {
      PARTNERID: "10000000",
      PARTNERROLE: "1",
      EMAILADDRESS: "karl.mueller@sap.com",
      COMPANYNAME: "SAP",
      ADDRESSID: "1000000034",
      CITY: "Walldorf",
      POSTALCODE: "69190",
      STREET: "Dietmar-Hopp-Allee"
    },
    - {
      PARTNERID: "10000002",
    }
  ]
}

```

14. Now let's recreate a few of the XSJS samples from Exercise 2.1. We will begin with the **whoAmI** exercise. In 2.1 we used the **\$session** to get the current user. However, in pure Node.js we don't have the \$ APIs. Instead we can read the same information, and more, from the request **authInfo** object.

As the next part of this exercise you can return to **ex2.js** and add a sixth route handler, **/whoAmI**, that outputs the **req.authInfo**.

```

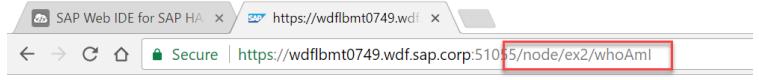
app.get("/whoAmI", function(req, res) {
  var userContext = req.authInfo;
  var result = JSON.stringify({
    userContext: userContext
  });

  res.type("application/json").status(200).send(result);
});

```

Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_7

15. Then run the **js** module and test again at **/node/ex2/whoAmI**



```
{
  - userContext: {
    token: "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpxVCJ9.eyJqdGkiOiJKNGIxOTNmYS1lNmVjLTQzMmUtOGRhYi05YW95xfohkhVmW2X0jg_rIALzTvtRn1DEzlihPJsaF1s5KF2C5HI2d-P8zDjr0vGRY3Nqsc4u9gWF53mW109_",
    - config: {
      tenantmode: "dedicated",
      clientid: "sb-hbd260!i2",
      verificationkey: "-----BEGIN PUBLIC KEY-----MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIB",
      xsappname: "hbd260!i2",
      identityzone: "uaa",
      clientsecret: "ohw/Ywb+2Md5Pdyft7r15Zm+dpycU0I02MKZ4A3xs0xIVP4BHYH6ErwER6gUauduO",
      url: "https://wdflbmt0749.wdf.sap.corp:30032/uaa-security"
    },
    xsappname: "hbd260!i2",
    isUserInfoInitialized: false,
    isForeignMode: false,
    tokenContainsAttributes: true,
    tokenContainsAdditionalAuthAttributes: false,
    - userInfo: {
      logonName: "HBD260",
      firstName: "unknown",
      lastName: "HBD260",
      email: "HBD260@unknown"
    },
    - scopes: [
      "hbd260!i2.Create",
      "hbd260!i2.Display",
      "hbd260!i2.Delete",
      "hbd260!i2.Edit",
      "openid",
      "hbd260!i2.ODATASERVICEADMIN",
      "hbd260!i2.ODATASERVICEUSER"
    ],
    identityZone: "uaa",
    - userAttributes: {
      - client: [
        "300"
      ]
    }
  },
  - scopes: [
    "hbd260!i2.Create",
    "hbd260!i2.Display",
    "hbd260!i2.Delete",
    "hbd260!i2.Edit",
    "openid",
    "hbd260!i2.ODATASERVICEADMIN",
    "hbd260!i2.ODATASERVICEUSER"
  ],
  identityZone: "uaa",
  - userAttributes: {
    - client: [
      "300"
    ]
  }
}
```

16. We will continue converting xsjs exercise from earlier. For this step, convert the hdb.xsjs example that read from the Purchase Order table and output the results as JSON.

As the next part of this exercise you can return to **ex2.js** and add a seventh route handler, **/hdb**, that outputs the results of a selection from **PurchaseOrder.Item**.

Note: if you don't want to type this code, we recommend that you cut

```
app.get("/hdb", function(req, res) {
  var client = req.db;
  client.prepare(
    "SELECT FROM PurchaseOrder.Item
{ " +
    " POHeader.PURCHASEORDERID as
\"PurchaseorderId\", " +
    " PRODUCT as \"ProductID\", " +
    " GROSSAMOUNT as \"Amount\" " +
    " }",
    function(err, statement) {
      if (err) {
```

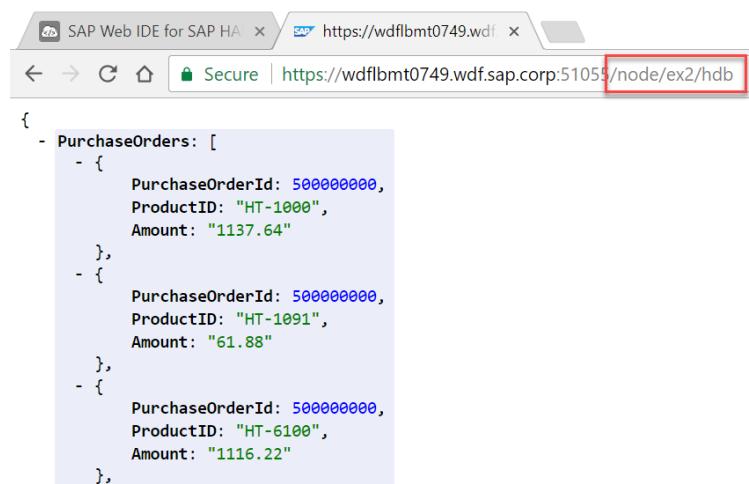
and paste it from this web address
https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_8

```

        res.type("text/plain").status(500).send("ERROR: " +
err.toString());
                                return;
    }
    statement.exec([], function(err, results) {
        if (err) {

            res.type("text/plain").status(500).send("ERROR: " +
err.toString());
            return;
        } else {
            var
result = JSON.stringify({
    PurchaseOrders: results
});
            res.type("application/json").status(200).send(result);
        }
    });
});
});
```

17. Then run the **js** module and test again at **/node/ex2/hdb**



18. For the final conversion, let's bring over **os.xsjs**. From pure Node.js we can simply require the **os** module.

As the next part of this exercise you can return to **ex2.js** and add an eight route handler, **/os**, that outputs the results from a call to the **os** module.

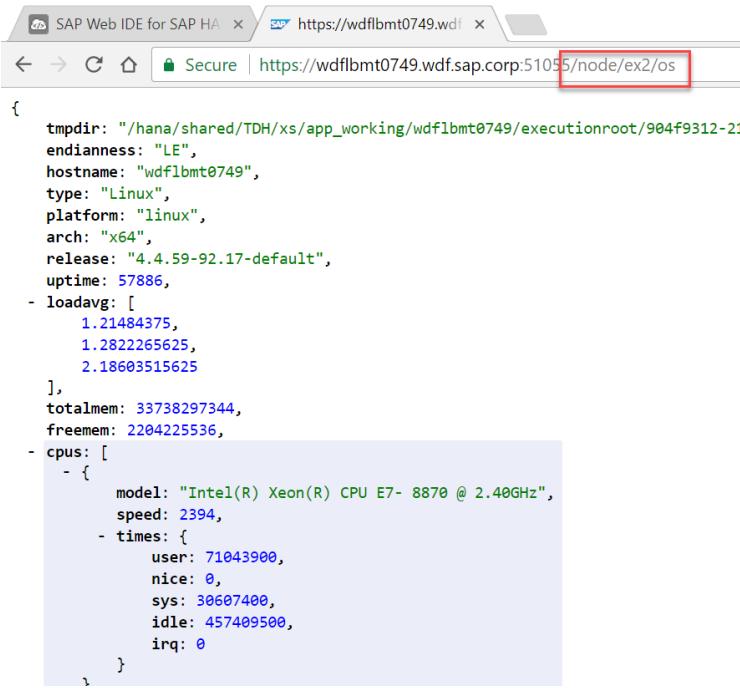
Note: if you don't want to type this

```

app.get("/os", function(req, res) {
    var os = require("os");
    var output = {};

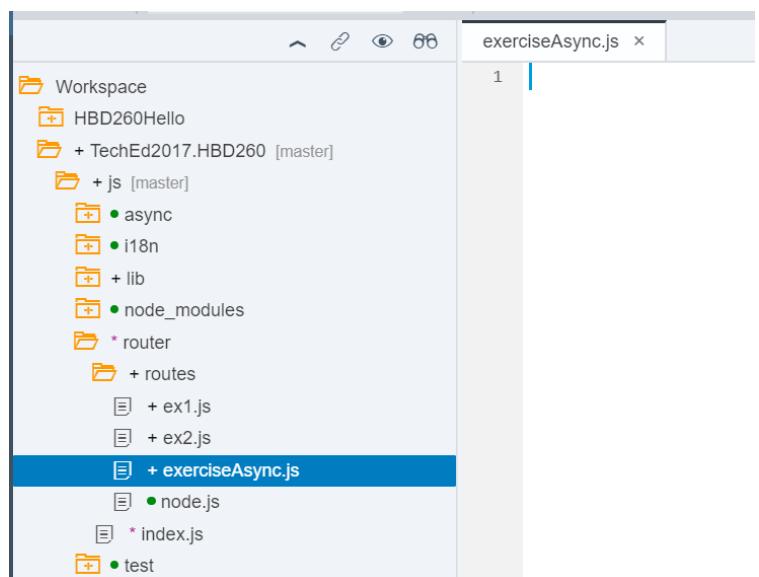
    output.tmpdir = os.tmpdir();
    output.endianness = os.endianness();
    output.hostname = os.hostname();
    output.type = os.type();
    output.platform = os.platform();
    output.arch = os.arch();
    output.release = os.release();
```



<p>code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_9</p>	<pre>output.uptime = os.uptime(); output.loadavg = os.loadavg(); output.totalmem = os.totalmem(); output.freemem = os.freemem(); output.cpus = os.cpus(); output.networkInfraces = os.networkInterfaces(); var result = JSON.stringify(output); res.type("application/json").status(200).send(result); });</pre>
<p>19. Then run the js module and test again at /node/ex2/os</p>	 <pre>{ tmpdir: "/hana/shared/TDH/xs/app_working/wdflbmt0749/executionroot/904f9312-21", endianness: "LE", hostname: "wdflbmt0749", type: "Linux", platform: "linux", arch: "x64", release: "4.4.59-92.17-default", uptime: 57886, - loadavg: [1.21484375, 1.2822265625, 2.18603515625], totalmem: 33738297344, freemem: 2204225536, - cpus: [- { model: "Intel(R) Xeon(R) CPU E7- 8870 @ 2.40GHz", speed: 2394, - times: { user: 71043900, nice: 0, sys: 30607400, idle: 457409500, irq: 0 } }] }</pre>

Exercise 3.3: Asynchronous Non-Blocking I/O

In this exercise, you will learn about the fundamentals of the asynchronous nature of Node.js. We will also 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 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.

Explanation	Screenshot
<p>1. Return to the js module and the router/index.js source file. Add a second module require statement for ./exerciseAsync</p>	<pre data-bbox="723 614 1373 889"> "use strict"; module.exports = function(app, server) { app.use("/node", require("./routes/node")()); app.use("/node/ex1", require("./routes/ex1")()); app.use("/node/ex2", require("./routes/ex2")()); app.use("/node/excAsync", require("./routes/exerciseAsync")(server)); };</pre>
<p>2. Create a new file in your router/routes folder called exerciseAsync.js.</p>	 <p>The screenshot shows a file explorer window with the following directory structure:</p> <ul style="list-style-type: none"> Workspace <ul style="list-style-type: none"> HBD260Hello + TechEd2017.HBD260 [master] <ul style="list-style-type: none"> + js [master] <ul style="list-style-type: none"> async i18n + lib node_modules * router <ul style="list-style-type: none"> + routes <ul style="list-style-type: none"> + ex1.js + ex2.js + exerciseAsync.js node.js index.js
<p>3. Add the following code to your exerciseAsync.js file. You can look at this code if you want, but isn't what we want you to focus on for this exercise. Instead this is just a test framework.</p> <p>We will instead write the important parts of this exercise in a series of additional Node.js modules – each focusing on a different Node.js asynchronous aspect.</p>	<pre data-bbox="723 1529 1373 1861"> /*eslint no-console: 0, no-unused-vars: 0, new-cap:0 */ "use strict"; var express = require("express"); var WebSocketServer = require("ws").Server; module.exports = function(server) { var app = express.Router(); var asyncLib = require(global.__base + "async/async.js"); var dbAsync = require(global.__base + "async/databaseAsync.js");</pre>

Note: if you don't want to type this code, we recommend that you cut and paste it from this web address
https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_10

```

var dbAsync2 = require(global.__base +
"async/databaseAsync2.js");
var fileSync = require(global.__base +
"async/fileSync.js");
var fileAsync = require(global.__base +
"async/fileAsync.js");
var httpClient = require(global.__base +
"async/httpClient.js");

app.use(function(req, res) {
    var output = "<H1>Asynchronous
Examples</H1><br>" +
    "<a
href="/exerciseAsync"/>exerciseAsync</a> - Test Framework
for Async Examples<br>";
    res.type("text/html").status(200).send(output);
});

var wss = new WebSocketServer({
    server: server,
    path: "/node/excAsync"
});

wss.broadcast = function(data) {
    var message = JSON.stringify({
        text: data
    });
    wss.clients.forEach(function each(client) {
        try {
            client.send(message);
        } catch (e) {
            console.log("Broadcast Error:
%s", e.toString());
        }
    });
    console.log("sent: %s", message);
};

wss.on("connection", function(ws) {
    console.log("Connected");

    ws.on("message", function(message) {
        console.log("received: %s", message);
        var data = JSON.parse(message);
        switch (data.action) {
            case "async":
                asyncLib.asyncDemo(wss);
                break;
            case "fileSync":
                fileSync.fileDemo(wss);
                break;
            case "fileAsync":

```



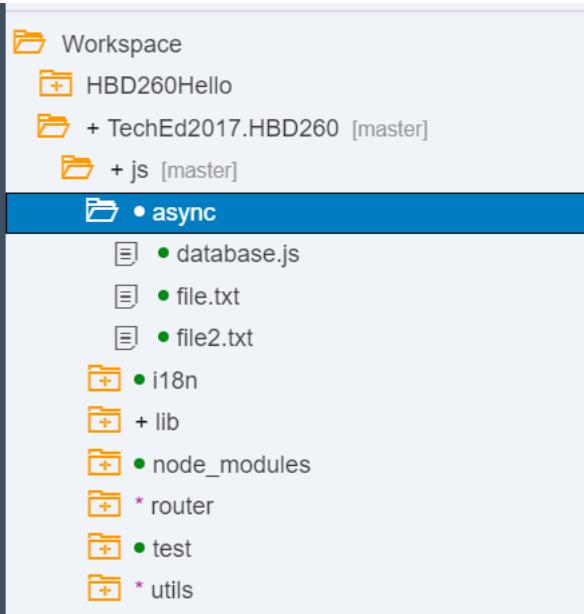
```

        fileAsync.fileDemo(wss);
                break;
        case "httpClient":
            httpClient.callService(wss);
                break;
        case "dbAsync":
            dbAsync.dbCall(wss);
                break;
        case "dbAsync2":
            dbAsync2.dbCall(wss);
                break;
        default:
            wss.broadcast("Error:
Undefined Action: " + data.action);
                break;
    }
}); ws.send(JSON.stringify({
    text: "Connected to Exercise 3"
}));
});

return app;
}

```

4. Our skeleton project we imported in Exercise 1 had a folder under **js** called **async**. This is where we will create the rest of the files in this exercise part.



5. In the **js/async** folder, create a file named **async.js**. In this code, we will output a start message, then set a timer which will issue a message after 3

```

"use strict";
module.exports = {
    asyncDemo: function(wss){
        wss.broadcast("Start");
    }
};

```

<p>seconds. Finally, we will issue an ending message.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_11</p> <p>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.</p>	<pre>setTimeout(function(){ wss.broadcast("Wait Timer Over"); }, 3000); wss.broadcast("End"); });</pre>
<p>6. Let's first look at the difference between synchronous and asynchronous file operations.</p> <p>In the js/async folder, create a file named fileSync.js. The fileSync.js is using the fs library and the function readFileSync to read each of the two text files. After each read operation output there is a message.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_12</p>	<pre>"use strict"; var fs = require("fs"); module.exports = { fileDemo: function(wss) { var text = fs.readFileSync("./async/file.txt", "utf8"); wss.broadcast(text); wss.broadcast("After First Read\n"); text = fs.readFileSync("./async/file2.txt", "utf8"); wss.broadcast(text); wss.broadcast("After Second Read\n"); } };</pre>
<p>7. In the js/async folder, create a file named fileAsync.js. Notice that the message output 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.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it</p>	<pre>"use strict"; var fs = require("fs"); module.exports = { fileDemo: function(wss) { fs.readFile("./async/file.txt", "utf8", function(error, text) { wss.broadcast(text); }); wss.broadcast("After First Read\n"); } };</pre>



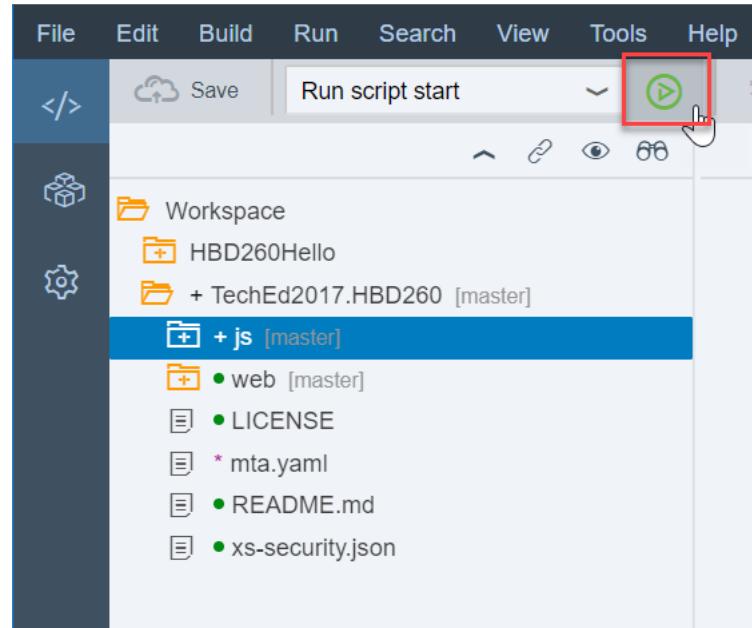
<p>from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_13</p>	<pre>fs.readFile("./async/file2.txt", "utf8", function(error, text) { wss.broadcast(text); }); wss.broadcast("After Second Read\n"); } };</pre>
<p>8. In the js/async folder, create a file named httpClient.js.</p> <p>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.</p> <p>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 message before and after the HTTP request.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_14</p>	<pre>"use strict"; var http = require("http"); module.exports = { callService: function(wss) { wss.broadcast("Before HTTP Call\n"); try { http.get({ path: "http://www.loc.gov/pictures/search/?fo=json&q=SAP&", host: "proxy.wdf.sap.corp", port: "8080", headers: { host: "www.loc.gov" } }, function(response) { response.setEncoding("utf8"); response.on("data", function(data) { wss.broadcast(data.substring(0, 100)); }); response.on("error", wss.broadcast); }) } catch (err) { wss.broadcast(err.toString()); } wss.broadcast("After HTTP Call\n"); } }; };</pre>

<p>9. In the js/async folder, create a file named databaseAsync.js.</p> <p>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.</p> <p>Earlier in this exercise, you learned 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.</p> <p>First we have databaseAsync.js. This issues a message, then calls two functions (callHANA1 and callHANA2), then issues another message. This will execute two different queries in the HANA database.</p> <p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_15</p>	<pre>"use strict"; var hana = require("./database"); module.exports = { dbCall: function(wss){ function dummy(){} wss.broadcast("Before Database Call"); hana.callHANA1(dummy, wss); hana.callHANA2(dummy, wss); wss.broadcast("After Database Call"); } };</pre>
<p>10. In the js/async folder, create a file named databaseAsync2.js.</p> <p>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.</p> <p>In databaseAsync2.js we 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.</p>	<pre>/*eslint no-console: 0, no-unused-vars: 0, no-undef: 0, no-shadow: 0*/ "use strict"; var hana = require("./database"); var async = require("async"); module.exports = { dbCall: function(wss){ async.parallel([function(cb){wss.broadcast("Before Database Call"); cb();}, function(cb){hana.callHANA1(cb, wss); }, function(cb){hana.callHANA2(cb, wss); }, function(cb){wss.broadcast("After Database Call"); cb();}], function(err){ wss.broadcast("---Everything's Really Done Now. Go Home!--"); }); } };</pre>

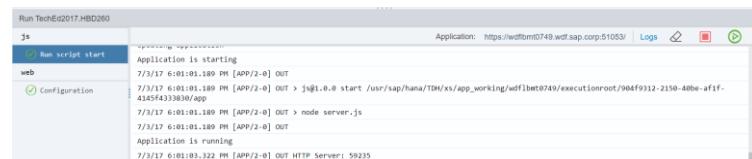
Note: if you don't want to type this code, we recommend that you cut and paste it from this web address
https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_16

};

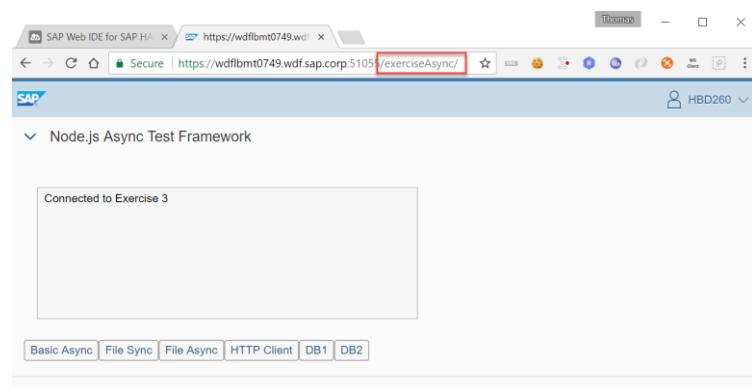
11. We can now run the js module.



12. You should see that the build and deploy was successful.



13. In the other browser tab, you should still have the web module URL from the previous exercises. Now change the path in the browser to **/exerciseAsync**. You should see the test framework for this exercise.





14. Basic Async button runs the **async.js** code. 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.

SAP

▼ Node.js Async Test Framework

Start
 End
 Wait Timer Over

[Basic Async](#) [File Sync](#) [File Async](#) [HTTP Client](#) [DB1](#) [DB2](#)

15. Test your **fileSync.js** from the UI test tool.

As you might expect, everything is output in 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 finished.

SAP

▼ Node.js Async Test Framework

This is my first file
 After First Read

This is my second file
 After Second Read

[Basic Async](#) [File Sync](#) [File Async](#) [HTTP Client](#) [DB1](#) [DB2](#)

16. Now run **fileAsync.js** from the test UI. 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.

SAP

▼ Node.js Async Test Framework

After First Read

 After Second Read

 This is my first file
 This is my second file

[Basic Async](#) [File Sync](#) [File Async](#) [HTTP Client](#) [DB1](#) [DB2](#)

17. Test your **httpClient.js** from the test UI.

Like the earlier file exercise, the after-
http call console message is output
before the response from the HTTP
request.

The screenshot shows a SAP Node.js Async Test Framework interface. At the top, there's a blue header bar with the SAP logo. Below it, a section titled "Node.js Async Test Framework" is expanded. Inside this section, there are two rows of text: "Before HTTP Call" and "After HTTP Call". Under "After HTTP Call", there is a JSON object: {"search": { "field": null, "hits": 186, "sort_order": null}}. At the bottom of the interface, there is a navigation bar with several buttons: Basic Async, File Sync, File Async, **HTTP Client**, DB1, and DB2. A mouse cursor is hovering over the DB2 button.

18. Test your **databaseAsync.js** from test UI.

As you are hopefully learning to expect, the messages you issued after the database requests are output first. Only then are the database query results returned. There is also no guarantee that query 1 will finish before query 2.

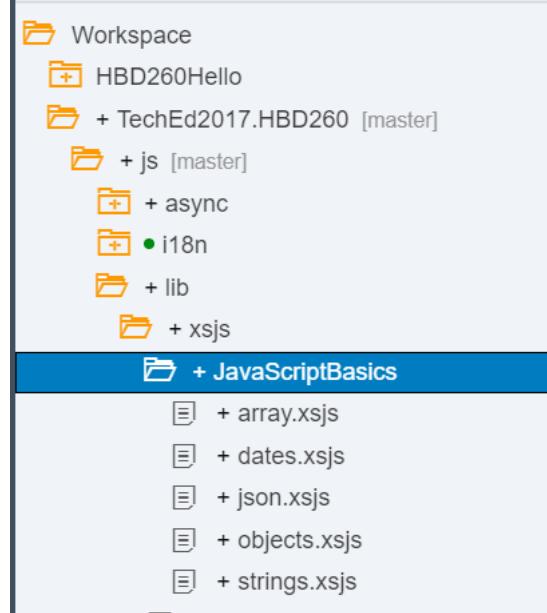
The screenshot shows a SAP Node.js Async Test Framework interface. At the top, there's a blue header bar with the SAP logo. Below it, a section titled "Node.js Async Test Framework" is expanded. Inside this section, there are several lines of text: "Before Database Call", "After Database Call", "Database Connected #1", "Database Connected #2", "Database Call #1", "--PO Header", "300000000: 13224.47", and "300000001: 12493.73". At the bottom of the interface, there is a navigation bar with several buttons: Basic Async, File Sync, File Async, HTTP Client, DB1, and DB2. A mouse cursor is hovering over the DB2 button.

19. Test your **databaseAsync2.js** from the test UI.

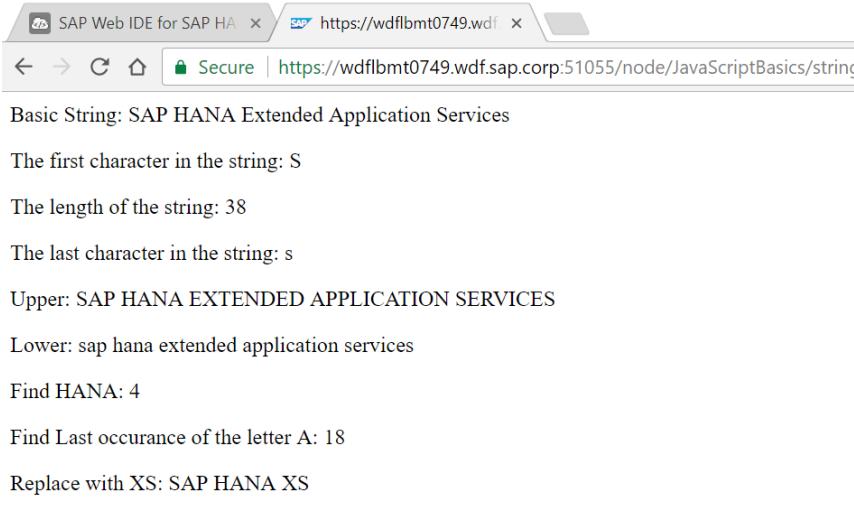
The execution is like before, but now we have the final message after all queries are complete. Because we have a sync point after all parallel execution is complete, we can output this final message after both queries are complete.

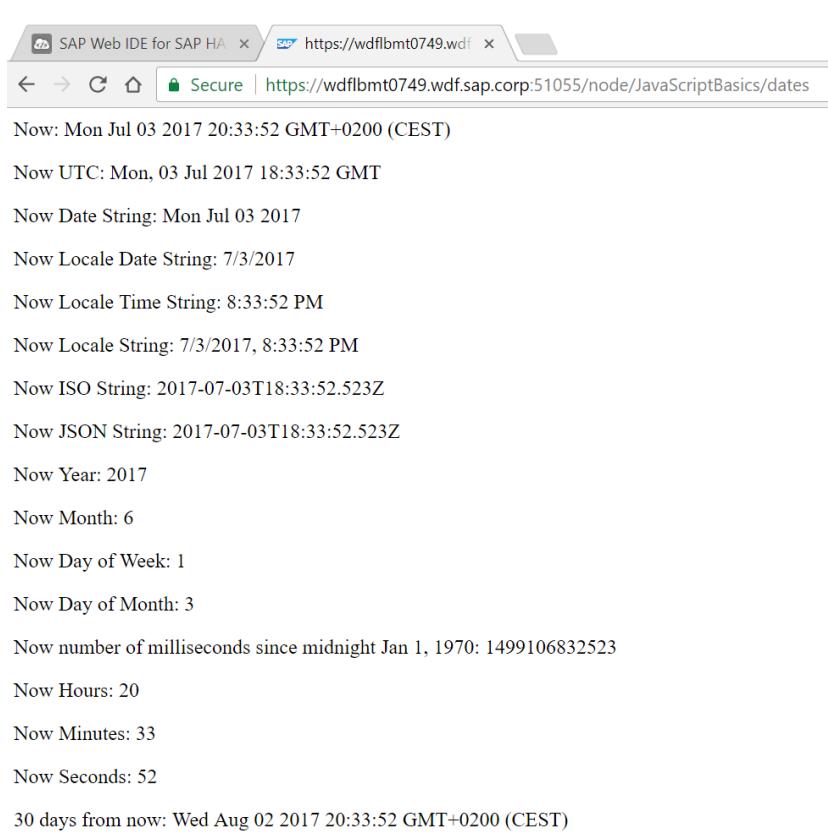
The screenshot shows a SAP Node.js Async Test Framework interface. At the top, there's a blue header bar with the SAP logo. Below it, a section titled "Node.js Async Test Framework" is expanded. Inside this section, there are several lines of text: "300000011: HT-1007", "300000012: HT-1102", "300000013: HT-1103", "Database Disconnected #2", "End Waterfall #2", and "---Everything's Really Done Now. Go Home!---". At the bottom of the interface, there is a navigation bar with several buttons: Basic Async, File Sync, File Async, HTTP Client, DB1, and DB2. A mouse cursor is hovering over the DB2 button.

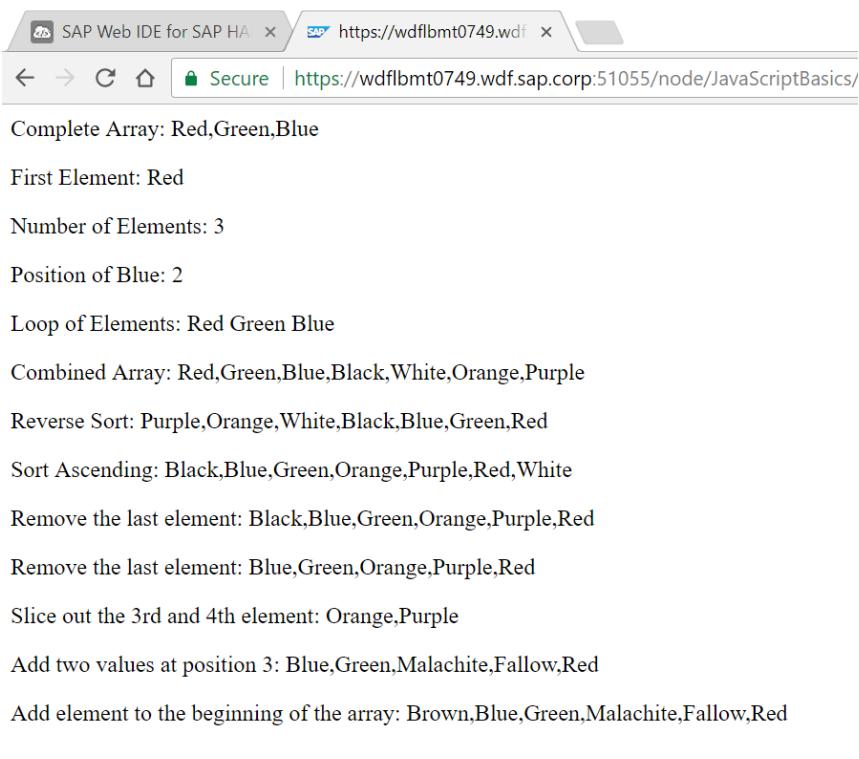
Exercise 3.4: Exploring JavaScript Language Features

Explanation	Screenshot
<p>1) Using what you have learned already, let's migrate the JavaScript Basics in Exercise 2.3 from XSJS to pure Node.js. Extend the router index.js with a path for JavaScriptBasics.</p>	<pre>"use strict"; module.exports = function(app, server) { app.use("/node", require("./routes/node")()); app.use("/node/ex1", require("./routes/ex1")()); app.use("/node/ex2", require("./routes/ex2")()); app.use("/node/excAsync", require("./routes/exerciseAsync")(server)); app.use("/node/JavaScriptBasics", require("./routes/JavaScriptBasics")()); }; /*eslint no-console: 0, no-unused-vars: 0, no-shadow: 0, new-cap: 0*/ </pre>
<p>2) In the js/router/routes folder add the file JavaScriptBasics.js with the same express router structure we've used throughout Exercise 3.</p>	<pre>"use strict"; var express = require("express"); module.exports = function() { var app = express.Router(); //Hello Router app.get("/", function(req, res) { res.send("JavaScript Basics Exercise"); }); return app; };</pre>
<p>3) This exercise will be a bit different from all others. Instead of following along step-by-step or working toward a very specific final product; this exercise will instead allow you to explore various JavaScript language features. Specifically we use what we've just learned about Node.js non-blocking, callback nature to adjust the XSJS exercises from earlier to pure Node.js</p> <p>There is no strict right or wrong solution to many of these exercise parts. However, you can view sample solutions from: https://github.com/l809764/TechEd2017.HBD260/tree/solution/js</p>	



<p><u>/router/routes/JavaScriptBasics.js</u></p> <p>Feel free to create similar Node.js routes to explore these language features. However, the main point of this exercise is to focus on general JavaScript language constructs and how to adapt them to unique aspects of Node.js.</p>	
<p>4) First we will learn about strings in JavaScript.</p> <p>Create a string with some sample text in it (hint: <code>=</code>).</p> <p>Access the first character of the string. (hint: offsets are zero based and specified via <code>[]</code>)</p> <p>Access the last character in the string (hint: <code>slice</code>).</p> <p>Output the length of the string (hint: <code>length</code>).</p> <p>Convert the string to all upper and all lower case (hint: <code>toUpperCase</code>).</p> <p>Find the position of a text part of the string (hint: <code>indexOf</code>).</p> <p>Find the last occurrence of a character in the string. Perform a find and replace operation on the string (hint: <code>lastIndexOf</code>).</p> <p>Sample implementation: <code>/strings</code></p>	

<p>5) Create a Data object with the current date (hint: new Date()).</p> <p>Convert the data to various other formats (UTC, Local Data, Local Time, ISO, JSON (hint: to*String with various modifies UTC, Date, LocalDate, LocaleTime, etc).</p> <p>Access the sub-parts of the date: Year, Month, Day, Hours, Minutes, and Seconds (hint: get* with modifiers for FullYear, Month, Day, Date, Time, Hours, Minutes, Seconds).</p> <p>Perform math on a date, adding 30 days to the current date (hint: setDate and simple math).</p> <p>Sample implementation: /dates</p>	 <pre>Now: Mon Jul 03 2017 20:33:52 GMT+0200 (CEST) Now UTC: Mon, 03 Jul 2017 18:33:52 GMT Now Date String: Mon Jul 03 2017 Now Locale Date String: 7/3/2017 Now Locale Time String: 8:33:52 PM Now Locale String: 7/3/2017, 8:33:52 PM Now ISO String: 2017-07-03T18:33:52.523Z Now JSON String: 2017-07-03T18:33:52.523Z Now Year: 2017 Now Month: 6 Now Day of Week: 1 Now Day of Month: 3 Now number of milliseconds since midnight Jan 1, 1970: 1499106832523 Now Hours: 20 Now Minutes: 33 Now Seconds: 52 30 days from now: Wed Aug 02 2017 20:33:52 GMT+0200 (CEST)</pre>
--	--

<p>6) Create two arrays: one with the colors Red, Green, and Blue and one with Black, White, Orange, and Purple (hint: ["one", "two", "three"]).</p> <p>Output the first array as a string (hint: <code>toString</code>).</p> <p>Access the first element by index (hint: zero based).</p> <p>Get the number of elements in the first array (hint: <code>length</code>).</p> <p>Get the index of Blue in the first array (hint: <code>indexOf</code>).</p> <p>Loop over the first array and output each element (hint: <code>for(var i; i < length; i++)</code>).</p> <p>Combine the two arrays (hint: <code>concat</code>).</p> <p>Reverse Sort and then Sort Ascending (hint: <code>reverse, sort</code>).</p> <p>Remove the last and then first element (hint: <code>pop, shift</code>).</p> <p>Slice out the 3rd and 4th element (hint: <code>slice</code>).</p> <p>Add two new colors (Malachite and Fallow) at the 3rd position (hint: <code>splice</code>).</p> <p>Add Brown to the start of the Array (hint: <code>unshift</code>).</p> <p>Sample implementation: <code>/array</code></p>	 <p>The screenshot shows a browser window with the URL <code>https://wdflbmt0749.wdf.sap.corp:51055/node/JavaScriptBasics/</code>. The page displays the following text output from the SAP Web IDE:</p> <pre> Complete Array: Red,Green,Blue First Element: Red Number of Elements: 3 Position of Blue: 2 Loop of Elements: Red Green Blue Combined Array: Red,Green,Blue,Black,White,Orange,Purple Reverse Sort: Purple,Orange,White,Black,Blue,Green,Red Sort Ascending: Black,Blue,Green,Orange,Purple,Red,White Remove the last element: Black,Blue,Green,Orange,Purple,Red Remove the last element: Blue,Green,Orange,Purple,Red Slice out the 3rd and 4th element: Orange,Purple Add two values at position 3: Blue,Green,Malachite,Fallow,Red Add element to the beginning of the array: Brown,Blue,Green,Malachite,Fallow,Red </pre>
---	--

- 7) Select 10 records from **PO.Header** and convert the results to JSON (hint: req.db and callbacks).

Loop over the elements in the JSON object and create a new **DISCOUNTAMOUNT** attribute by discounting the **GROSSAMOUNT** by 10% (hint: for (var i; i < length; i++)).

Convert the JSON object to a string and output it to the frontend (hint: JSON.stringify).

Sample implementation: /json

```
[  
  - {  
    PURCHASEORDERID: 30000000,  
    HISTORY.CREATEDBY.EMPLOYEEID: 20,  
    HISTORY.CREATEDAT: "2015-01-01",  
    HISTORY.CHANGEDBY.EMPLOYEEID: 17,  
    HISTORY.CHANGEDAT: "2015-01-01",  
    NOTEID: "900000001",  
    PARTNER.PARTNERID: 10000000,  
    CURRENCY: "EUR",  
    GROSSAMOUNT: "13224.47",  
    NETAMOUNT: "11113.00",  
    TAXAMOUNT: "2111.47",  
    LIFECYCLESTATUS: "N",  
    APPROVALSTATUS: "I",  
    CONFIRMSTATUS: "I",  
    ORDERINGSTATUS: "I",  
    INVOICINGSTATUS: "I",  
    DISCOUNTAMOUNT: 11902.023  
  },  
  - {  
    PURCHASEORDERID: 30000001,  
    HISTORY.CREATEDBY.EMPLOYEEID: 18,  
    HISTORY.CREATEDAT: "2015-01-02",  
    HISTORY.CHANGEDBY.EMPLOYEEID: 1,  
    HISTORY.CHANGEDAT: "2015-01-02",  
    -----  
  }]
```

- 8) Create an object literal called colors. Give it three properties – red with the value #FF0000, green with the value #00FF00, and blue with the value #0000FF. Give the object a function named favoriteColor. On Monday, it should return blue, all other days it returns red (hint: var colors = {}).

Access the colors in the object including the Favorite color (hint: object.<variable>).

Prove that objects are assigned by reference and not assigned by value (hint: just set a value to a regular variable and an object and see the difference).

Create a purchase order object for **PO.Header** which returns PO Id, gross amount, and a calculated discount amount (10% off gross amount) (hint:

Object Literals

- Red
- Blue
- Green
- Favorite Color

References

- Value 1: New Value
- Value 2: First Value
- Value 3: New Value
- Value 4: New Value

Object Constructor

Purchase Order: 30000001 Gross Amount: 12493.73 Discount Amount: 11244.357

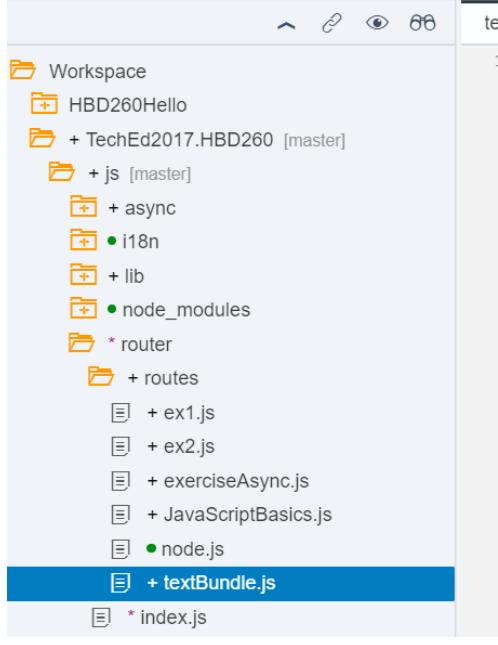
Purchase Order: 30000002 Gross Amount: 11666.69 Discount Amount: 10500.021

function with logic from last step – json.xsjs).

Create two instances of this object – one for PO 300000000 and one for 300000001 (hint: use `async.parallel` so you have a sync point after both objects are finished)

Sample implementation:
`/objects`

Exercise 3.5: Text Bundles

Explanation	Screenshot
1. Return to the js module.	
2. Add a route for a new module named textBundle that corresponds to /node/textBundle in the router/index.js	<pre data-bbox="707 483 1352 885"> "use strict"; module.exports = function(app, server) { app.use("/node", require("./routes/node")()); app.use("/node/ex1", require("./routes/ex1")()); app.use("/node/ex2", require("./routes/ex2")()); app.use("/node/excAsync", require("./routes/exerciseAsync")(server)); app.use("/node/JavaScriptBasics", require("./routes/JavaScriptBasics")()); app.use("/node/textBundle", require("./routes/textBundle")()); };</pre>
3. Create a new file in your router/routes folder called textBundle.js .	 <pre data-bbox="707 914 1205 1559"> Workspace HBD260Hello + TechEd2017.HBD260 [master] + js [master] + async • i18n + lib • node_modules * router + routes + ex1.js + ex2.js + exerciseAsync.js + JavaScriptBasics.js • node.js + textBundle.js * index.js</pre>
4. Add the following code to your textBundle.js file. It has implemented 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	<pre data-bbox="707 1591 1441 1891"> /*eslint no-console: 0, no-unused-vars: 0, consistent-return: 0, new-cap: 0*/ "use strict"; var express = require("express"); var app = express.Router(); var os = require("os"); var TextBundle = require("@sap/textbundle").TextBundle; var langparser = require("accept-language-parser"); function getLocale(req) {</pre>

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()**).

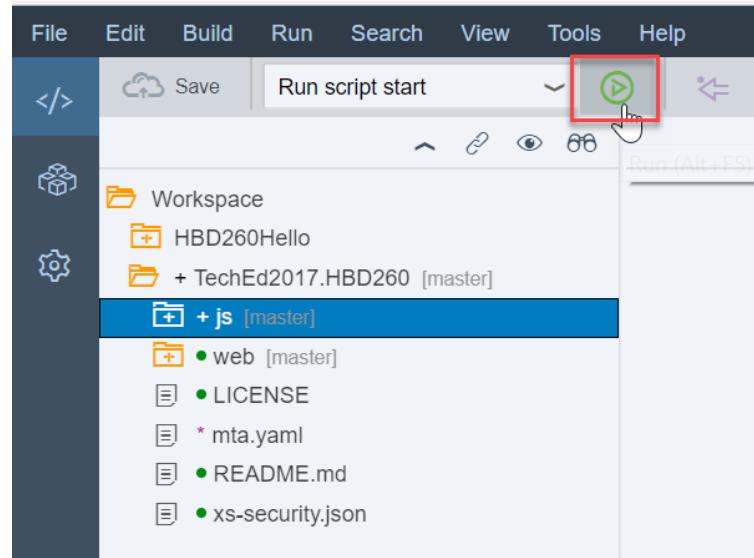
Note: if you don't want to type this code, we recommend that you cut and paste it from this web address
https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_17

```
var lang = req.headers["accept-language"];
if (!lang) {
    return;
}
var arr = langparser.parse(lang);
if (!arr || arr.length < 1) {
    return;
}
var locale = arr[0].code;
if (arr[0].region) {
    locale += "_" + arr[0].region;
}
return locale;
}

module.exports = function() {

    app.get("/", function(req, res) {
        var bundle = new TextBundle(global.__base +
            "i18n/messages", getLocale(req));
        res.writeHead(200, {
            "Content-Type": "text/plain;
charset=utf-8"
        });
        var greeting = bundle.getText("greeting",
            [os.hostname(), os.type()]);
        res.end(greeting, "utf-8");
    });
    return app;
};
```

5. We can now run the **js** module.



6. You should see that the build and deploy was successful.

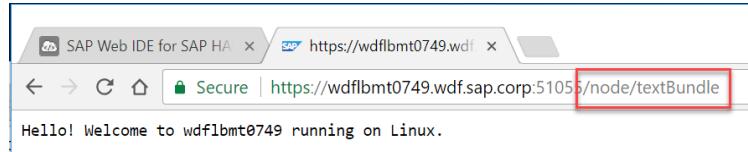
```

Run TechEd2017 HBD260
Logs Application: https://wdflbmt0749.wdf.sap.corp:51053/ Logs

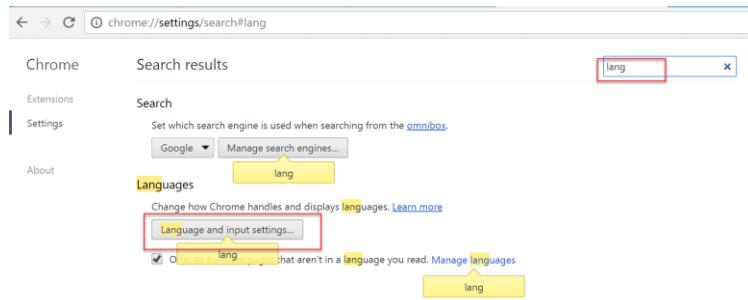
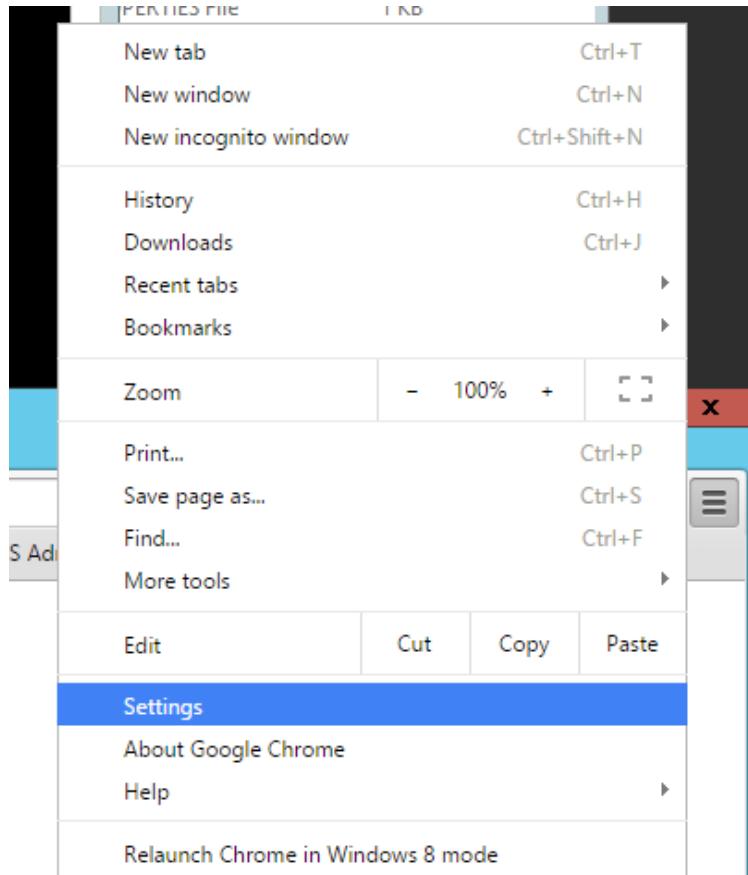
Run script start
Application is starting
7/3/17 6:45:05,650 PM [APP/2-0] OUT
Configuration
7/3/17 6:45:05,651 PM [APP/2-0] OUT > js@1.0.0 start /usr/sap/hana/TDB/xs/app_working/wdflbmt0749/executionroot/904f9312-2150-40be-
af1f-4145f4310380/app
7/3/17 6:45:05,651 PM [APP/2-0] OUT > node server.js
7/3/17 6:45:05,651 PM [APP/2-0] OUT
Application is running
7/3/17 6:45:07,785 PM [APP/2-0] OUT HTTP Server: 59235

```

7. In your other browser tab where we've been testing, change the path in the browser to **/node/textBundle**. You should see the English message output from your text file in the i18n folder.

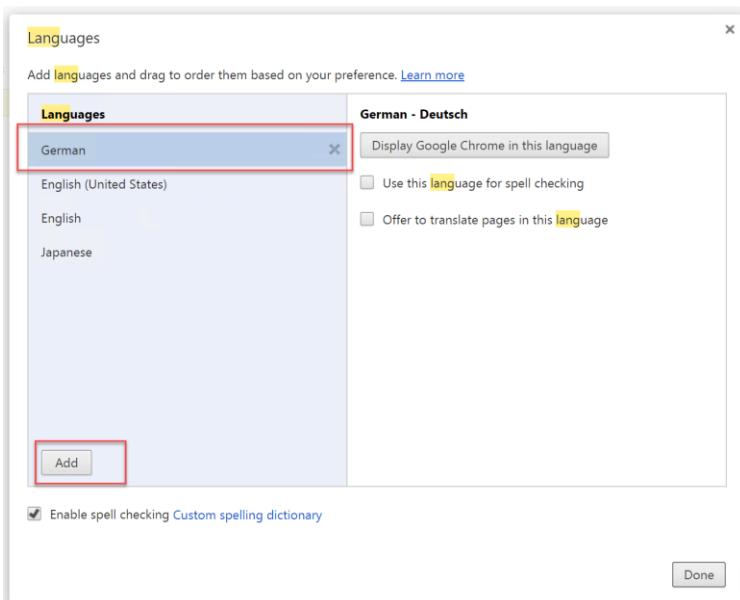


8. To test the translated strings, go into the browser Settings. Search for Lang and then choose the Language and input settings button.





9. Add German and Japanese to the language list. Drag and drop to raise German to the top of the list



10. Refresh the web browser and you should now see the German text.



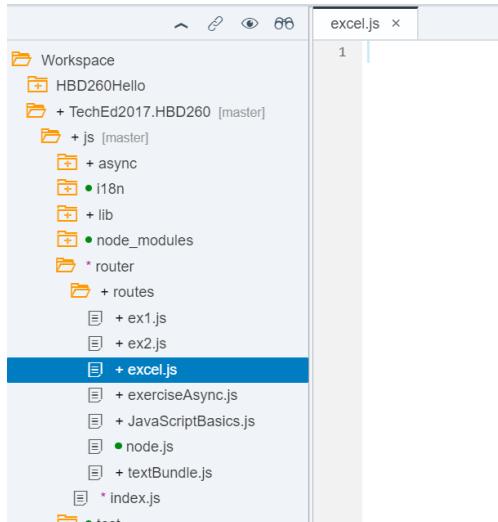
11. Repeat the process raising Japanese [ja] to the top of the list and refresh the web page



12. Please return the browser settings to English at the top.

Exercise 3.6: Open Source Modules

We've already seen how key Node.js functionality is provided by non-SAP open source modules. You've used express, async, and others already in this exercise. Now we will see other common programming tasks can be accomplished without the need of SAP provided APIs. We will convert database query results to Microsoft Excel, utilize ZIP compression, and XML parsing – all using existing open source Node.js Modules.

Explanation	Screenshot
1. Return to the js module.	
2. Add a route for a new module named excel that corresponds to /node/excel in the router/index.js	<pre>"use strict"; module.exports = function(app, server) { app.use("/node", require("./routes/node")()); app.use("/node/ex1", require("./routes/ex1")()); app.use("/node/ex2", require("./routes/ex2")()); app.use("/node/excAsync", require("./routes/exerciseAsync")(server)); app.use("/node/JavaScriptBasics", require("./routes/JavaScriptBasics")()); app.use("/node/textBundle", require("./routes/textBundle")()); app.use("/node/excel", require("./routes/excel")()); };</pre>
3. Create a new file in your router/routes folder called excel.js .	 <p>The screenshot shows a code editor interface with a sidebar displaying a file tree. The tree shows a project structure with several folders and files. The 'excel.js' file is located within the 'routes/excel' folder, which is itself within the 'routes' folder under 'router'. Other files visible include 'ex1.js', 'ex2.js', 'exerciseAsync.js', 'JavaScriptBasics.js', 'node.js', and 'textBundle.js'.</p>
4. Add the following code to your excel.js file. We are using the node-xlsx module. With one call to a function in this module we can pass in query results as JSON and receive back the binary string that	<pre>/*eslint no-console: 0, no-unused-vars: 0, no-shadow: 0, dot-notation: 0, new-cap: 0*/ "use strict"; var express = require("express"); var excel = require("node-xlsx");</pre>

contains the true Excel content format (not just tab or comma delimited string).

Note: if you don't want to type this code, we recommend that you cut and paste it from this web address
https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_18

```
var fs = require("fs");
var path = require("path");

module.exports = function() {
    var app = express.Router();

    //Hello Router
    app.get("/", function(req, res) {
        var output = "<H1>Excel
Examples</H1><br>" +
            "<a href=\"" + req.baseUrl +
        "/download\">/download</a> - Download data in Excel XLSX
format<br>";
        res.type("text/html").status(200).send(output);
    });

    //Simple Database Select - In-line Callbacks
    app.get("/download", function(req, res) {
        var client = req.db;
        var query = "SELECT TOP 10 " +
            "\"HEADER.PURCHASEORDERID\""
        as \"PurchaseOrderItemId\", " +
            "\"PRODUCT.PRODUCTID\" as
\"ProductID\", " +
            " GROSSAMOUNT as \"Amount\" " +
            " FROM \"PO.Item\" ";
        client.prepare(
            query,
            function(err, statement) {
                if (err) {

                    res.type("text/plain").status(500).send("ERROR: " +
err.toString());
                    return;
                }
                statement.exec([], {
                    function(err, rs) {
                        if (err) {

                            res.type("text/plain").status(500).send("ERROR: " +
err);
                            } else {
                                var out
= [];
                                for
(var i = 0; i < rs.length; i++) {

                                    out.push([rs[i]["PurchaseOrderItemId"],
rs[i]["ProductID"], rs[i]["Amount"]]);
                                }
                                result = excel.build([

```



```

    name: "Purchase Orders",
    data: out
  }]);

  res.header("Content-Disposition", "attachment";
  filename=Excel.xlsx");

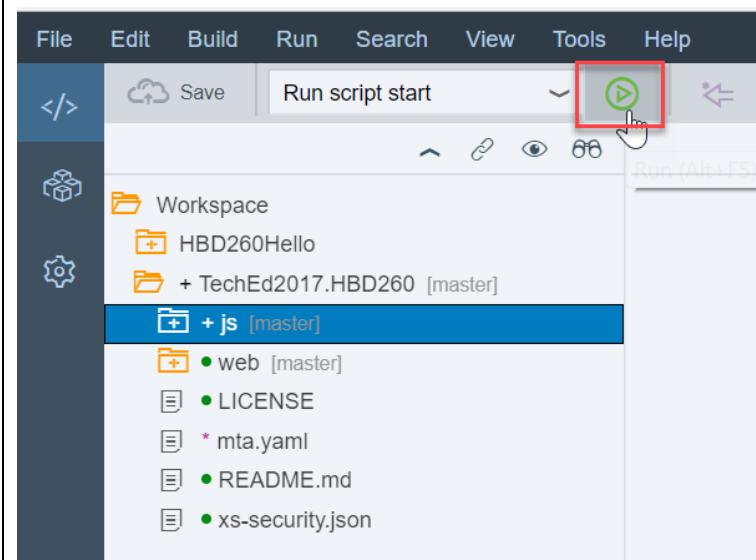
  res.type("application/vnd.ms-
  excel").status(200).send(result);
}

});

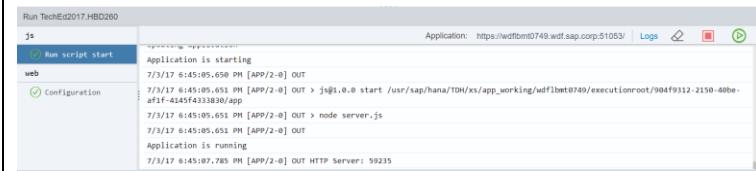
return app;
}

```

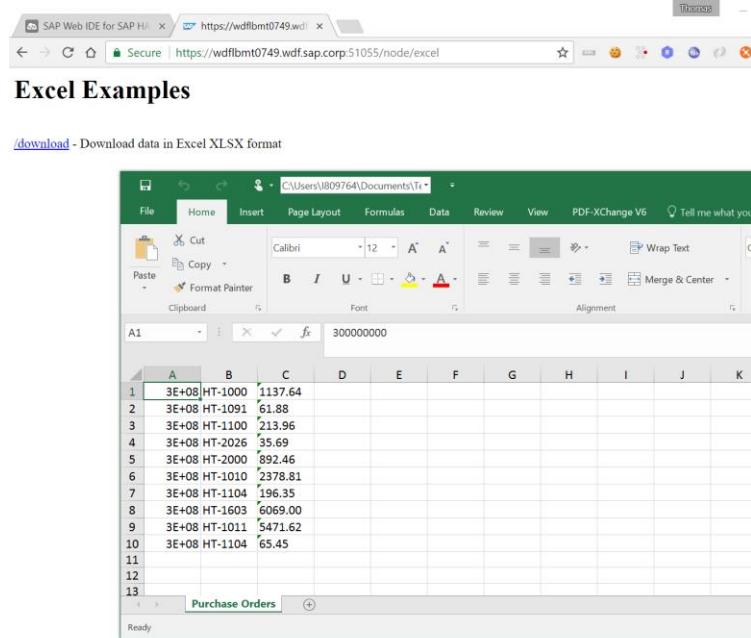
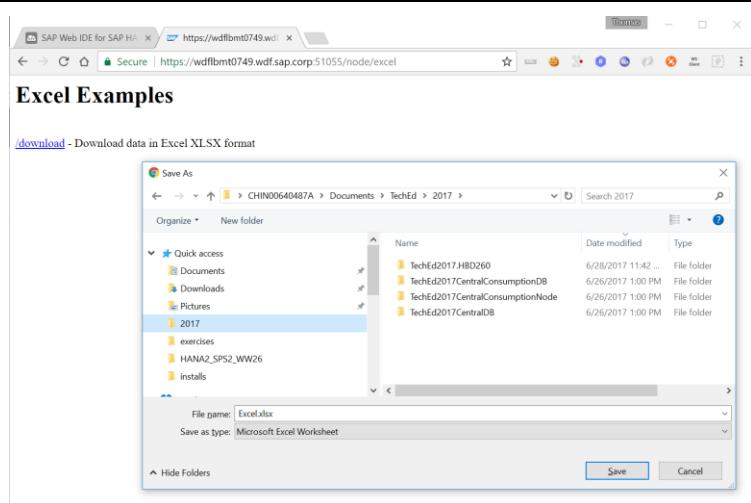
5. We can now run the **js** module.



6. You should see that the build and deploy was successful.



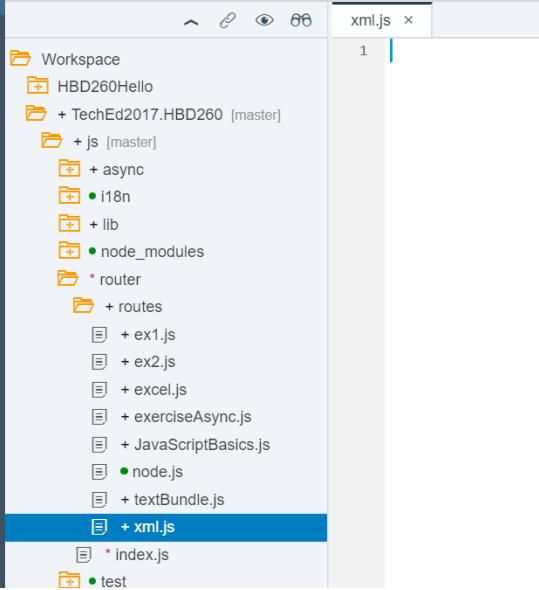
7. In your other browser tab where we've been testing, change the path in the browser to **/node/excel/download**. You should receive the file save dialog of the browser for a file named **Excel.xlsx**.



8. Return to the **js** module. Add a route for a new module named **xml** that corresponds to **/node/xml** in the **router/index.js**

```
"use strict";

module.exports = function(app, server) {
    app.use("/node", require("./routes/node")());
    app.use("/node/ex1", require("./routes/ex1")());
    app.use("/node/ex2", require("./routes/ex2")());
    app.use("/node/excAsync",
        require("./routes/exerciseAsync")(server));
    app.use("/node/JavaScriptBasics",
        require("./routes/JavaScriptBasics")());
    app.use("/node/textBundle",
        require("./routes/textBundle")());
}
```

	<pre>app.use("/node/excel", require("./routes/excel")()); app.use("/node/xml", require("./routes/xml")()); };</pre>
9. Create a new file in your router/routes folder called xml.js .	
10. Add the following code to your xml.js file. We are using the xmldoc module. This module provides XML parsing and rendering functions. We will provide a simple, hardcoded XML string and use this module to parse it and output results for each child of the root node we find.	<p>Note: if you don't want to type this code, we recommend that you cut and paste it from this web address https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_19</p> <pre>/*eslint no-console: 0, no-unused-vars: 0, no-shadow: 0, quotes: 0, new-cap: 0*/ "use strict"; var express = require("express"); var XmlDocument = require("xmldoc").XmlDocument; module.exports = function() { var app = express.Router(); //Hello Router app.get("/", function(req, res) { var output = "<H1>XML Examples</H1>
" + "/example1 - Simple XML parsing
"; res.type("text/html").status(200).send(output); }); //Simple Database Select - In-line Callbacks app.get("/example1", function(req, res) { var xml = '<?xml version="1.0" encoding="UTF-8" standalone="yes"?> <!-- this is a note --> <root> <child1>Value 1</child1> <child2>Value 2</child2> </root>'; res.type("text/xml").status(200).send(xml); }); }</pre>



```

'<note
  noteName="NoteName">' +
  heading</heading>' +
  '<to>To</to>' +
  '<from>From</from>' +
  '<heading>Note
  body = "";
  var note = new XmlDocument(xml);
  note.eachChild(function(item) {
    body += item.val + '<br>';
  });

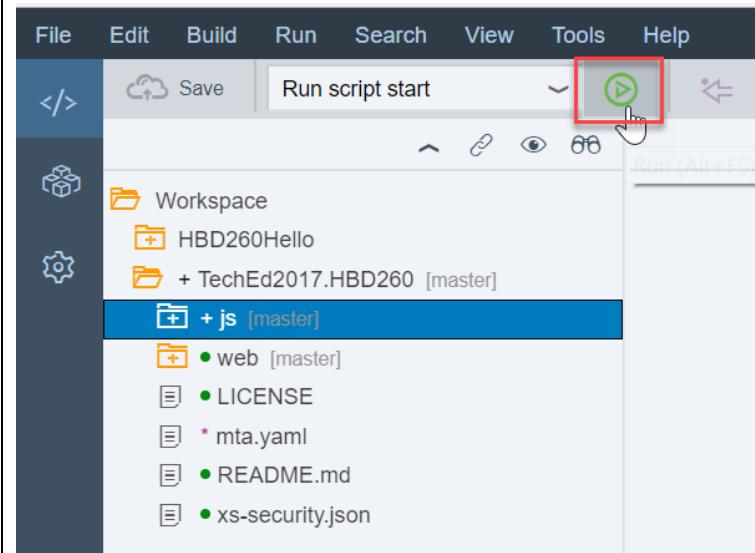
  res.type("text/html").status(200).send(body);

});

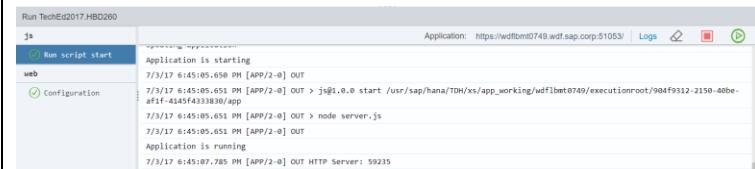
return app;
};

```

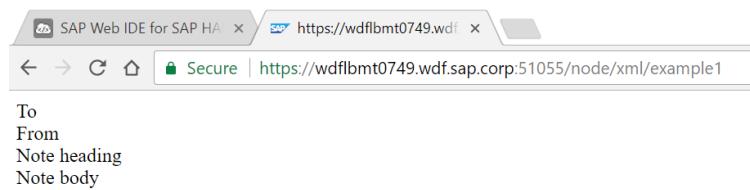
11. We can now run the **js** module.



12. You should see that the build and deploy was successful.



13. In your other browser tab where we've been testing, change the path in the browser to **/node/xml/example1**. You should see the results of parsing the sample XML content.



14. Return to the **js** module. Add a route for a new module named **zip** that corresponds to **/node/zip** in the **router/index.js**

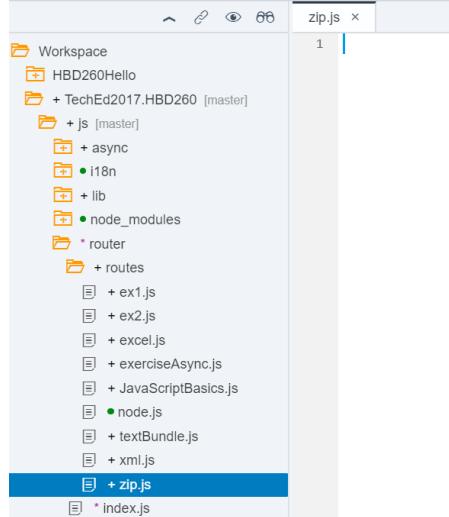
```

"use strict";

module.exports = function(app, server) {
    app.use("/node", require("./routes/node")());
    app.use("/node/ex1", require("./routes/ex1")());
    app.use("/node/ex2", require("./routes/ex2")());
    app.use("/node/excAsync",
        require("./routes/exerciseAsync")(server));
    app.use("/node/JavaScriptBasics",
        require("./routes/JavaScriptBasics")());
    app.use("/node/textBundle",
        require("./routes/textBundle")());
    app.use("/node/excel", require("./routes/excel")());
    app.use("/node/xml", require("./routes/xml")());
    app.use('/node/zip', require("./routes/zip")());
};


```

15. Create a new file in your **router/routes** folder called **zip.js**.



16. Add the following code to your **zip.js** file. We are using the **node-zip** module. With one call to a function in this module can create zip content; adding files, folders, or any content to the output zip archive.

Note: if you don't want to type this code, we recommend that you cut and paste it from this web address
https://github.com/l809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_20

```
/*eslint no-console: 0, no-unused-vars: 0, no-shadow: 0, dot-notation: 0, new-cap: 0*/
"use strict";
var express = require("express");

module.exports = function() {
    var app = express.Router();

    //Hello Router
    app.get("/", function(req, res) {
        var output = "<H1>ZIP Examples</H1><br>" +
                    "<a href=\"" + req.baseUrl +
                    "/example1">/example1</a> - Download data in ZIP format -<br>" +
                    "folder and files<br>";
        res.type("text/html").status(200).send(output);
    });

    //Simple Database Select - In-line Callbacks
    app.get("/example1", function(req, res) {

        var zip = new require("node-zip")();
        zip.file("folder1/demo1.txt", "This is the new ZIP Processing in Node.js");
        zip.file("demo2.txt", "This is also the new ZIP Processing in Node.js");
        var data = zip.generate({
            base64: false,
            compression: "DEFLATE"
        });

        res.header("Content-Disposition", "attachment;filename=ZipExample.zip");

        res.type("application/zip").status(200).send(new
Buffer(data, "binary"));

    });

    //Simple Database Select - In-line Callbacks
    app.get("/zipPO", function(req, res) {
        var client = req.db;
        var query =
            "SELECT TOP 25000
            \"PurchaseOrderId\", \"PartnerId\", \"CompanyName\",
            \"CreatedByName\", \"CreatedAt\", \"GrossAmount\" " +
            "FROM \"PO.HeaderView\" order by
            \"PurchaseOrderId\" ";
        client.prepare(
            query,
            function(err, statement) {
                if (err) {

                    res.type("text/plain").status(500).send("ERROR: " +
err.toString());
                }
            }
        );
    });
}
```

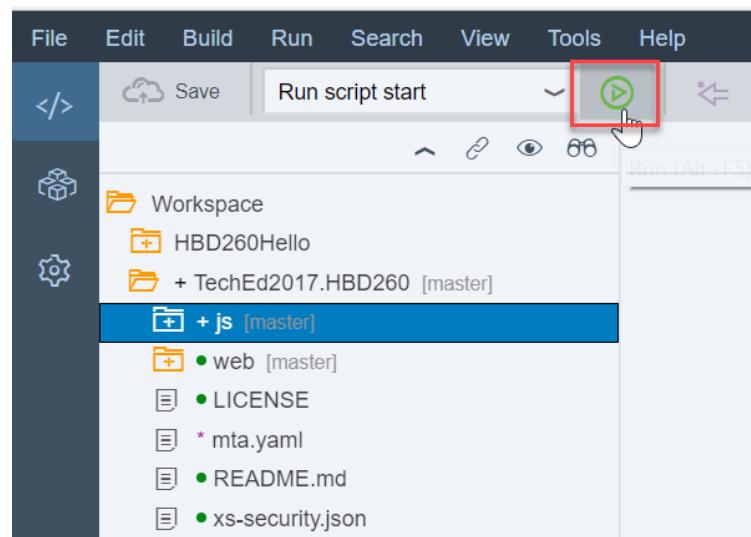
```

        return;
    }
    statement.exec([], function(err, rs) {
        if (err) {
            res.type("text/plain").status(500).send("ERROR: " + err);
        } else {
            var out = "";
            for (var i = 0; i < rs.length; i++) {
                out += rs[i].PurchaseOrderId + "\t" + rs[i].PartnerId +
"\t" + rs[i].CompanyName + "\t" + rs[i].CreatedByLoginName +
"\t" + rs[
                    i].CreatedAt + "\t" + rs[i].GrossAmount + "\n";
            }
            var zip = new require("node-zip")();
            zip.file("po.txt", out);
            var data = zip.generate({
                base64: false,
                compression: "DEFLATE"
            });

            res.header("Content-Disposition", "attachment; filename=poWorklist.zip");
            res.type("application/zip").status(200).send(new Buffer(data, "binary"));
        }
    });
    return app;
});

```

17. We can now run the **js** module.

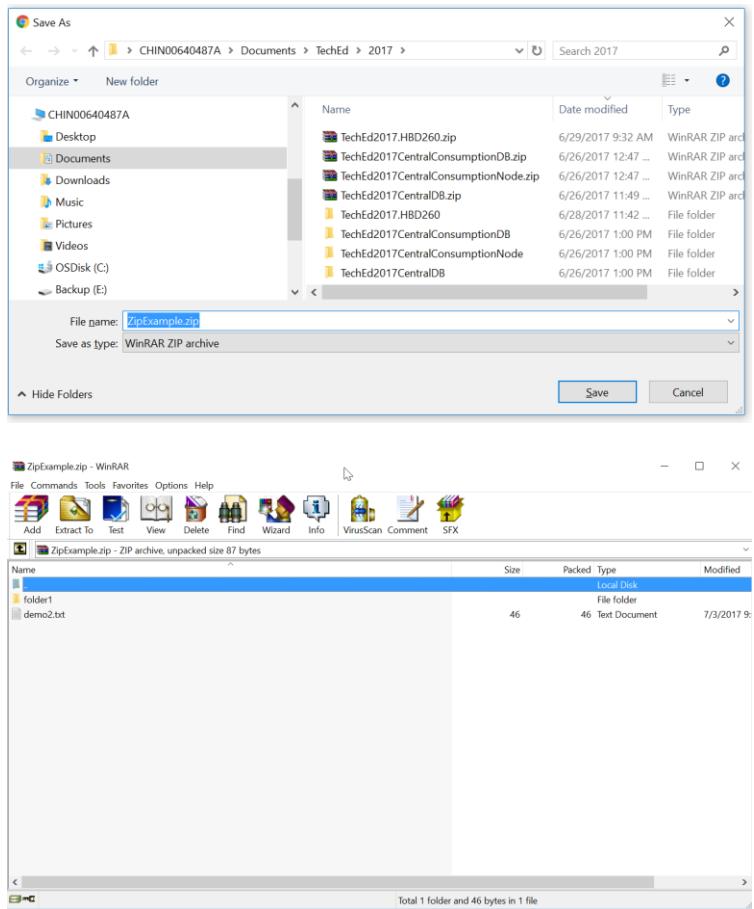


18. You should see that the build and deploy was successful.



19. In your other browser tab where we've been testing, change the path in the browser to **/node/zip/example1**. You should receive the file save dialog of the browser for a file named ZipExample.zip.

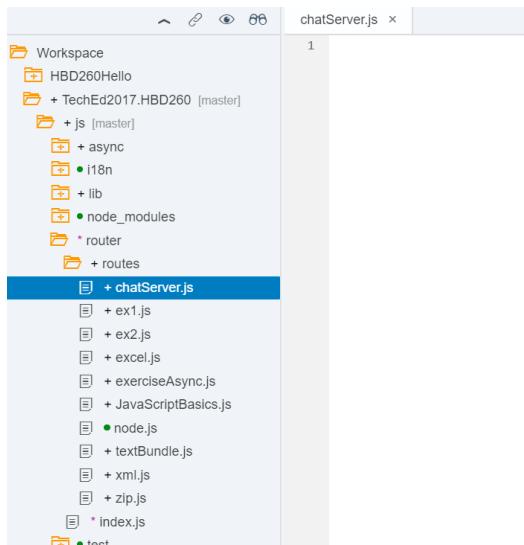
This zip file has a folder named, folder1, with a file named demo1.txt. It also has a file named demo2.txt in the root.





Exercise 3.7: Web Sockets

Our final exercise part in this section 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.

Explanation	Screenshot
1. Return to the js module.	
2. Add an express route handler for this chatServer module in the router/index.js and pass the server variable in as well.	<pre data-bbox="707 631 1394 1189"> "use strict"; module.exports = function(app, server) { app.use("/node", require("./routes/node")()); app.use("/node/ex1", require("./routes/ex1")()); app.use("/node/ex2", require("./routes/ex2")()); app.use("/node/excAsync", require("./routes/exerciseAsync")(server)); app.use("/node/JavaScriptBasics", require("./routes/JavaScriptBasics")()); app.use("/node/textBundle", require("./routes/textBundle")()); app.use("/node/excel", require("./routes/excel")()); app.use("/node/xml", require("./routes/xml")()); app.use("/node/zip", require("./routes/zip")()); app.use("/node/chat", require("./routes/chatServer")(server)); };</pre>
3. Create a new file in your js/router/routes folder called chatServer.js .	 <pre data-bbox="707 1210 1231 1755"> chatServer.js x 1 + Workspace + HBD260Hello + TechEd2017.HBD260 [master] + js [master] + async + i18n + lib + node_modules + router + routes + chatServer.js + ex1.js + ex2.js + excel.js + exerciseAsync.js + JavaScriptBasics.js + node.js + textBundle.js + xml.js + zip.js * index.js + test</pre>

-
4. Add the following code to your chatServer.js file.

The Node.js module for Web Sockets which we are going to use is **ws**. Require it and create a new instance of the **WebSocketServer**.

This is then a recommended implementation for the remainder of the Web Sockets functionality to both receive and send messages.

Note: if you don't want to type this code, we recommend that you cut and paste it from this web address
https://github.com/1809764/TechEd2017.HBD260/blob/snippets/ex3/ex3_21

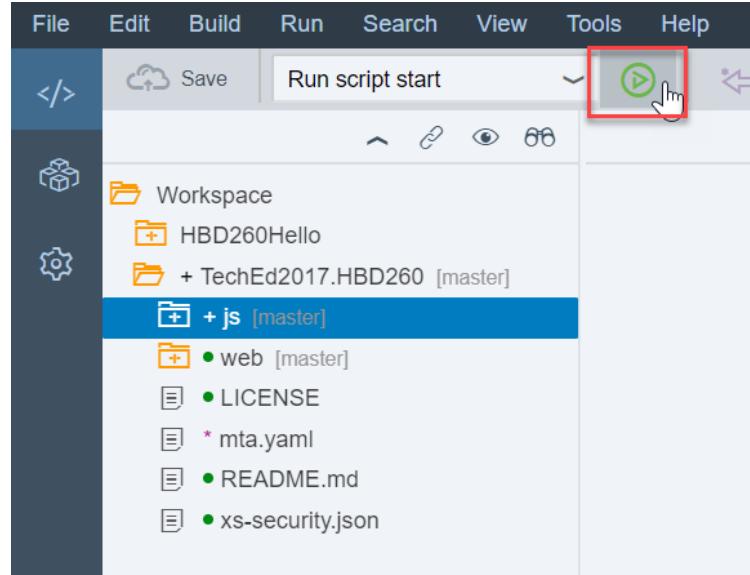
```
/*eslint no-console: 0, no-unused-vars: 0, new-cap: 0*/
"use strict";
var WebSocketServer = require("ws").Server;
var express = require("express");

module.exports = function(server) {
    var app = express.Router();
    app.use(function(req, res) {
        var output = "<H1>Node.js Web Socket Examples</H1><br>" +
                    "<a href='/exerciseChat'>/exerciseChat</a> - Chat Application for Web Socket Example<br>";
        res.type("text/html").status(200).send(output);
    });
    var wss = new WebSocketServer({
        server: server,
        path: "/node/chatServer"
    });

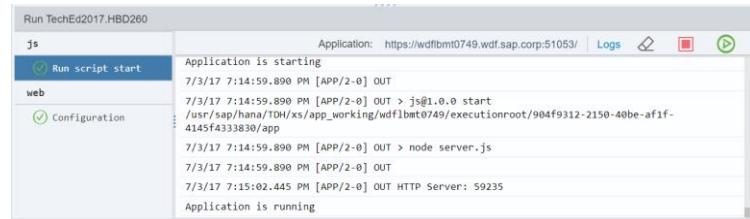
    wss.broadcast = function(data) {
        wss.clients.forEach(function each(client) {
            try {
                client.send(data);
            } catch (e) {
                console.log("Broadcast Error: %s", e.toString());
            }
        });
        console.log("sent: %s", data);
    };

    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"
        }));
    });
    return app;
};
```

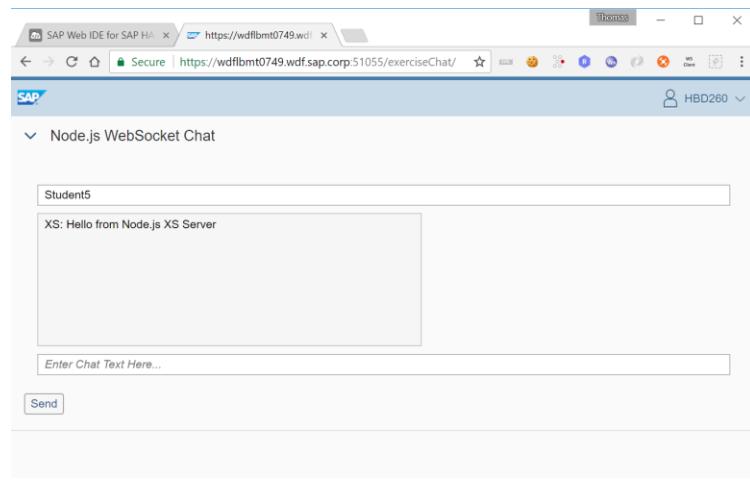
5. We can now run the **js** module.



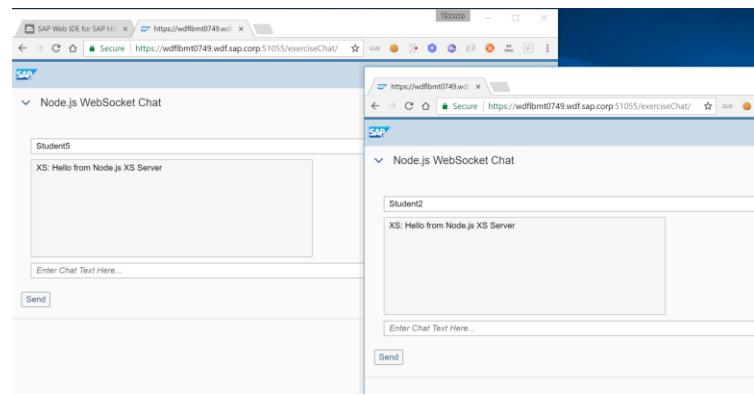
6. You should see that the build and deploy was successful.



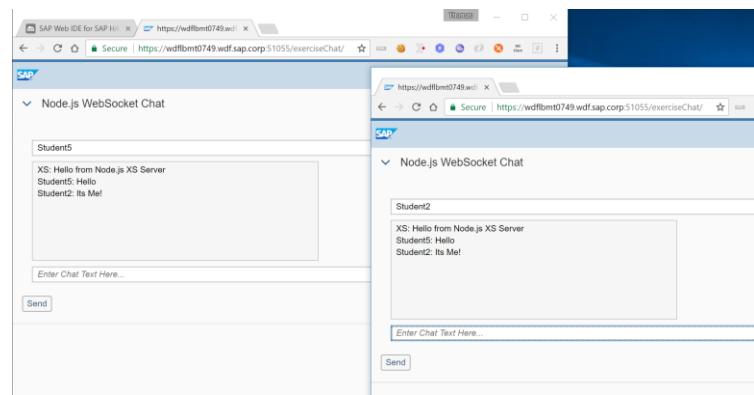
7. In the browser tab, we've been using for testing, change the path in the browser to **/exerciseChat**. You should see the simple chat user interface.



-
8. Open a second browser window and cut and paste the chat application URL into it.



-
9. 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.





www.sap.com/contactsap

© 2017 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.

The information contained herein may be changed without prior notice. Some software products marketed by SAP SE and its distributors contain proprietary software components of other software vendors.
National product specifications may vary.

These materials are provided by SAP SE or an SAP affiliate company for informational purposes only, without representation or warranty of any kind, and SAP or its affiliated companies shall not be liable for errors or omissions with respect to the materials. The only warranties for SAP or SAP affiliate company products and services are those that are set forth in the express warranty statements accompanying such products and services, if any. Nothing herein should be construed as constituting an additional warranty.

In particular, SAP SE or its affiliated companies have no obligation to pursue any course of business outlined in this document or any related presentation, or to develop or release any functionality mentioned therein. This document, or any related presentation, and SAP SE's or its affiliated companies' strategy and possible future developments, products, and/or platform directions and functionality are all subject to change and may be changed by SAP SE or its affiliated companies at any time for any reason without notice. The information in this document is not a commitment, promise, or legal obligation to deliver any material, code, or functionality. All forward-looking statements are subject to various risks and uncertainties that could cause actual results to differ materially from expectations. Readers are cautioned not to place undue reliance on these forward-looking statements, and they should not be relied upon in making purchasing decisions.

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. All other product and service names mentioned are the trademarks of their respective companies. See <http://www.sap.com/corporate-en/legal/copyright/index.epx> for additional trademark information and notices.

