

AMIS SIG Introduction Node.js–Hands-on

15th May 2017

In this hands-on, you will get going with Node.js. You will walk through some of the basics – running your first node application (hello world!), working with the file system and other core modules and with some packages downloaded through npm.

Then we step it up a little with some practices around http. You will create a simple static file server as well as some programs that handle more interesting http requests, similar to Java servlets. We look at making http call outs from the Node.js program in addition to listening and responding to incoming calls.

Then we introduce the Express framework – the more or less de facto standard web application framework for Node.js applications that serve UI or API oriented applications. We will look at the implementation of simple REST API.

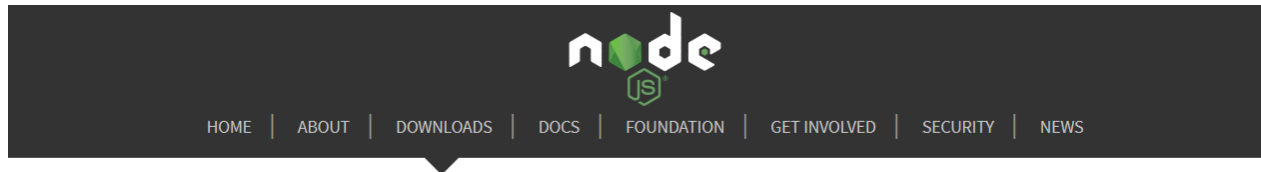
One example of a rich client web application that can be served from Node.js is the Oracle JET application that you will work on a little. Note that similar applications, for example based on Angular2 or other frameworks, can run from Node.js just as easily.

Somewhat more advanced topics are included as bonus practices, such as sending emails, gathering user input from the command line, implement server to client push with server sent events. We look at asynchronous programming and callback challenges and introduce new language features such as promises and generators and mechanisms for parallel processing (the latter applied in an artist api that provides enriched artist information by leveraging APIs from Spotify). We also discuss the interaction between Node and NoSQL Database MongoDB. After briefly touching upon using Node in Docker Containers, we also look at the Node driver for Oracle DB Database Driver; this driver allows us to access an Oracle Database from a Node.js program, similar to the way JDBC allows us to access a database from Java applications.

You can get access to the sources for the practices from the GitHub repository:.

1. Installation of Node.js

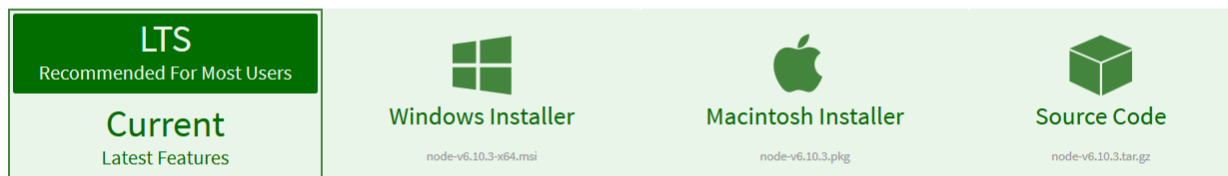
Native versions of Node.js are available for Mac OS X, Windows and Linux. Go to <https://nodejs.org/en/download/> and download the installer for your platform.



Downloads

Latest LTS Version: v6.10.3 (includes npm 3.10.10)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.



Windows Installer (.msi)
Windows Binary (.zip)
macOS Installer (.pkg)
macOS Binaries (.tar.gz)
Linux Binaries (x86/x64)
Linux Binaries (ARM)
Source Code

32-bit	64-bit
32-bit	64-bit
64-bit	
64-bit	
32-bit	64-bit
ARMv6	ARMv7
node-v6.10.3.tar.gz	

Next, run the installer to perform the installation. You can check on the success of the installation by opening a command line window or terminal and typing `node -v`. This should result in an indication of your current version of Node.js – which should be 6.x.y:

```
C:\Users\lucas_j>node -v
v6.9.4
```

Alternatively you can also work in a Docker container. The official Docker image for Node.js is on https://hub.docker.com/_/node/. This page also contains instructions on how to build a container with Node.js set up inside, and run it.

A third way to go is to use a prebuilt Virtual Machine image. One source for such images is Bitnami: <https://bitnami.com/stack/nodejs/virtual-machine> (and this article explaining how to load the VM Ware images into Virtual Box: https://wiki.bitnami.com/Virtual_Appliances_Quick_Start_Guide#How_to_start_your_Bitnami_Virtual_Appliance.3f).

A fourth option of working with Node.js is in a Cloud environment, such as Google App Engine, Azure, Heroku, AWS and Oracle Application Container Cloud. Given the light weight nature of node.js you will be able to use the “free tier” resources in most cases.

We will not explore these alternative options in this workshop.

In section 7 of this hands-on document are we going to be (optionally) using a Pre Built VM – for Oracle Database development. You can skip ahead to that section, read how to get this VM installed and running and how to configure it for using Node.js and use that VM as your Node.js development environment.

Editing Node programs can be done in any text editor. Some editors are ‘Node aware’ – that means they can format Node code, check the syntax of the code and even run and debug the code. A popular editor for developing Node programs is Visual Studio Code from Microsoft – free, open source, available on Windows, MacOS and Linux. For this workshop, I suggest that you download and install Visual Studio Code – from <https://code.visualstudio.com/> .

We will be looking at the implementation of REST APIs with Node. In order to verify the success of our work, we will want to invoke those REST APIs. This can be done from a browser (for GET requests), using curl (from the command line) and using many other tools. A popular tool for making such REST requests is Postman. Some labs in this workshop will include Postman collections – test sets with REST API calls. I therefore strongly recommend that you install Postman: <https://www.getpostman.com/> .

2. Basic steps with Node.js

A Node.js application is a JavaScript application (more formally: ECMA Script v6 or ES6) that can be run - outside the browser – on the Node.js platform [which contains the V8 engine for JavaScript].

A node application – the informal way of referring to Node.js applications – is simply run on the command line using the command:

node app.js

where app.js is the presumed name of the (main) application script. The name is yours to decide. Other common names include main.js and of course hello-world.js.

In this section, you will edit, create and run a few simple Node applications.

- a) Open a command line (terminal) in the part1-hello-world directory of the workshop resources.
Run

```
node hello-world.js
```

and see what happens. (well, the expected of course!). Open the file hello-world.js. No big surprises?

- b) Most Node programs will be more structured than just a bunch of lines of code. Creating functions is a level of structuring found in almost all Node programs. The hello-world-2.js application does the same thing as hello-world.js, in a more structured way.

```
node hello-world2.js
```

- c) Run hello-world-3.js and pass a command line parameter. For example:

```
node hello-world-3.js Johan
```

See what the result is. Inspect the code in hello-word-3.js. You will see how node applications can get access to the command line parameters and how functions can be used to organize program logic. Again, no big surprises.

- d) Now also check hello-world-4.js. This application has the same functionality as the previous one. The interesting thing is that a [reference to a] function is assigned to a variable. And that this reference is leveraged to invoke the function. Passing around references to functions is not

necessarily an everyday affair in all program languages you may have encountered.

Open `hello-world-5.js`. The functionality is still the same as the previous two programs. New is the use of function *reception*. This function is the unit where greeting of visitors is performed. The reception is strictly instructed as to how to perform the greeting. In fact, the function it should use for greeting is injected. Another example of passing around a reference to a function.

`hello-world-6.js` shows to we can use anonymous functions instead of named functions. Just so you know.

- e) Scheduling functions for delayed and possibly periodic execution is easily done in JavaScript and therefore in Node applications. Using `setTimeout()` and `setInterval()`, a reference to function is associated with a time period expressed in milliseconds to schedule the execution of the function. Check out `hello-world-7a.js` for examples.

```
node hello-world-7a.js
```

Notice how you may think that the program is complete – when in fact it is still running.

When the scheduled function has associated data – input parameters that are used in its execution – it is not trivial to pass that data context into the function. A naïve approach is shown in `hello-world-7b.js` – and hopelessly fails.

- f) Program `hello-world-8.js` deserves special attention. It shows how we not only can pass around references to functions but to the combination of a function and its [data] context. This package – function plus context – is called a *closure*. And that is one of the hot concepts in programming in recent times.

Try:

```
node hello-world-8.js kwik kwek kwak
```

and see the outcome.

Now take (another) look at the code. In the `args.forEach()` call, for each program argument there is a function registered through `setTimeout` to be executed at a later point in time. The function to be executed – when the timeout happens – is returned from `getGreeter()`. For `forEach()` invokes `getGreeter()` to return a function and registers that function with `setTimeout()`. Note however that `getGreeter()` does not just return a function: it returns a functions with a reference to the local `toGreet` variable. And here we have the closure: the combination of the function – a reference to the function defined in `var g` – and its context: the variable `toGreet` at the time of returning the capture.

getGreeter() is invoked for each command line argument. This results in a distinct capture for each of these arguments, with distinct values for the toGreet variable, part of the closure.

- g) The program hello-world-8a.js does the same thing as hello-world-8.js, except that the code is organized differently. The program leverages a module, referred to as *greeter* and imported from the file greeter-module.js.

Check how the functionality in the file greeter-module.js is exposed – and what is not exposed. Verify how in hello-world-8a.js we get access to what is exposed by the module.

Remove the call to the localPrivateFunction(). Add a function welcomeAll() to the greeter-module.js file. Have this function write a single line to the console, to welcome everyone: 'Welcome y'all!'. Expose this function from the module.

Invoke this new function welcomeAll() to hello_world-8a.js.

- h) A small additional step: hello-world-9.js adds our first core Node module. By adding the first line in the program:

```
var util = require('util');
```

we can now make use of the functionality in the core module *util* in the program. Examples are: util.log and util.format.

See documentation about this core module: <https://nodejs.org/dist/latest-v6.x/docs/api/util.html> .

- i) File manipulation can easily be done in Node.js applications. The program file-writer.js is an example: it writes a file with all command line arguments on separate lines in the file. And it does so in very few lines of code – just check out: file-writer.js.

Also run this program and inspect the file that gets created:

```
node file-writer.js kwik kwek kwak
```

3. Handling HTTP Requests

One of the key strengths of Node.js is its ability to help out with all kinds of HTTP(S) interactions. In the next section we will make use of a package (Express) that adds very convenient capabilities on top of core Node.js. For now, we will stick to the core functionality available out of the box.

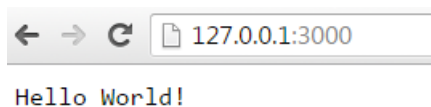
The sources discussed in this section are in directory part2-http.

- a) The first program creates an HTTP Web Server that listens for HTTP requests on port 3000 and returns a static response to any request. You can test this server by calling <http://127.0.0.1:3000> from your browser, CURL, wget, SoapUI, Postman or any other HTTP speaking tool.

Run:

```
node http.js
```

and access the URL as described above.



This may sound daunting: creating an HTTP server. Wow. That must be quite a lot of work. And with Node.js of course it is not. Inspect the code in http.js. Note how the core module http is used – require is similar to import.

Change the response to: Bonjour Monde!

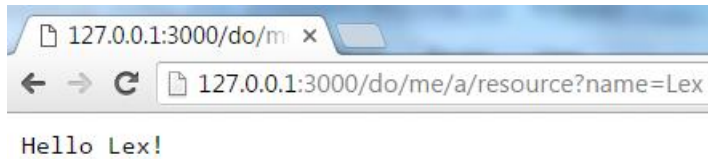
- b) With this first major step to easily made, we may get greedy. What about processing query parameters in the URL – the HTTP equivalent to command line parameters? Easy. And we will tackle URL paths at the same time.

Run http-2.js:

```
node http-2.js
```

and access the url: <http://127.0.0.1:3000/do/me/a/resource?name=Lex>

Check the HTTP response:



And check in the command line (terminal) window:

```
C:\data\sigs-node-js-31march2016\sigs-nodejs-amis-2016\part2-http>node http-2.js
server running on port 3000
URL /do/me/a/resource?name=Lex
path: /do/me/a/resource
queryObject: {"name":"Lex"}
URL /favicon.ico
path: /favicon.ico
queryObject: {}
```

We can see how this node application could easily learn about both query parameters and URL path.

Now open http-2.js in a text editor and analyze the code – not there is a lot of it.

- c) The program http-3.js will return the file index.html in the public folder for any HTTP request to port 3000. Try it out.

Note how core module fs is leveraged to read the file, alongside core module http to handle the http request. Also note that any HTTP request sent to port 3000 will get this same response – regardless of URL path, query parameters and HTTP method.

- d) Program http-4.js also returns the same response to any HTTP request. However, instead of returning the contents from a local file as the response, it goes out to fetch a resource from the internet. Just run http-4.js:

```
node http-4.js
```

and access port 3000 on the local host.

Note: it takes quite a while to load this document. If you add a log statement to the program to log each request – and better yet: the URL for each request – you would understand why this takes so much longer than expected – and why the page looks so much poorer than expected. Hint: `console.log(req.url);`

4. Express Web Application Framework

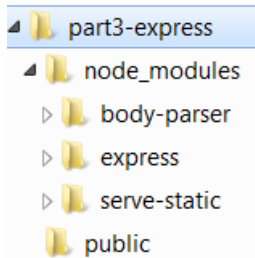
Many Node.js applications solve a similar challenge: handling HTTP requests. This happens for example for static file serving, rich client web applications and for the implementation of REST APIs. Although the core capabilities in Node.js for dealing with HTTP interactions, with the Express framework developers get even better facilities for constructing Node.js applications that handle HTTP requests.

In this section, we will get introduced to Express. We will use the resources in folder part3-express.

- a) As a first step, you need to install a few packages that we will use on top of core Node.js. On the command line (in the terminal window), in the part3-express directory, please use npm to install three packages, like this – or skip to the section just below the next figure:

```
npm install express  
npm install body-parser  
npm install serve-static
```

After you have executed these three installation steps, verify that a subfolder node_modules was created. Inspect the contents of this folder.



Alternatively – and now I tell you – you could have simply used:

```
npm install
```

The npm tool would have inspected the package.json file in your current directory, would have found three dependencies and installed those dependencies. Note that you can use npm install in combination with a package.json file at any time to refresh the dependencies and bring in the latest (allowed) versions of these packages.

- b) The simplest web application we can run with Express is defined in express.js. Run this program and open URL <http://127.0.0.1:3000>. A static response is returned.

express-2.js does exactly the same, in an even more compact manner. Check out the source.

- c) The most compactly code web server you have ever seen: open express-3.js and behold! Run express-3.js and access the same URL: <http://127.0.0.1:3000> from the browser.

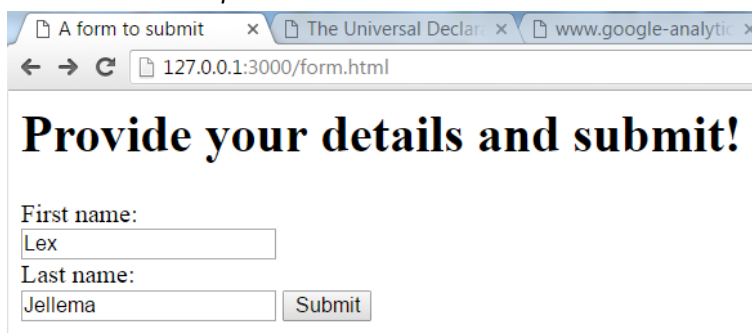
Note how the browser shows an image – also fetched from the Node.js application. Click on the download link for the PDF document. This document too is returned by the Node.js application.

Where do these resources come from?

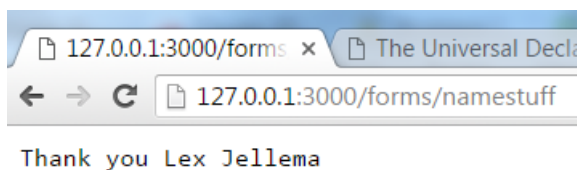
- d) Inspect the sources for express-4.js. Here we handle a form submission. This source more or less belongs together with the HTML source form.html in directory public. Check out that file. try to understand how the form and the node application fit together.

Now run express-4.js and access <http://127.0.0.1:3000> from the browser.

Click on the link *Open Form*. Fill in the form fields and click on Submit.



You will get thanked for your submission; the thank you messages uses the values you had submitted from the form.



- e) REST APIs are an important area for Node.js. The application express-5.js is an example of a simple REST API. Run this application.

Then access the URL <http://127.0.0.1:3000/departments> . You will get a JSON response with a list of departments.

Pick one of the department identifier values from the and access URL: http://127.0.0.1:3000/departments/the_value_you_picked (for example <http://127.0.0.1:3000/departments/10>). You will get a detail response: a single department record.

The application behind this REST API is about 20 lines long. That is it. 20 lines. That is a good time to say: wow!

Open express-5.js and analyze the code.

The data exposed by this API is loaded from a static file on the local file system into a variable *departments*, when the application is started. Handler functions are registered with Express for GET requests to the URL paths /departments and /departments/:departmentId. The functions perform simple yet effective actions in response to requests.

- f) Try to extend express-5.js in the following way: GET requests for the URL path /time should return a response with the current time. Hint: a string representation of the current time can be created using `new Date().toUTCString()`.
- g) Program express-6.js has maybe six lines more than express-5.js. Just so you know. Now run express-6.js.

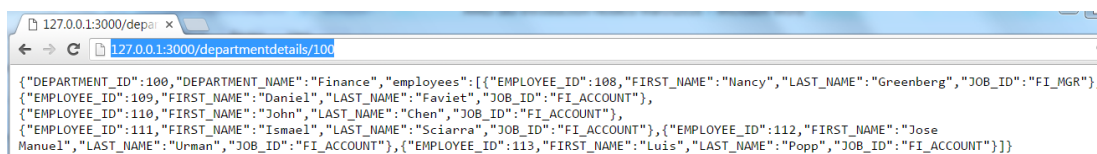
List all departments using: <http://127.0.0.1:3000/departments>.

Now open the department form: <http://127.0.0.1:3000/departmentForm.html> . Enter details for a new department and press submit.

Next, list all departments again, using <http://127.0.0.1:3000/departments> . You should see the new department added to the collection.

This is a very simple example of a REST API that not only supports GET but POST as well. Inspect the source of express-6.js to see how this is done.

- h) Express-7.js adds the *departmentdetails* resource – accessible at http://127.0.0.1:3000/departmentdetails/DEPARTMENT_ID, for example <http://127.0.0.1:3000/departmentdetails/100>. Run express-7.js and try out this department details resource.



Then inspect the source for express-7.js. Where do the data for *departmentdetails* come from?

You will probably figure out soon enough that the data is retrieved from an external API. So here we have an example of a Node.js application making an external HTTP(S) call. The most interesting bits:

- see how the request details (host, port, path and method) are configured
- see how the response to this request is handled in callback function that receives the response and how event listeners are registered for the data, end and error events
- see how the final response is created (`res.send(JavaScript record)`) in the *end* event handler

5. Run Rich Client Web application on and from Node.js

Rich client web applications do their work in the client – the web browser. The role of the middle tier is reduced for these applications to exposing APIs (REST/JSON) that the rich client application can make use of. The middle tier can also add what is sometimes called mBaaS – Mobile Backend as a Service – to handle various features that mobile apps benefit from (device management, proxying, analytics, caching, push notification, authentication,...). Some of these mechanisms may be useful for ‘regular’ [rich client] web applications to.

One middle tier role is indispensable for now: serving the [static] rich client application’s resources – the HTML, JS, CSS and image files, at least the first time the application is started in a browser. Note that static file serving is a very trivial affair with Node.js. It scales well, serves files well and is easy to configure. It can do what Nginx can do – and more. And this *more* is why Node.js is often used as the web server (and REST API provider) for rich client web applications.

In this section we will quickly create an Oracle JET application – just as an example of a rich client web application - and run it on Node.js. Next, we will add a data bound section to the application and hook it up with a REST API that we inject into the Node.js backend for the application.

- a) Create a new directory part4. Navigate to that directory in the command line [terminal]. Install express and express-generator using npm:

```
npm install express
npm install express-generator -g
```

You may see many error messages. It is probably okay to ignore them.

- b) Generate a new express application called jet-on-node:

```
express jet-on-node
```

A new application scaffold is generated for *jet-on-node*.

```
c:\data\workshop-node-js-15may2017\part4>express jet-on-node
```

```
create : jet-on-node
create : jet-on-node/package.json
create : jet-on-node/app.js
create : jet-on-node/public/stylesheets
create : jet-on-node/public/stylesheets/style.css
create : jet-on-node/public
create : jet-on-node/public/javascripts
create : jet-on-node/public/images
create : jet-on-node/views
create : jet-on-node/views/index.jade
create : jet-on-node/views/layout.jade
create : jet-on-node/views/error.jade
create : jet-on-node/routes
create : jet-on-node/routes/index.js
create : jet-on-node/routes/users.js
create : jet-on-node/bin
create : jet-on-node/bin/www
```

```
install dependencies:
> cd jet-on-node && npm install
```

```
run the app:
> SET DEBUG=jet-on-node:* & npm start
```

This has nothing yet to do with Oracle JET – you can do this for any Express application!

- c) Navigate to directory *jet-on-node*. Add dependencies to the application using:

```
npm install
```

This installs a number of node modules. You can check directory *part4\jet-on-node\node-modules* to get an impression.

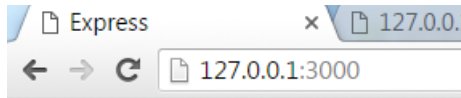
```
c:\data\workshop-node-js-15may2017\part4\jet-on-node>npm install
jet-on-node@0.0.0 c:\data\workshop-node-js-15may2017\part4\jet-on-node
+-- body-parser@1.13.3
| +-- bytes@2.1.0
| +-- content-type@1.0.2
| +-- depd@1.0.1
| +-- http-errors@1.3.1
| | +-- inherits@2.0.3
| | +-- statuses@1.3.1
| +-- iconv-lite@0.4.11
| +-- on-finished@2.3.0
| | +-- ee-first@1.1.1
| +-- qs@4.0.0
| +-- raw-body@2.1.7
| | +-- bytes@2.4.0
| | +-- iconv-lite@0.4.13
| | +-- unpipe@1.0.0
| | +-- type-is@1.6.15
| | +-- media-typer@0.3.0
| | +-- mime-types@2.1.15
| | +-- mime-db@1.27.0
+-- cookie-parser@1.3.5
| +-- cookie@0.1.3
```

- d) At this point we have a runnable Express application. No Oracle JET elements have been added yet; we will do that shortly. Let's try out the application quickly. From directory *part4\jet-on-*

node, execute:

SET DEBUG=jet-on-node:* & npm start

Now access the application at <http://127.0.0.1:3000/>.



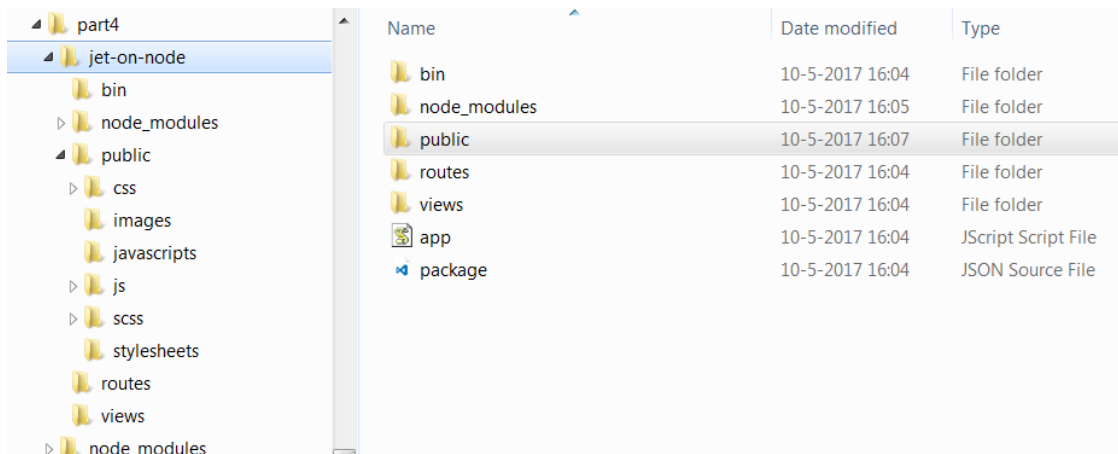
Express

Welcome to Express

- e) We will now add Oracle JET – using the quickstart application. Details, downloads and documentation on JET are available from Oracle Technology Network:

<http://www.oracle.com/technetwork/developer-tools/jet/downloads/index.html>.

For now, you can take a shortcut and download the following zip-file which contains a demo JET application: <https://dl.dropboxusercontent.com/u/12217570/WorkBetter.zip> (15 MB). Extract the contents of this application into directory `part4\jet-on-node\public` (50 MB).



Now access the application –again - at <http://127.0.0.1:3000/>.



- f) Feel free to browse the Oracle JET application a little. The pivotal file – the starting point for the application – is `index.html` in directory `part4\jet-on-node\public`. That file is served to the browser when the request comes in to this application at port 3000.
- g) Our next move is to add a data bound component to the JET application – and make it retrieve data from a REST API that we also need to add. Node is both the server of the rich client web application files as well as the REST API that this application calls to retrieve data.

The steps you need to go through:

- copy or move `hrm.js` from the workshop folder `part4-JET` to `part4\jet-on-node\public\js\viewsModels`
- copy or move `hrm.html` from the workshop folder `part4-JET` to `part4\jet-on-node\public\js\views`
- copy or move `departments.json` from the workshop folder `part4-JET` to `part4\jet-on-node`
- edit file `main.js` in `part4\jet-on-node\public\js` to make it look like `main.js` in workshop folder `part4-JET` (see instructions below)
- edit file `header.js` in `part4\jet-on-node\public\js` to make it look like `header.js` in workshop folder `part4-JET` (see instructions below)
- edit file `app.js` in `part4\jet-on-node` to make it look like `app.js` in workshop folder `part4-JET` (see instructions below)

Name	Date modified	Type	Size
app	11-5-2017 20:53	JScript Script File	2 KB
departments	11-5-2017 21:59	JSON Source File	3 KB
header	11-5-2017 21:04	JScript Script File	4 KB
hrm	28-3-2016 17:12	Firefox HTML Doc...	1 KB
hrm	28-3-2016 17:34	JScript Script File	2 KB
main	11-5-2017 20:59	JScript Script File	6 KB
package	10-5-2017 14:01	JSON Source File	1 KB

In part4\jet-on-node\public\js\main.js, you need to add one small section of code, to add a new HRM tab. Add:

```
'hrm': {label: 'HRM'},
```

in the definition of router.configure:

```
main.js x
69 ojs/ojinputtext
70 ],
71     function (oj, ko, $, utils) {
72         var router = oj.Router.rootInstance;
73         router.configure({
74             'dashboard': {label: 'Dashboard', isDefault: true},
75             'people': {label: 'People'},
76             'hrm': {label: 'HRM'},
77             'organization': {label: 'Organization'},
78             'person': {label: 'Person',
79                 exit: function () {
80                     var childRouter = router.currentState().value;
81                     childRouter.dispose();
82                 },
83                 enter: function () {
84                     var childRouter = router.createChildRouter('emp');
85                     childRouter.defaultStateId = '100';
86                     router.currentState().value = childRouter;
87                 }
88             }
89         });
90     });
91
92     function MainViewModel() {
93         var self = this;
94         self.router = router;
95         utils.readSettings();
```

In file jet-on-node\public\js\viewModels\header.js change the value of property disabled from *true* to *false* for the third element in the appNavData array:

```

header.js
91         "content": '#pageContent'
92     };
93
94     //
95     // Data for application navigation
96     //
97     var router = oj.Router.rootInstance;
98     var appNavData = [
99         {
100             name: router.states[0].label,
101             id: router.states[0].id,
102             disabled: 'false',
103             iconClass: 'demo-navi-dashboard-icon oj-navigationlist-item-icon'
104         },
105         {
106             name: router.states[1].label,
107             id: router.states[1].id,
108             disabled: 'false',
109             iconClass: 'demo-navi-person-icon oj-navigationlist-item-icon'
110         },
111         {
112             name: router.states[2].label,
113             id: router.states[2].id,
114             disabled: 'false',
115             iconClass: 'oj-disabled demo-navi-people-icon oj-navigationlist-item-icon'
116         }
117     ];
118     self.dataSource = new oj.ArrayTableDataSource(appNavData, {idAttribute: 'id'});
119
120     self.toggleAppDrawer = function ()

```

Finally, in app.js – the Node.js application itself – we need to add support for the REST API, by adding these two lines, just under `var app = express();` :

```

var departments = JSON.parse(require('fs').readFileSync('departments.json', 'utf8'));
app.get('/departments', function (req, res) { //process

```

```

    res.send( departments); //using send to stringify and set content-type

});

```

```

var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var routes = require('./routes/index');
var users = require('./routes/users');

var app = express();

var departments = JSON.parse(require('fs').readFileSync('departments.json', 'utf8'));
app.get('/departments', function (req, res) { //process
    res.send( departments); //using send to stringify and set content-type
});

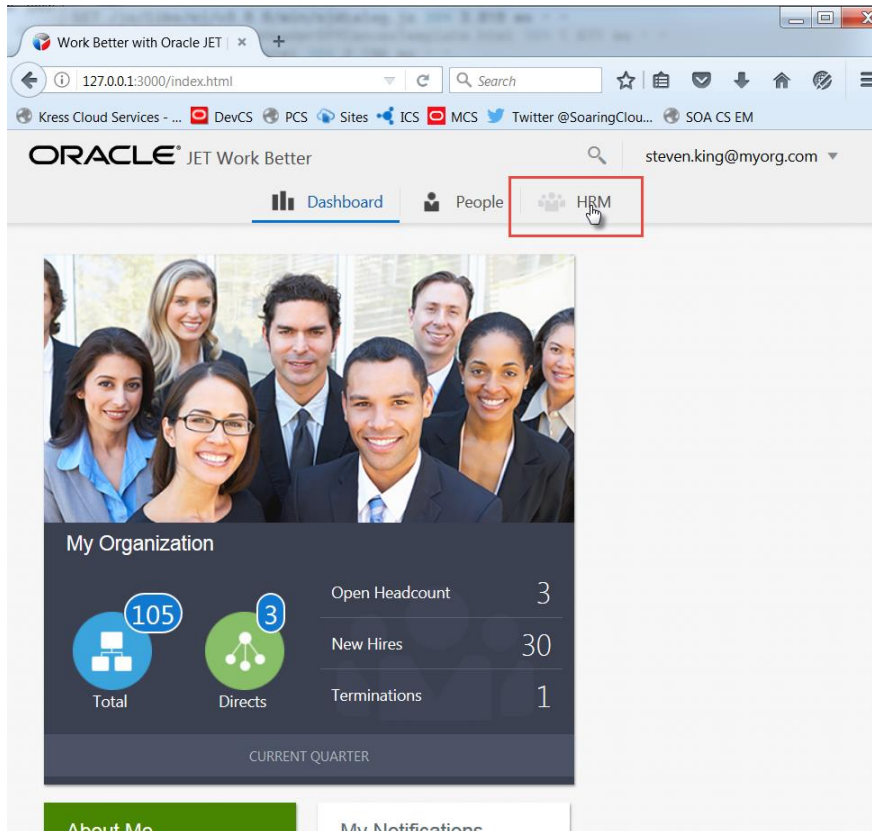
// view engine setup
app.set('views', path.join(__dirname, 'views'));

```

h) Restart the node application – again with

`SET DEBUG=jet-on-node:* & npm start`

and reload the application in the browser. The HRM tab has been added:



Navigate to the HRM tab. You will now see data in the table component. This is the data loaded from the `departments.json` file in the root folder of the application. You can verify this by changing the data in the file – and restarting the application (the data is cached when the application starts).

ORACLE[®] JET Work Better

🔍

steven.king@myorg.com ▼

Dashboard

People

HRM

HRM Content

The HRM Details

Department Id	Department Name	Location
5	Fontys Hogeschool	Zoetermeer
10	Administration	Zoetermeer
20	Marketing	Zoetermeer
30	Purchasing	Zoetermeer
40	Human Capital	Zoetermeer
50	Shipping	Zoetermeer
60	IT	Zoetermeer
70	Public Relations	Zoetermeer
80	Sales	Zoetermeer
90	Executive	Zoetermeer
100	Finance	Zoetermeer

6. Complex REST API for retrieving rich Artist information

The REST API that we exposed in the previous section was quite straightforward. Information retrieved from a static local file, doing a little manipulation. It was a fine start. In this section, we make it a little bit more interesting. We will work on an API that returns a JSON document in return to an HTTP GET request that specifies the name of a particular artist. This JSON document contains details on the artist – such as a genre label, a biography, a list of the most recently released albums with their details and more. To gather this information, the Node application that exposes this API has to go out and fetch data from external services such as the Spotify API (<https://api.spotify.com/v1>).

The situation can be described like this:



We will build up this API in a number of steps. You will find the resources in folder `part5-artist-api`.

- Open the command line window in folder `part5-artist-api`. Before the application can be started, you need to use npm to install a few packages: `express`, `request`, `body-parser` and `async`.

```
npm install express request body-parser async
```

- Run the `artist-enricher-api-1.js` program. It listens to HTTP requests of the form <http://127.0.0.1:5100/artists/get?artist=artistName> (such as <http://127.0.0.1:5100/artists/get?artist=b52s> or <http://127.0.0.1:5100/artists/get?artist=u2>) and similar calls of the form `127.0.0.1:5100/artists/artistName` such as `127.0.0.1:5100/artists/u2`. Try out such a call from your browser or using CURL.
- Inspect the code in `artist-enricher-api-1.js`. The familiar Express style configuration of a web server to handle GET requests can be found, for three different URL paths. The configuration of a callback function for the URL path `/artists/*` that hands off the work to function `handleArtists()` while passing the value of the `artist` query parameter and a similar one for `/artists?artist=*` and the root path `/`.

Also check the function `handleArtists()` that gathers data from the external API and then calls the function `composeArtistResponse ()` to return the response. Check how the Spotify API is invoked using the `request()` function and how the response is processed.

- d) Stop `artist-enricher-api-1.js` and run `artist-enricher-api-2.js`. Make the same HTTP GET call as before. You will now see – after a somewhat longer waiting time - additional information in the response: details for the albums released by our artist. Spotify offers an API that provides a list of albums – maximum of 50 per call. For each album, we get the name and an image URL for the cover image. We do not get a release date.

Inspect the code. A small section is added to invoke Spotify a second time, to fetch details about albums. One thing you could notice is that the code slightly becomes harder to read – additional indentation for each consecutive outbound call and callback function to handle the response. The call to fetch album information uses information from the first call, so in this case it is justified that the two calls take place sequentially instead of in parallel. However, sometimes multiple callouts can be done at the same time because they are mutually independent. Waiting for the two response at the same time makes the overall wait shorter.

The handling of asynchronous callbacks and the coordination of parallel calls can be quite complex. We will next look at package `async` that helps bringing some order to that chaos.

- e) Package `node-async` (<https://github.com/caolan/async> and installable with `npm install async`) is a utility module which provides straight-forward, powerful functions for working with asynchronous JavaScript. It can be used in `node.js` applications as well as in client side JavaScript running in a web browser. It makes it quite easy for example to perform multiple asynchronous activities in parallel and work with the combined result from all these activities.

We will make use of `async` to orchestrate the two outbound REST API calls we make to Spotify. Notice how `async.waterfall` is used to organize the sequential calls to the Spotify API. The functionality is not spectacular – control of the program flow and readability of the code increases notably. Note how `callback()` is used to indicate the completion of *unit* in `async waterfall`. Also note how `callback(null, artist.spotifyId);` is used to specify that no error occurred (the first parameter is null) and also to pass a result from this unit to the next: `artist.spotifyId` is passed in as the first input argument to the next unit in the waterfall. (through the shared context variable `artist` this same value is accessible using `artist.spotifyId`).

Run `artist-enricher-api-3.js` and make one or more API calls from your browser, CURL, Postman, SoapUI or whatever your favorite tool is.

- f) Spotify offers yet another API that allows us to retrieve details for albums – primarily the release date. In `artist-enricher-api-4.js`, you will find the waterfall extended with a third unit. This third unit calls the albums details API on Spotify. This can be done for up to 15 albums at one time. Because the second call to Spotify in the waterfall results in a list of up to 50 albums, we need multiple calls to the album details API. Of course these calls should not be made sequentially. In the source code for `artist-enricher-api-4.js` you will see how a special async mechanism is used: `async.forEachOf`. The `forEachOf` function is handed an array – in this case an array of album arrays. For each element in the array, `forEachOf` will execute the function. When all function calls for the elements in the array have signaled their completion – through the `callback()` call – the completion function in `forEachOf` is executed to do any final processing of the joint results. In this example, all that needs to be done is another call to `callback()` to tell `async.waterfall` that this unit is complete.

Run `artist-enricher-api-4.js`, make one or more calls and verify that the release date is now added for each album. You can try to vary the size of the chunk – the number of albums in one array – to see if that impacts the processing time. The smaller the chunk, the larger the number of [parallel] calls to Spotify.

7. Miscellaneous and Advanced Topics

In this section, we will look at a number topics, some which advanced. These include:

- sending emails from node.js applications
- prompting command line input from users
- working with Server Sent Events to push data from the Node.js server application to a browser client application
- use of new language features: Promises and Generators

Check the scripts in folder part6-miscellaneous.

Sending Emails

Sending emails is one of many operations that are easily performed from Node using one of many modules available for Node. File `emailer.js` creates a (custom) module – leveraging the `emailjs` module for sending emails. In `mailclient.js`, this custom module is used to send an email.

- a) Run `npm install` in folder `part6-miscellaneous\email`, to install the `emailjs` node module.
- b) Open file `emailer.js`. See how the module `emailerAPI` is exposed from this file. Modify the file with your settings for email account credentials and mail server.
- c) Open file `mailclient.js`. See how the (local, custom) module `emailerAPI` is required. Update the subject, body and addressees – to send your own personalized email message to your own addressee of choice. Then run the `mailclient`:

```
node mailclient
```

Check if the email was sent (and received) successfully.

Prompting input from users

Node is frequently used as backend for web applications. Any input required is gathered through the web interface in the browser. However, Node applications will frequently run as server side programs without this browser based outlet. In this case too they can get user input, for example just on the command line. Node module `prompt` (<https://www.npmjs.com/package/prompt>) can be used for this, as is seen in the code sample in directory `part6-miscellaneous\prompting`.

- a) Open command line in directory `part6-miscellaneous\prompting`
- b) Run `npm install` to download node module `prompt`
- c) Run sample program:

```
node index.js
```


Provide the requested input and see how it is processed.

Type *exit* to end the dialog.

- d) Inspect the – very simple – code in index.js. If you want to experiment with various settings and options, such as validations or colors in the command line dialog, feel free to do so – using the documentation at <https://www.npmjs.com/package/prompt>.

Server Sent Events to Push Messages to Web Clients

Node can push messages to browser clients, using WebSockets and the simpler Server Sent Events (SSE) mechanism. We will use the latter in this next lab.

- a) Open command line in directory part6-miscellaneous\server-push.
- b) Run `npm install` to get hold of required node modules (body-parser, serve-static, prompt and express).
- c) Inspect file app.js. This file serves a static file – index.html – and exposes a single endpoint - /updates – where SSE clients can register.

File index.html contains a JavaScript fragment, where the client registers with the SSE Server – at the /updates endpoint. A listener is associated with this SSE connection – to read the consumed message and update the UI subsequently, in the plainest way imaginable.

Back to app.js: it uses the prompt module to solicit input from the user on the command line. Every piece of input is published as server sent event to all SSE clients, to be published in the browser UI.

You can look at file sse.js for some more details on how the SSE clients and topics internals are implemented. You will notice that there is not too much to it.

- d) Run app.js on the command line:

```
node app.js
```

And open your browser at `http://127.0.0.1:3000/`.

Type some input on the command line. This input should appear in the browser.

Open a second browser window at the same URL. Type some more input on the command line.

This input should appear in all browsers.

Callbacks and Promises

Node (and JavaScript in general) makes heavy use of asynchronous mechanisms. Many operations are performed asynchronously with regard to the invoker and will inform the invoker about the result of the action through a callback. Many calls to functions in Node programs will carry a reference to a callback function – the function that should be invoked to asynchronously return the result of the operation.

- a) Navigate to directory part6-miscellaneous\promises. Open the file app.js. Here is an example of an asynchronous execution. Function doStuff is called – it will do its thing (including a 1.5 second wait) and then callback to report its outcome. The call to doStuff has to provide a callback function in order to be notified of the result from doStuff.

Run app.js: `node app.js`

You will see that the last line in the program – is executed prior to the execution of the callback function. This is not uncommon in Node programs: what appears to be the last line is in fact not the last action taken. Note how the `setTimeout()` built in timer also takes a callback function: the function to be executed when the time is up.

- b) Open app2.js. Timing is added, to check on the total execution time. Run app2.js.
- c) In app3.js, the variable `numberOfLoops` dictates how many times and for which values the `doStuff()` function gets executed. In a sequential for loop, each call to `doStuff()` is made. And here we see one of the powers of asynchronous operations: the calls to `doStuff()` are made sequentially but since we do not wait for `doStuff()` to complete its work before making the next iteration in the loop, the calls are virtually parallel. Instead of waiting five subsequent times for a call to `doStuff()` to be completed – which would take 5 times 1.5 seconds or a little over 7.5 seconds – we have to wait less than half that time. Or do we?

Run app3.js.

Check if the results are reported on the console in the expected order – step 0..step5. If they are not, how can that be?

- d) In app4.js an array called `results` is introduced to try to capture the results from the calls to `doStuff()`. Check out the logic in app4 and consider whether this will do the job.

Next, run app4.js and verify whether our objective – end up with an array with all results

produced by `doStuff()` collected. We will get back to this.

- e) It is quite common to take the result from a call to a function and use that result in a subsequent call to another function – and use the result of that call to invoke yet another function. With asynchronous functions (or functions that return their result in a callback) that is not so trivial as in a purely synchronous programming model is the case. Take a look at `app5.js`. Here, the result from `doStuff()` is passed to `doAdditionalStuff()` and that outcome to `doMoreStuff()`. Each of these functions returns its result asynchronously. Therefore, we end up with a number of nested callback functions.

Run `app5.js` to see what it produces.

Having to work with nested callback functions is one of the consequences of the JavaScript programming model. And it can be the cause of programs of which the logic is hard to track and that are therefore difficult to create and certainly to maintain. Organizing the work properly – ensuring that everything happens in parallel when that is possible and is orchestrated in the proper sequence when that is required – is not a simple thing.

We will see two ways in which this “callback hell” as it is sometimes referred to can be addressed.

- f) On the command line, enter

```
npm install
```

to have node module `async` (for documentation check out <http://caolan.github.io/async/>) installed. This module can help bring simplicity to the asynchronous programming model.

Open `app6.js`. Compare `app6.js` to `app4.js`. The two are very similar. Note that the for loop that drove the calls to `doStuff()` in `app5` and previous versions of the application is no longer there. Instead, the `async.forEachOf` operation from the `async` module organizes the parallel calls, driven by the `steps` array. This array is initialized with *numberOfLoops* elements that are set using the `fill` method and subsequently handed to the `map()` method (see https://www.w3schools.com/jsref/jsref_map.asp). In the `map()` method, the value for each element in the array is mapped to the desired value which is in this case just the index of the array element.

When all calls have returned their asynchronous result, the function passed to `async.forEachOf` as second parameter is invoked to wrap up the whole affair.

Now run `app6.js`.

The results are gathered into the result array correctly this time. The evaluation of the value of *i* – the index variable into the results array – is done in the proper context and at the right time. Note that the total execution time is now written to the console only once.

- g) Take a look at app11.js. This program shows the usage of the waterfall operation in `async` – used to chain together asynchronous operations that should be executed subsequently and that may take input from their predecessor.

Run app11.js. Check the final outcome. Try to determine how that outcome was arrived at.

- h) In app7.js, the nested callbacks from app5.js have been reimplemented using the waterfall mechanism in `async`. Instead of nested callback, in the waterfall construct we specify a number of subsequent asynchronous calls to make. Each step concludes with a call to the waterfall callback function, passing the outcome of the step and in doing so making that outcome available for the next step in the waterfall.

Run app7.js.

Try to understand the flow through the program. Verify whether the result produced by three subsequent asynchronous function calls taking place in parallel for multiple values in the original *steps* array is gathered correctly in the *results* array. Hopefully through the use of `async`, the nested callbacks are no longer quite as bad as before.

- i) Add logic in app7.js: a fourth function should be created – along the lines of `doStuff()` and companions – to add the string *'For starters, then '* at the beginning of the parameter passed in. This function should be called last in the `async.waterfall` sequence. The result for Step 0 should then become: *"For starters, then STEP 0 Enriched! (The finishing touch)"*.
- j) One of the recent evolutions in JavaScript (ES6 and beyond) has been the introduction of the Promise. See <https://strongloop.com/strongblog/promises-in-node-js-an-alternative-to-callbacks/> for background. Using Promises, even more than with module `async`, asynchronous calls can be dealt with in a way that almost looks and feels synchronous or at least sequential.

In app9.js, Promises have been used to implement the nested series of asynchronous calls that are needed for Step 1, similar to `async.waterfall()` in app7.js. Run app9.js, verify the outcome, and try to understand the logic.

- k) Next, in app10.js, the logic from app7.js has been reimplemented using Promises (`Promise.all` instead of `async.forEachOf`). The code is compact. Try to understand its logic.

Run app10.js. Verify the results produced.

- l) Just as before, extend app10.js: a fourth function should be created – along the lines of doStuff() and companions – to add the string *'For starters, then '* at the beginning of the parameter passed in. This function should be called last – this time in the promise chain. The result for Step 0 should then become: *"For starters, then STEP 0 Enriched! (The finishing touch)"*.
- m) Program app12.js is the Promise based counterpart to program app11.js. See how much more compact the programming is using Promises than with async – which is already way better than nested callbacks!

Run app12.js and verify that it produces the required result.

8. Interacting with MongoDB

MongoDB is a NoSQL database that stores JSON documents and is operated largely through JavaScript. MongoDB is part of the so called MEAN stack – along with Node, Express and the rich client web application framework Angular. We have seen two parts of the stack already – Express and Node – and will now take a look at how from Node.js we can interact with the third – MongoDB.

We will work with the scripts in folder `part7-mongodb`.

Note: you can either work with a local installation of MongoDB directly on your operating system or a MongoDB instance running in a Docker Container or a Virtual Machine. Through mLab (<https://mlab.com>) or other providers, you can also work with a MongoDB instance in the cloud.

Local Installation of MongoDB

Go to <https://docs.mongodb.com/manual/administration/install-community/> to choose between three links for installing MongoDB on Linux, MacOS or Windows respectively.

Pick the link for your platform. It will take you to a step by step document, for the MongoDB Community Edition. The required software is available from the download center: <https://www.mongodb.com/download-center#community>. The download size is roughly 150 MB. The installation process will take less than 5 minutes.

For today's workshop: ideally you have your own MongoDB instance to work against. However, alternatively you can use these connection details to run against a shared cloud based MongoDB instance:

```
var mongodbHost = 'ds139791.mlab.com';  
  
var mongodbPort = '39791';
```

The credentials to use for this shared database are `mongouser/mongopass`.

- a) Open the command line for folder `part7-mongodb`. Run `npm install` to download the `mongodb` node module.
- b) These practices assume that the `countries` data set has been loaded into a MongoDB database called *world*. This can easily be done with the following steps (note: these steps have already been performed for the shared cloud based instance of MongoDB):
 - open a command line or terminal window in directory *country-queries*.
 - import the file `countries.csv` into collection *countries* in database *world* using the following command (assuming that the `bin` directory of the MongoDB installation is in the `PATH` environment variable):

```
mongoimport --host 127.0.0.1:27017 --db world --collection countries  
--drop --file countries.csv --type csv --fieldFile
```

countriesFields.txt

- c) Open file `connect.js` in directory `MongoDB-NodeJS-Client` in your favorite text editor. Verify and change if required the values of the variables `mongodbHost`, `mongodbPort` and `mongodbDatabase`. Use boolean variable `cloud` to indicate whether you want to use the shared cloud instance.

Then run this NodeJS module, using:
`node connect`

- d) Open the file `query6.js`. This file uses Promises instead of asynchronous callbacks. If you prefer the callbacks or have an older version of NodeJS that does not support the Promises, then use file `query.js`.

The code performs the following steps – check how these steps are implemented in NodeJS, using the NodeJS driver for MongoDB

- * connect
- * fetch the top 20 largest countries (by area) into an array of country objects (and print two randomly selected countries from that array)
- * retrieve a cursor that returns all countries in Asia; get the number of records from the cursor; iterate through the cursor and write the name of each country returned from the cursor
- * retrieve through the aggregation framework using an aggregation query object a cursor that returns the largest country for each continent and write out the full country object

Note how well NodeJS can work with the JSON documents returned from MongoDB: these documents are native data objects in JavaScript and therefore in the NodeJS application.

- e) Open the file `crud6.js`. This file uses Promises instead of asynchronous callbacks. If you prefer the callbacks or have an older version of NodeJS that does not support the Promises, then use file `crud.js`.

The code performs the following steps – check how these steps are implemented in NodeJS:

- * connect
- * create a new collection (or work with an existing one) and add a list of two documents to it in a single statement (`insertMany`)
- * verify the creation of the documents by retrieving the newly created documents using a cursor
- * update one of the records, using a find filters and by applying a complex `$set` definition – with nested attributes and nested collections
- * verify the updated document
- * delete all documents
- * drop the collection

Please feel free to try out some more interaction between NodeJS and MongoDB. The documentation for the driver can be found here: <http://mongodb.github.io/node-mongodb-native/2.0/> .

9. Dockerizing Node Applications

Running Node applications in Docker containers is extremely simple to accomplish. It is a very common approach for implementing REST APIs, web serving applications and especially microservices.

If you are interested in trying this out – take a look at the tutorial on Dockerizing a Web Application at <https://nodejs.org/en/docs/guides/nodejs-docker-webapp/> . If you follow the steps described here, you will have a Dockerized Node application in no time.

10. Node OracleDB Database Driver

Node applications may want to have access to databases (relational and NoSQL) – similar to access to the file system, external APIs over HTTP and other resources. Just like Java applications use a JDBC driver to access a relational database, Node.js applications can leverage a database driver for the database they want to access. There is no standard for database drivers or for interacting with a relational database. There is no JDBC-like API. Working with different databases is done in different ways.

Oracle has created an open source project – available on [GitHub](#) – called node-oracledb. This project provides a Node.js database driver to the Oracle Database. It leverages Oracle Database client libraries – the thick OCI API – that need to be installed on the server running the node application. Through node-oracledb, the interaction from a node application with an Oracle Database becomes fairly easy and straightforward – similar to JDBC but simpler. And asynchronous of course – using callbacks to handle the responses returned from the database. Performing queries, DDL and DML as well as PL/SQL calls is all supported pretty well as are native Oracle data types.

One way to trying out this Oracle Database driver for Node.js applications is using the Prebuilt VirtualBox image available from OTN for trying out database development.

Another option is to build a Virtual Machine that contains Oracle 11gR2 XE, Node 7.x and CentOS 7.2 using the OXAR GitHub Repository (<https://github.com/OraOpenSource/OXAR>) in combination with Vagrant and VirtualBox.

Rolling a Virtual Box image using OraOpenSource OXAR

The steps for creating your own VM image are as follows:

1. make sure that you have Vagrant and VirtualBox installed locally
2. get the OXAR repository content

git clone <https://github.com/OraOpenSource/OXAR>

3. copy oracle-xe-11.2.0-1.0.x86_64.rpm.zip (the Linux installer for Oracle 11gR2 XE Database downloaded from OTN) to the OXAR/files directory
4. edit the file config.properties in the OXAR root directory – set parameter OOS_ORACLE_FILE_URL to file:///vagrant/files/oracle-xe-11.2.0-1.0.x86_64.rpm.zip and save the change:

```
OOS_ORACLE_FILE_URL=file:///vagrant/files/oracle-xe-11.2.0-1.0.x86_64.rpm.zip
```

5. run vagrant using the statement:

```
vagrant up
```

this will run for a file, download the CentOS base image and create the VM, install all of Git client, Oracle 11gR2 XE Database, Node and Node OracleDB Driver

6. after rebooting the system, the VM will be started (or you can start it using vagrant up again or by using the VirtualBox manager).

You can start an SSH session into it by connecting to localhost:50022, then login to Linux using vagrant/vagrant

7. connect to the database using sqlplus hr/oracle
8. download the workshop resources using

git clone <https://github.com/lucasjellema/nodejs-introduction-workshop-may2017>

9. navigate to the directory nodejs-introduction-workshop-may2017\part9-oracledb. Edit file dbconfig.js: set the connectString property to an empty string. Save the changes.

10. Now run the file select.js using:

```
node select
```

it will do a simple select from table HR.DEPARTMENTS.

11. Run select_set.js using

```
node select_set.js
```

Using the Pre Built OraOpenSource OXAR VM image

The VM created in the previous steps is also available as an exported appliance. Copy the file OXAR-XE11gR2-Node.ova locally and import this file as an the appliance into VirtualBox.

The next steps are

1. Start the VM after importing it
2. Start an SSH session into it by connecting to localhost:50022, then login to Linux using vagrant/vagrant
3. connect to the database using sqlplus hr/oracle
4. navigate to the directory nodejs-introduction-workshop-may2017\part9-oracledb. Edit file dbconfig.js: set the connectString property to an empty string. Save the changes.
5. Now run the file select.js using:

node select

it will do a simple select from table HR.DEPARTMENTS.

6. Run select_set.js using

node select_set.js

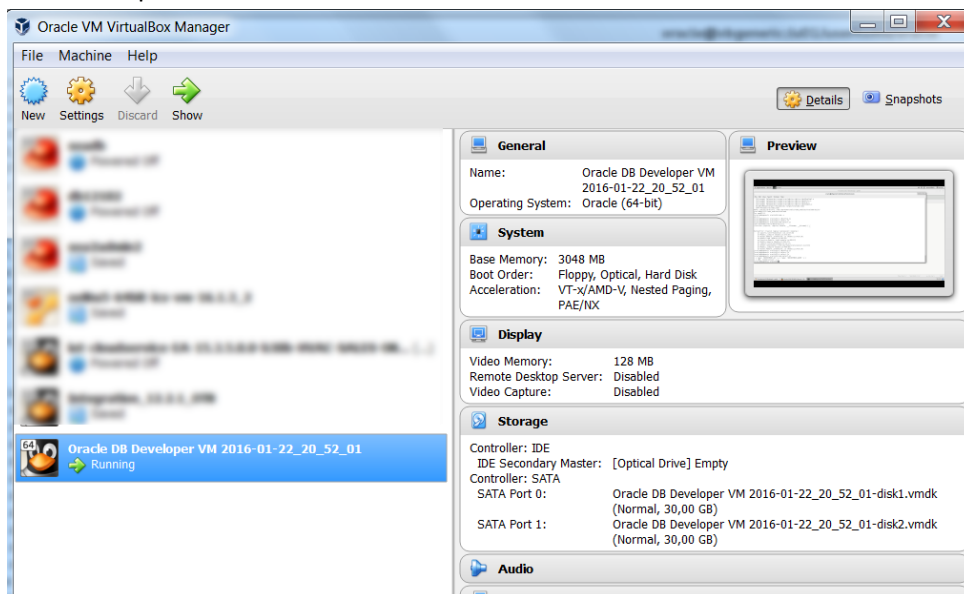
Using the Pre Built Virtual Box Developer VM Image

The steps for using the are as follows:

- a) Download the Virtual Box image from <http://www.oracle.com/technetwork/database/enterprise-edition/databaseappdev-vm-161299.html> (8 GB).
- b) Import the downloaded file – for example DeveloperDaysVM2016-01-22_23.ova – into VirtualBox.

When the import is complete, you can increase the allocated memory – to 4GB for example.

- c) Run the imported machine.



Some details for the VM:

Oracle SID : orcl12c

Pluggable DB : orcl

ALL PASSWORDS ARE : oracle

ORACLE_HOME (should be set to): /u01/app/oracle/product/12.2.0.1/db_1

d) Open a Terminal.

start a sudo session:

sudo -s

(sudo password is *oracle*)

```
root@vbgeneric:/u01/userhome/oracle
File Edit View Search Terminal Help
[oracle@vbgeneric oracle]$ sudo -s
[sudo] password for oracle:
[root@vbgeneric oracle]#
```

And perform these installation steps described here (also see <https://tecadmin.net/install-latest-nodejs-and-npm-on-centos> , or use these steps for upgrading: <http://www.hostingadvice.com/how-to/update-node-js-latest-version/>):

yum -y install nodejs

```
[root@vbgeneric oracle]# yum -y install nodejs
Loaded plugins: langpacks
nodesource/x86_64/primary_db | 30 kB 00:01
Resolving Dependencies
--> Running transaction check
--> Package nodejs.x86_64 0:4.4.1-1nodesource.el7.centos will be installed
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package Arch Version Repository Size
=====
Installing:
nodejs x86_64 4.4.1-1nodesource.el7.centos nodesource 8.5 M
Transaction Summary
=====
Install 1 Package

Total download size: 8.5 M
Installed size: 29 M
Downloading packages:
nodejs-4.4.1-1nodesource.e 97% [=====] 739 kB/s | 8.3 MB 00:00 ETA
```

yum -y install git

Exit the sudo session – and resume as user oracle.

Check if ORACLE_HOME is set (echo \$ORACLE_HOME). If it is not set, set it:

export ORACLE_HOME=/u01/app/oracle/product/12.2.0.1/db_1

Now it is time to install node-oracledb. Run:

```
npm install oracledb
```

This will do the installation of the Node Oracle Database Driver.

```
[oracle@vbgeneric oracle]$ echo $ORACLE_HOME
/u01/app/oracle/product/12.1.0.2/db_1
[oracle@vbgeneric oracle]$ npm install oracledb
/
> oracledb@1.8.0 install /u01/userhome/oracle/node_modules/oracledb
> node-gyp rebuild

make: Entering directory `/u01/userhome/oracle/node_modules/oracledb/build'
CXX(target) Release/obj.target/oracledb/src/njs/src/njsOracle.o
CXX(target) Release/obj.target/oracledb/src/njs/src/njsPool.o
CXX(target) Release/obj.target/oracledb/src/njs/src/njsConnection.o
CXX(target) Release/obj.target/oracledb/src/njs/src/njsResultSet.o
CXX(target) Release/obj.target/oracledb/src/njs/src/njsMessages.o
CXX(target) Release/obj.target/oracledb/src/njs/src/njsIntLob.o
CXX(target) Release/obj.target/oracledb/src/dpi/src/dpiEnv.o
CXX(target) Release/obj.target/oracledb/src/dpi/src/dpiEnvImpl.o
CXX(target) Release/obj.target/oracledb/src/dpi/src/dpiException.o
CXX(target) Release/obj.target/oracledb/src/dpi/src/dpiExceptionImpl.o
CXX(target) Release/obj.target/oracledb/src/dpi/src/dpiConnImpl.o
CXX(target) Release/obj.target/oracledb/src/dpi/src/dpiDateTimeArrayImpl.o
CXX(target) Release/obj.target/oracledb/src/dpi/src/dpiPoolImpl.o
CXX(target) Release/obj.target/oracledb/src/dpi/src/dpiStmtImpl.o
CXX(target) Release/obj.target/oracledb/src/dpi/src/dpiUtils.o
CXX(target) Release/obj.target/oracledb/src/dpi/src/dpiLob.o
CXX(target) Release/obj.target/oracledb/src/dpi/src/dpiCommon.o
SOLINK_MODULE(target) Release/obj.target/oracledb.node
COPY Release/oracledb.node
make: Leaving directory `/u01/userhome/oracle/node_modules/oracledb/build'
oracledb@1.8.0 node_modules/oracledb
└─ nan@2.2.1
```

e) To get hold of the workshop resources inside the VM:

```
git clone https://github.com/lucasjellema/nodejs-introduction-workshop-may2017
```

Trying out the Node OracleDB Driver

a) Navigate to /nodejs-introduction-workshop-may2017/part9-oracledb. Run the node program select.js:

```
node select.js
```

This will perform a select operation against the HR schema in the local database. You should see a department record being reported on.

```
[ { name: 'DEPARTMENT_ID' }, { name: 'DEPARTMENT_NAME' } ]
[ [ 180, 'Construction' ] ]
```

Now check the contents of the select.js program. It shows you quite a bit about how the Oracle Database Driver for Node has you interact with the Oracle database. The connection details are configured in the file dbconfig.js that is *required* into select.js. This file is currently configured to connect to the HR schema in the local database.

Note how the dbconfig-remote.js configures connection details for a DBaaS instance running in the cloud. You can change the require statement in select.js to refer to this second set of configuration details.

b) Run select_set.js using

node select_set.js

```
[oracle@vbgeneric part6-oracledb]$ node select_set.js
{ rows: undefined,
  resultSet:
    ResultSet {
      metaData: [ [Object], [Object] ],
      close: [Function: close],
      getRow: [Function: getRow],
      getRows: [Function: getRows] },
  outBinds: undefined,
  rowsAffected: undefined,
  metaData: [ { name: 'EMPLOYEE_ID' }, { name: 'LAST_NAME' } ] }
fetchOneRowFromRS(): row 1
[ 100, 'King' ]
fetchOneRowFromRS(): row 2
[ 101, 'Kochhar' ]
fetchOneRowFromRS(): row 3
[ 102, 'De Haan' ]
fetchOneRowFromRS(): row 4
[ 103, 'Hunold' ]
fetchOneRowFromRS(): row 5
[ 104, 'Ernst' ]
fetchOneRowFromRS(): row 6
[ 105, 'Austin' ]
fetchOneRowFromRS(): row 7
[ 106, 'Pataballa' ]
fetchOneRowFromRS(): row 8
[ 107, 'Lorentz' ]
fetchOneRowFromRS(): row 9
[ 108, 'Greenberg' ]
fetchOneRowFromRS(): row 10
[ 109, 'Faviet' ]
```

This is an example for retrieving a set of records.

Note: also see examples of using the node oracle database driver at : <https://github.com/oracle/node-oracledb/tree/master/examples>

End to End Workshop

To bring it all together – the key aspects of Node applications including the ability to handle HTTP Requests, call out to external APIs and local databases and perform all tasks any modern general purpose programming language can perform – it would be a nice challenge to implement the following application:

A simple web application running in the browser shows a list of artists. The web application calls a REST API for this list; the list contents is retrieved from a text file. End users can click on the name of the artist and are then shown details about the artist – such as the albums released by the artist and an image of the artist. This information is fetched by the web application from its own backend. The backend retrieves the data from the Spotify API.

Users can like artists and optionally add comments about the artists. The number of likes per artist as well as the comments are sent from the browser to the backend and then stored in a MongoDB (or Oracle) Database. The number of likes is presented in the browser UI. The comments for an artist can be displayed as well.

Whenever a new *like* is received by the backend, all browser clients are notified through a Server Sent Event. The user will see a notification about the most recent *like*. The number of likes for the artist is updated in the UI.

Resources

Official documentation for Node.js: <https://nodejs.org/en/docs/>

Several useful how-to articles on Node.js: <https://docs.nodejitsu.com/>

Article that explains how to take a running Express web application and Dockerize it:
<https://nodejs.org/en/docs/guides/nodejs-docker-webapp/> .

GitHub Repository for Oracle JET : <https://github.com/oracle/oraclejet>

GitHub Repository for Node OracleDB (database driver): <https://github.com/oracle/node-oracledb>

Quick Start with MongoDB: <https://docs.mongodb.com/manual/installation/>

