# AMIS SIG Introduction Node.js–Hands-on

31st March 2016

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.

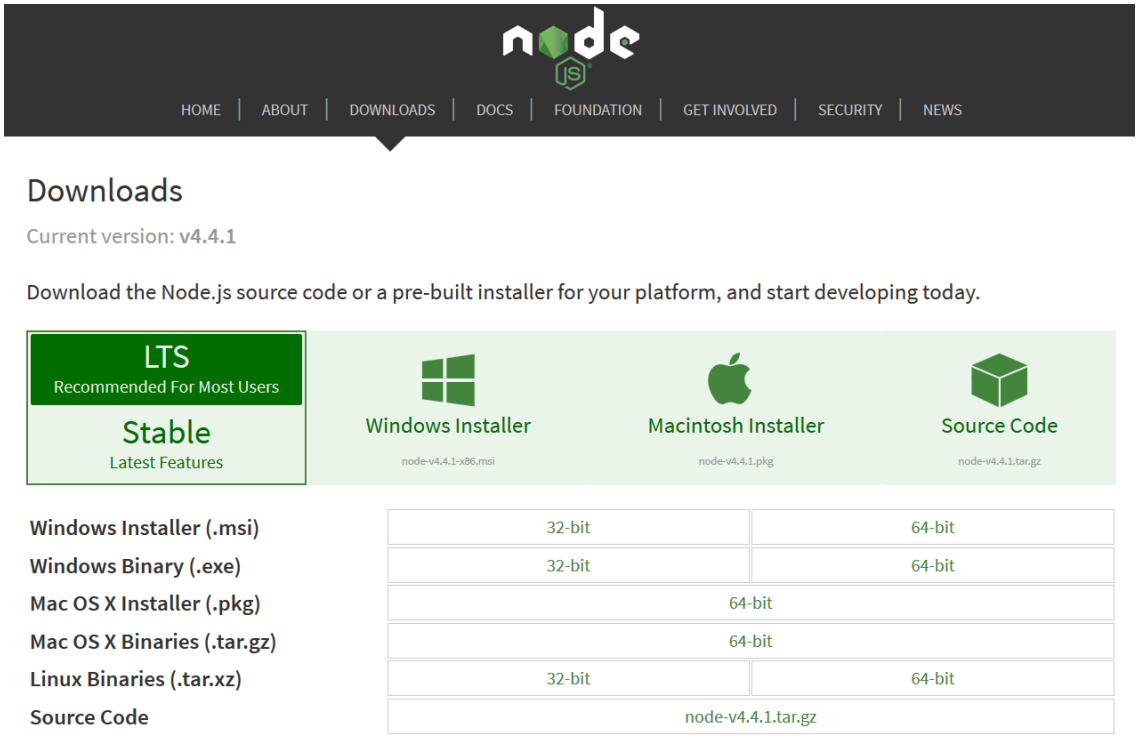
One example of a rich client web application that can be served from Node.js is the Oracle JET application that you will work on a little. Note that similar applications, for example based on Angular2 or other frameworks, can run from Node.js just as easily.

Somewhat more advanced topics are suggested as bonus practices, looking at 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 and Echonest). We also discuss the Node Oracle DB Database Driver for Node.js; 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: https://github.com/lucasjellema/sig-nodejs-amis-2016 .

# 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.




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:

```
C:\>node -v
v4.4.1
```

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 options in this workshop.

## 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) 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.

c) 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.

d) 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 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.

e) 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-v4.x/docs/api/util.html .

f) 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.

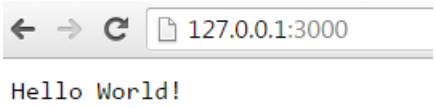
The sources discussed in this section are in directory part2-http.

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\sig-node-js-31march2016\sig-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

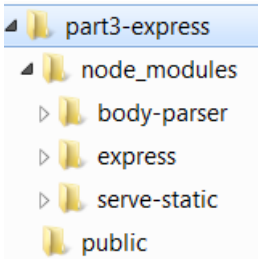
Many Node.js application solve a similar challenge: handling HTTP requests. This happens for example for static file serving, rich client web applications and for the implementation of REST APIs. Although the core capabilities in Node.js for dealing with HTTP interactions, with the Express framework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

   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.




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.

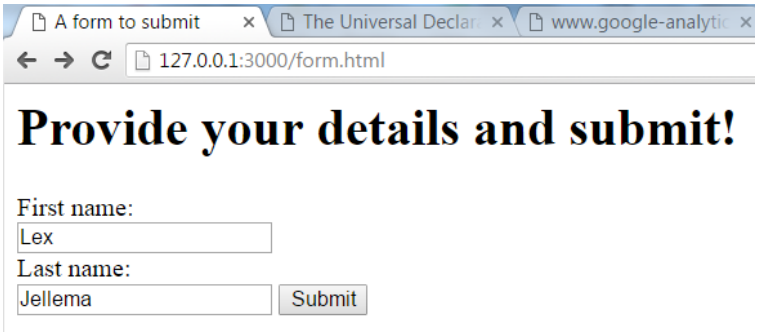
   Note how the browser shows an image – also fetched from the Node.js application. Click on the download link for the PDF document. This document too is returned by the Node.js application.

   Where do these resources come from?

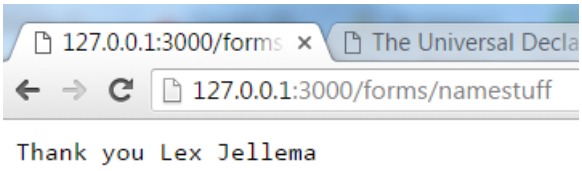
d) Inspect the sources for express-4.js. Here we handle a form submission. This source more or less belongs together with the HTML source form.html in directory public. Check out that file. try to understand how the form and the node application fit together.

Now run express4.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.

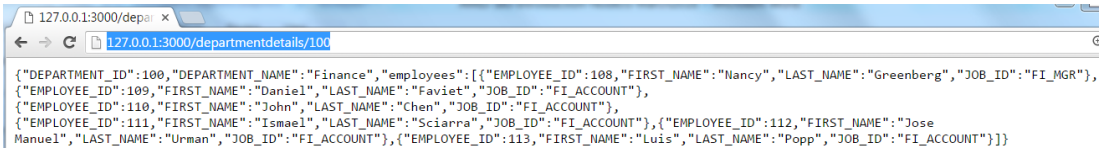
List all departments using: http://127.0.0.1:3000/departments.

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(Java Script record)) in the *end* event handler

## 5. Run Oracle JET application of 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

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

   express jet-on-node

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

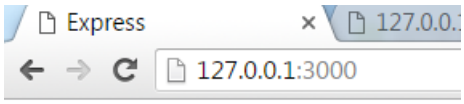
   npm install

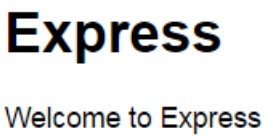
   This installs a number of node modules. You can check directory part4\jet-on-node\node-modules to get an impression.

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/.

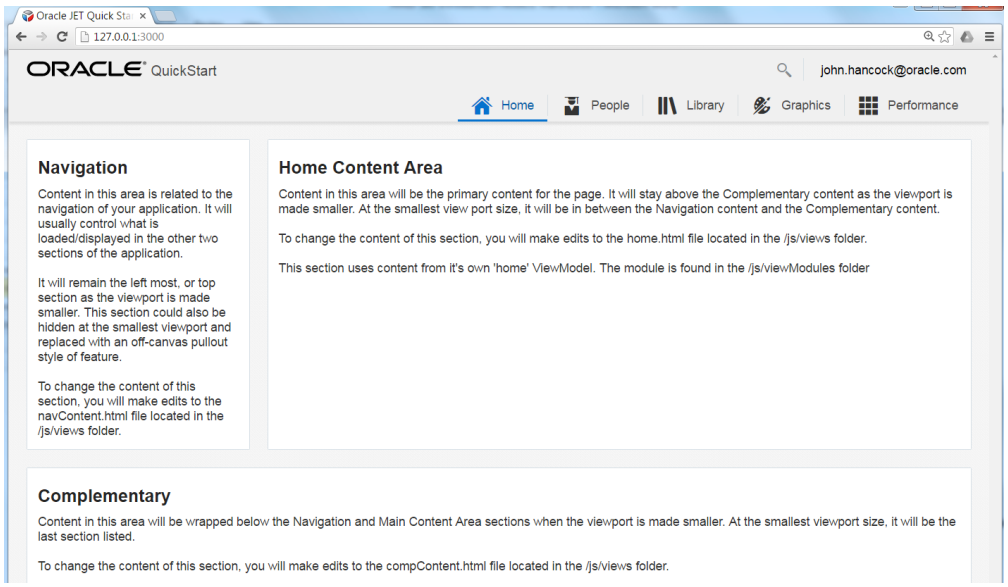



e) We will now add Oracle JET – using the quickstart application. Go to http://www.oracle.com/technetwork/developer-tools/jet/downloads/index.html and download Quickstart: Basic Starter Template with Oracle JavaScript Extension Toolkit Pre-configured (http://download.oracle.com/otn/JET/20/OracleJET_QuickStartBasic.zip).

Extract all files from this zip-file to directory part4\jet-on-node\public.

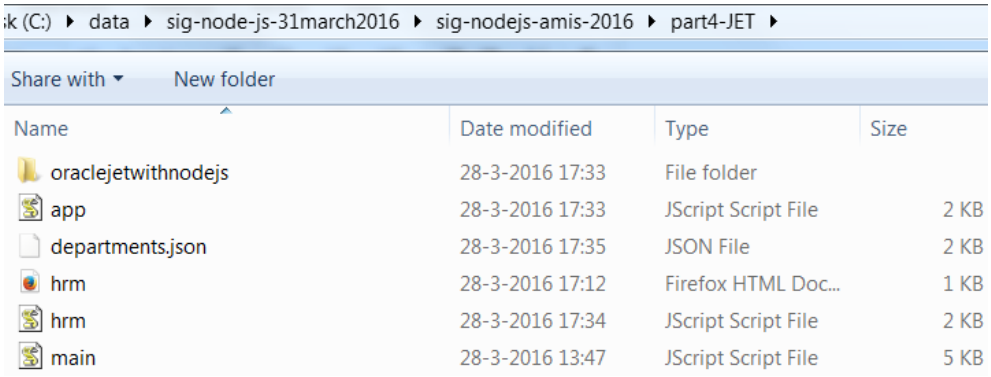
Refresh the browser. You will now see the Oracle JET quick start application:




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.

The steps you need to go through:
- copy or move hrm.js from the workshop folder part4-JET to jet-on-node\public\js\viewModels
- copy or move hrm.html from the workshop folder part4-JET to jet-on-node\public\js\views
- copy or move departments.json from the workshop folder part4-JET to jet-on-node
- edit file main.js in jet-on-node\public\js to make it look like main.js in workshop folder part4-JET (see instructions below)
- edit file app.js in jet-on-node to make it look like app.js in workshop folder part4-JET (see instructions below)




In main.js, you need two add two small sections of code, to add a new HRM tab and component. First, add:

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

in the definition of router.configure:

```
require(['ojs/ojcore', 'knockout', 'jquery', 'ojs/ojknockout', 'ojs/ojrouter',
  'ojs/ojmodule', 'ojs/ojoffcanvas', 'ojs/ojnavigationlist', 'ojs/ojarraytabledatasource'],
  function (oj, ko, $) { // this callback gets executed when all required modules are loaded
    var router = oj.Router.rootInstance;
    router.configure({
      'home': {label: 'Home', isDefault: true},
      'people': {label: 'People'},
      'hrm': {label: 'HRM'},
      'library': {label: 'Library'},
      'graphics': {label: 'Graphics'},
      'performance': {label: 'Performance'}
    });
```

then add

{name: 'HRM', id: 'hrm',
iconClass: 'demo-education-icon-24 demo-icon-font-24 oj-navigationlist-item-icon'},

in the definition of var navData:

```
function RootViewModel() {
  var self = this;
  self.router = router;

  // Shared navigation data and callbacks for nav bar (medium+ screens) and nav list (small screens)
  var navData = [
    {name: 'Home', id: 'home',
      iconClass: 'demo-home-icon-24 demo-icon-font-24 oj-navigationlist-item-icon'},
    {name: 'People', id: 'people',
      iconClass: 'demo-education-icon-24 demo-icon-font-24 oj-navigationlist-item-icon'},
    {name: 'HRM', id: 'hrm',
      iconClass: 'demo-education-icon-24 demo-icon-font-24 oj-navigationlist-item-icon'},
    {name: 'Library', id: 'library',
      iconClass: 'demo-library-icon-24 demo-icon-font-24 oj-navigationlist-item-icon'},
    {name: 'Graphics', id: 'graphics',
```

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

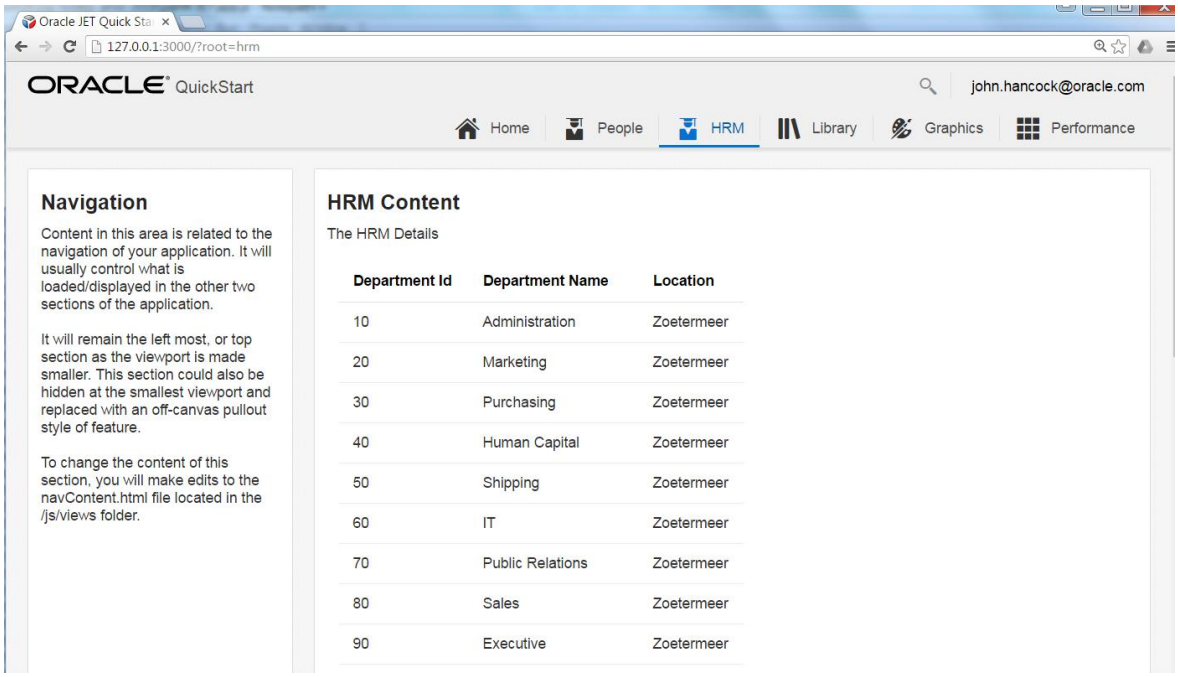
});

```
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. 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).




i) Instead of using the local REST API, you can also link up the application to a remote API – also implemented using Node.js (on Oracle Application Container Cloud) and running live against a DBaaS instance. The URL is https://data-api-lucasjellema.apaas.em2.oraclecloud.com/departments . You can look in the file jet-on-node\public\js \viewModels\hrm.js and swap the local url for this remote endpoint. The data will be slightly different and the waiting will be somewhat longer. So apart from 'just because we can' there is not much reason for actually making this change.

## 6. Complex REST API for retrieving rich Artist information

The REST API that we discussed 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) and the Echonest API (http://developer.echonest.com/api/v4 ).

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.

a) 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.

b) 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) . Try out such a call from your browser or using CURL.

c) 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. 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 that function itself. 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: the artist's biography is included. This information comes from the echonest API.

Inspect the code in artist-enricher-api-2.js to learn about this second API call.

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. Also: the second call (to echonest) does not make use of the outcome from the first call. Yet these two calls are made sequentially, meaning that we spend waiting up to twice as long as should necessary.

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 perform the two outbound REST API calls in parallel rather than sequentially. Check out the documentation for async.parallel at https://github.com/caolan/async#parallel and check the code in artist-enricher-api-3.js to see how this is applied in our Artist API.

   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. Check if the overall response time is shorter than with artist-enricher-api-2.js and the sequential calls.

f) At this point, we do not get any 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.

   Run artist-enricher-api-4.js to see the first iteration of album details in the response. Then inspect the code – the extension to the initial call to Spotify that was added. 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 as it is in async parallel. 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).

g) Spotify offers yet another API that allows us to retrieve details for albums – primarily the release date. In artist-enricher-api-5.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-5.js you will see how a third async mechanism is used: async.forEachOf. The forEachOf function is handed an array – in this case an array of 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-5.js, make one or more calls and verify that the release date is now added for each album.

## 7. Node OracleDB Database Driver

Node applications may want to have access to databases – 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.

# 8. Advanced Topics

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

- use of new language features: Promises and Generators
- working with XML and SOAP

## 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