Track “Adventurer”
SynTouch Bits & Bites - Introduction Node.js

The purpose of this track is to be more challenging than the “Explorer” track, in which a lot of features of Node.js are demonstrated. Here, the intent is that you go out into the wild and actually write some code yourself in order to complete the challenges.

As I do not know exactly how many people will be taking up the challenges and how much development experience they have, the number of challenges is limited.

As always, if you’re faced with a problem, Google is your friend. If you’re stuck: ask!

Project #1 – Create a REST API using MEAN

What the LAMP (Linux, Apache, MySQL & PHP) stack was for early web development, has evolved into the MEAN stack for current web/api development. MEAN stands for:

- MongoDB – NoSQL database as datastore, JavaScript-based
- Express – Node’s favourite API/web site framework
- Angular – Typescript-based (transpiled into JavaScript) front-end
- Node.js – server-side java script framework

In this challenge, we will simply build the API itself and not be concerned with any front-end development (hence the striked-through ~~A~~).

Setting up

Let’s setup the environment by creating a new directory (beer-api) and creating a new package.json file by running an “npm init” inside your newly created directory.

After creating the package, we are going to install some packages to the application (and registering the dependencies in package.json):

- body-parser: parsing incoming request bodies
- morgan: for logging requests
- express: as an API framework
- mongodb: the officially supported database driver for MongoDB.

Importing the data

To have some “body” I have prepared an extract of some beer data (data was taken originally from https://data.world/socialmediadata/beeradvocate); the data is represented as comma-separated values,

the first line is a HEADER line describing the fields: brewery, beertype (LAGER/WEIZEN/INDIA_PALE_ALE), rating, name and ABV (alcohol percentage by volume).

On your Virtual Machine, MongoDB 3.4 has been installed and is started at boot time, does not require authentication and is running on the default port (27017). Consult the MongoDB documentation for mongoimport to find out how to import data into a database (https://docs.mongodb.com/v3.4/reference/program/mongoimport/). The datafile is provided in the Git repo.

Create a new database and a new collection ("table"), use the --drop option to remove the existing collection if present.

Start a different terminal session and connect to MongoDB from it; you can use the command line tool "mongo" (a.k.a. the Mongo shell) for interacting with the MongoDB (as an alternative, there exists a free GUI-based tool for interacting with Mongo as well - https://robomongo.org/).
In our case, the mongo command does not require any parameters.

From the Mongo shell, invoke the help command: help

This command will show you what kind of commands are possible inside mongo; one of the useful commands for exploring the MongoDB server is "show dbs" to list the database hosted by this mongo-server.

Try it and find the database you created on importing the CSV file.

Now, to set a database as the current one, use the "use <db_name>" command, where <db_name> is your database. Make your data great again – euh, set your database as the current one.

After setting the current database, you can use the "show collections" command to list the document collections present in the current database. If you have imported your data into a new database, you should only see one collection present.

Now, you can use help as a stand alone command, or you can invoke the help function on a command to see what options/subcommands are available. If your collection has been named "biertjes", you can see the operations that can be invoked on your collection by entering into Mongo shell:

db.biertjes.help()

One of the useful methods to invoke on the collection is the method "stats()"; this will show you – apart from a lot of other information – the number of rows in the collection and the indexes present. As we

have not (yet) created an index, there will only be an index on the _id column that has been generated by MongoDB upon importing this collection (comparable to the ROWID in the Oracle database).

Let's create a new index for the collection by invoking the createIndex method. As you can see from the MongoDB documentation for this function, you need to specify the parameters/configuration for the call as a JSON documents (https://docs.mongodb.com/v3.4/reference/method/db.collection.createIndex/).

Create a new index on the beer's name; the index must be made unique and be named "biernaam".

Verify after execution using the stats() method that the index has been created.

Getting started ~~with NoSQL~~ without SQL is straightforward; in order to retrieve a record from the collection, you should simply use:

db.collectionname.find({queryDocument}), where queryDocument is the filter to be applied. For equalities, this is easy: just use name : value.

Try to retrieve the beer named "Brand Up" from your collection using the mongo shell.

Can you find out how many WEIZEN beers have an alcohol percentage of 7.2 in this dataset? (Hint: if you provide conditions on multiple fields, these will be applied "AND'ed". For counting, you can append the count() method to your find operation…)

Set up the app

One of the annoying features of express/node is that the source files are not reread on change. So, during development, if you need to change one of your source code files, you will need to stop and start your server manually.
Fortunately, there are methods to circumvent the manual process (nodemon) and this module has been globally installed inside your VM, so every project on your machine can use it.

The standard way to use nodemon is by means of npm: you simply define a new start script inside npm to start the application by using nodemon; so, if your main file is called "app.js", you would add the following line to the script object:

```
"scripts": {
    "start": "nodemon app",
    "test": "echo \"Error: no test specified\" && exit 1"
  }
```

After you have created a runnable script file, you will be able to start it using "npm start" and it will automatically restart upon change of any javascript file inside your project's directories.

A basic version of the application has been provided, you can copy this into the directory where you have created the package.json file and installed the dependencies, in the “Setting up” section.

Small steps

Now, writing code is a process of small steps and regular testing your current setup. From a terminal window (shell) you can run the application with nodemon (which will automatically restart the application as soon as it detects any changes in the source code files) by typing “npm start”. This runs the ‘start’ script from the npm configuration, which you should have set to “`nodemon app.js`” (or whatever main source code file you selected instead).

You should see output of nodemon starting up, together with a log statement that the program has established a database connection and is listening on port 3000 (unless you have set the PORT environment variable):

```
node@node-VirtualBox:/tmp/beer-api$ npm start

> beer-api@1.0.0 start /tmp/beer-api
> nodemon app

[nodemon] 1.17.2
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node app.js`
Connection established, listening started on port: 3000
```

If you start a second terminal (or open another tab in this terminal window as you like), you can access the application by using the curl command line tool to fetch from a URL; as your app is running on your local machine on port 3000 and HTTP GET is the default operation, you can simply enter:

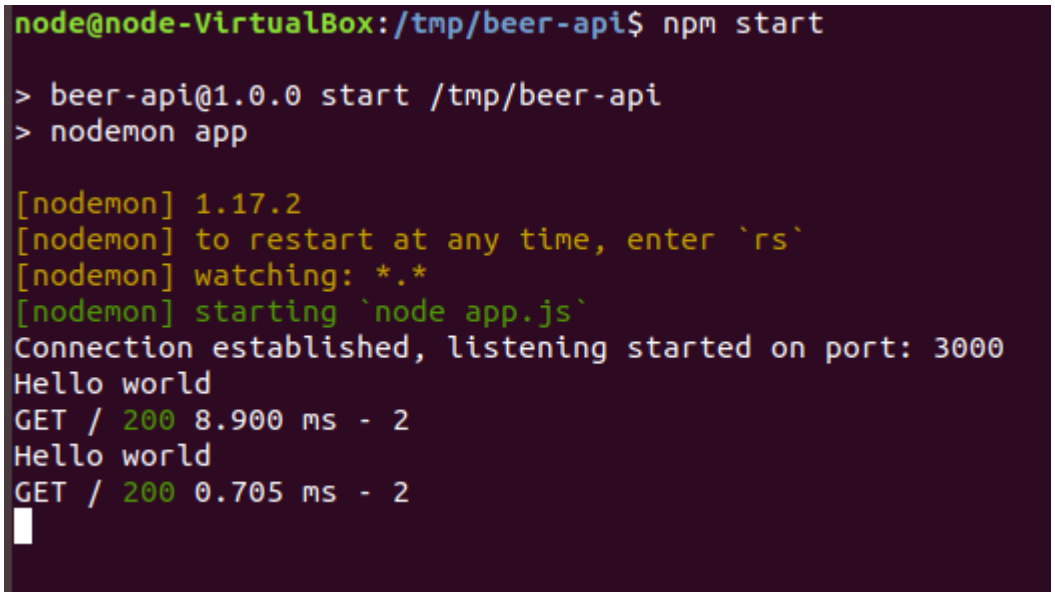
`curl http://localhost:3000`

This should return a text “OK” (verify the app.js source code file):

```
node@node-VirtualBox:/tmp/beer-api$ curl http://localhost:3000
OKnode@node-VirtualBox:/tmp/beer-api$
```

The only thing between this and your new killer app is your imagination!

The output in the nodemon window should be something like this:



"Hello world" is send using a console.log statement, the "OK" you get in response to your request is returned by the res.send/res.sendStatus command with statuscode "200" being the representation of "OK".

Try and see what happens if you omit the res.send/sendStatus command from the app.get request handler.

(For Node 6.x and higher, this should take two full minutes).

Challenge #1: implementing the beer browser

The first challenge in this API is to implement the functionality required to return a (subset) of beers to the requestor when the resource path "/api/beers" is invoked. In Express, the app.get function below implements a "request handler". In the base application that was provided, there is a constant MAX_RESULTS which represents the maximum number of results to be returning to the caller in one execution of the find() method.

Most of the APIs of MongoDB are "fluent", which means you can add 'decorators' modifying the API's behaviour to your operations.

```
// return a limited set of beers from the set
app.get("/api/beers", (req, res) => {

  /*
   * implement api to return a list of beers from the database.
   * the list should be limited to a specific set size.
   * Need to WRAP the results of the query into an JSON object, e.g. { beers : data }
   * additionally, you could extend the query to specify a STARTING position
   * as an HTTP query parameter (start=...); the http query parameter can be accessed
```

```
   * as part of the request object, e.g. req.query.start. You can test whether this
   * contains a valid numerical value by casting its value to a Number and test this
   * against the value "NaN" (Not-a-Number) …
   * And MAX_RESULTS should be applied ...
   *
   * Hint: use the find() operation against the collection containing your beers;
   * multiple results are usually converted using a toArray() operation.
   *
   * Whenever the query returns normally, you should be returning the wrapped results
   * with an HTTP OK (200) status, when the error in the callback is present, this
   * should be return with an status code HTTP Server Error (500)
   */

  YOUR-IMPLEMENTATION-HERE ...

})
```

Test your implementation! Verify what happens if you have implemented the "paging" mechanism by using the start parameter and you are providing different values in your curl command!

Challenge #2: single beer

This should be an easy one, you should be providing the beername as part of the request url and upon receiving such a request, retrieve the beer with the requested name from the database and return this with an HTTP/200 status code. If you do not find the beer, return a JSON object containing an error message (like `Panic! No beer ${req.params.beername}`) and a status code of HTTP NOT FOUND /404 (note that the backticks/backquotes/`` are required in ES6 to template strings where variables will be automatically expanded!):

```
// return a single beer; 200 if found, 404 otherwise with error message
beer not found
app.get("/api/beers/:beername", (req, res) => {

     // typically, this involves returning a SINGLE document by using
     // the findOne operation.
     // The resource name (:beername) in the path may be accessed as
     // req.params.beername

  YOUR-IMPLEMENTATION-HERE ...

});
```

Challenge #3 – beers by brewery, paginated

Return a paginated list (MAX_RESULTS, start parameter) of beers by a given brewery:

```
// return a limited set of beers by the given brewery
app.get("/api/beers/brewery/:breweryname", (req, res) => {
```

```
  YOUR-IMPLEMENTATION-HERE ...

});
```

Challenge #4 – adding beers!

Try to implement the API method to ADD new beers to the collection; the beer should be represented by a document like the following:

```
{
    "name" : "Syn Pale Ale",
    "ABV" : 7,
    "rating" : 5,
    "brewery" : "Brouwerij Dam",
    "beertype" : "INDIA_PALE_ALE"
}
```

To test your implemented, you could be using curl or postman. When using curl, you will need to instruct curl to:

- use the POST method (- X POST)
- provide the correct content type (-H "Content-Type: application/json")
- specify that the data is to be taken from a file instead of directly provided from the command line (e.g. –data @SynPaleAle.json if you have the data stored there – the @ is needed!)

```
/*
 * POST: create a new entry
 * responses: success: status: 201, return URI to created resource
 *            error:   status: 400, return error message
 */
app.post("/api/beers", (req, res) => {

    YOUR-IMPLEMENTATION-HERE ...
});
```

Can you also retrieve the created resource using the provided URI from the success response? (You might have to invoke a function to encode a URI component (the beer's name) …

Challenge #5 – Protecting your beers …

Ouch! Management feels that they should not be exposing their delicate brews without some form of traceability and it is decided that from now on, all invocations to you API must provided an additional HTTP header (APIKEY), whose value must be known to the applicaton.

To make things easy, you should just check against a single value; the verification can be done by so-called "middleware". You have been using some middleware in the previous challenges like logging middleware, the body parser etc.

Defining middleware is easy: a (regular) middleware function is a function that receives three arguments, req, res and next; the first two are the request and response objects, the third one is an actual function that needs to be called whenever the middleware function completes and needs to pass control of the request to the next middleware function (which could be the actual request handler).

So, your task is to implement a function:

```
function authorizeCall( req, res, next) {
  // verify presence of the correctly valued API key header here
  // call next() if it succeeds
  // send HTTP/Unauthorized (401) with a JSON error
  // "Unauthorized call" when the header is missing/has an
  // incorrect value
  // req.headers is an Object; from an Object x, you can access
  // attribute y using x.y
}
```

After implementing the function logic, you need to make sure that the function will be invoked for ALL request handlers. In Express, middleware (and request handlers) will execute in the order in which they are defined, so always calling the middleware function would require calling this with `app.use(functionName)` **before** all request handlers in the source code file.

Project #2 – Writing a Node CLI script

In this challenge, you will be writing a command line script that can be used directly from within the shell. This script:

- Will count words, lines and/or characters
- Take its input from the command line
- Accept multiple command line options
- Process one or more files and present the relevant statistics on a per-file and global basis.

Getting started

Create a new directory for your counting project, 'wc'

Change into this directory and initialize your new project by performing a "`npm init`"
This will walk you through a number of questions regarding project setup, ownership etc., all of which can be answered by accepting the default answer.
As a result, a package.json file will be written.

From the project directory, you can either use your favourite editor (vi, nano, gedit) or us the free Microsoft Visual Studio Code environment (invoke 'code' or launch using the icon from the favourites)

Create a new program file inside the project; probably you have kept the default (index.js), so create this file and write the 'process' variable to the console using the log operation.

When using the VSCode environment, you can open an integrated terminal window from the "View" menu, or by using the keyboard shortcut CONTROL-backtick (`). From this terminal (or your regular terminal window if not using VSCode), run the program using and specify some (dummy) parameters to see where these end up in the process object (this represent the node process, with environment options, filename used for running the script, arguments etc.), e.g.

```
node index.js -f file1 -w -c -l -f file2
```

If you cannot easily find the arguments you provided in the output generated, restrict the output to print the argv property of the process object instead. As you can see, the structure in which the arguments are passed is a simple array. First, you'de have to pop off the first two arguments as they represent the name of the executable (node) and the actual script file. Then you have no built-in checking for mandatory arguments, exclusive option or collection of similar repeated arguments (e.g. -f); also, the argument flags and argument values (-f and file1, file2) are simply associated by their order in the array.

Not very useful. Enter yargs (https://www.npmjs.com/package/yargs); scan the documentation and install this package into your project locally and register the dependency in the package.json file (i.e. use the --save flag)

Import the yargs module into your project file by require-ing it and print its argv property (as opposed to the process' argv property).

Now, to make the script easily runnable, add as a first line the UNIX incantation to have it executed using the node binary (note the space between env and node):

```
#!/usr/bin/env  node
```

Next, set the executable flag on the script file (e.g. chmod 755 index.js or chmod u+x index.js). This combination will enable you to execute the script by simply typing the path name to it (e.g. `./index.js` when in the same directory – having the current directory inside your PATH is considered a security risk – that is why you need to prefix it by ./)

Try and execute the script.

As the yargs module has a fluent interface (https://en.wikipedia.org/wiki/Fluent_interface), this means you can easily tack on some configurators to the required yargs object … and still return a yargs object, albeit a configured one!

Register your first command line option to register the flag used for specifying files using the options method to yargs; this option method requires an object to configure the possible flags (https://github.com/yargs/yargs/blob/ead118aac46fdeb9087bfba22d4ee6623ba886c5/docs/api.md#optionskey-opt):

For specifying the files, we'll use the option 'f'; alias the option to 'files' so you're allowed to use -f file1 file and --files file1 file2 as you like. This command line option must be present (demandOption: true) and must be of the array type to allow multiple values to be provided. Provide a description element (describe attribute) as well to document the behaviour.

Test your yargs setup by running the script with some bogus flags and options while omitting the required -f/--files option:

```
./index.js -w -c -x hoppeta
```

Now you should see a unix like usage message for your script!

Next, add a global description for the command by extending the method call chain on require with an extra call to the usage() method. Specify a description as an argument to usage, e.g. "Usage: $0 – count lines, words and/or characters in files').

Test your addition by again omitting the -f/--files argument to force out the usage information.

Adding counters

Now you have gotten started with yargs … add some options to determine what elements to count by extending the configuration object passed into the options() method, e.g. -w/--words, -c/--chars, -l/--lines.

Challenge: Can you find a way to make sure that at least on of these three options has been provided upon invocation, so you know WHAT you should be counting??

Get me my input!

Project Gutenberg hosts quite a number of public domain texts that are free of copyrights and ideal to use for testing purposes. I have selected some classics, you can obtain the same texts using curl, the easiest would be to do this from inside your project directory to avoid cluttering the command line when invoking your wc with all kinds of file paths:

```
curl http://www.gutenberg.org/cache/epub/16452/pg16452.txt > iliad.txt

curl https://www.gutenberg.org/files/24269/24269-0.txt > odyssey.txt

curl http://www.gutenberg.org/cache/epub/15975/pg15975.txt > camera-obscura.txt

curl http://www.gutenberg.org/cache/epub/11024/pg11024.txt > max-havelaar.txt
```

If you want to, you could have gotten rid of the Windows CR (carriage return, showing up as ^M in the text), by using the dos2unix command (not installed by default – invoking dos2unix will tell you how to install it!) or piping the curl output through the UNIX stream editor, sed, before redirecting the output to the file (e.g. `curl command here… | sed 's/^M//' > redirectionfilename`). The ^M is entered as CTRL-V CTRL-M

Processing arrays

Now, instead of simply requiring yargs, add the .argv attribute to the end of the required chained API and change the constant's name to argv as well, i.e.

```
const argv = require("yargs")
      .options({ some-option-configs })
      .usage("My usage-string").argv;
```

Wow, now argv contains all the parsed options!

Using the async module, you can leverage the map() method to apply a function to each element in a collection – which could potentially be an array of files – and upon completion invoke a callback function

Hence, require the async module and inspect the documentation for async.map (https://caolan.github.io/async/docs.html#map).

Next, you will need to implement the countFile() function using the code below for implementing the async map operation:

```
async.map(
    options.files,
    function (file, callback) {
       // read the file here using fs.readFile
       // countfile will add the current file's counts to a -dirty- GLOBAL var
       // called results
       fs.readFile(file, (err, data) => {
           if (err) callback(err);
           callback(null, countFile(options, file, data));
        });
    },
    function (err, results) {
       // iterate over every results element - i.e. every file's counts
       // print the count per file and keep a running total to be printed
       // after all elements have been processed.
    }
);
```

Each invocation to the countFile() will be provided with three arguments:

- Options: command flags, determine which aspect(s) to count
- File: the name of the file, to be used for labelling
- Data – the contents of the file.

You can create a function to return an object:

```
var result = { file: file };
```

Appending counts is easy and can be done *conditionally*:

```
if (options.line != undefined)
```

```
result.lines = lineCount;
```

In order to obtain the counts, you need to:

1) Initialize counters for words, lines and characters.
2) convert the data using toString()
3) split the data into LINES on each occurrence of a newline character (use the regex pattern /\n/ to achieve this!)
4) Next, you need to loop over all the lines in the result (e.g. by using forEach) and
    a. If line counting is requested, add +1 for each line
    b. If word counting is requested, split the line on every occurrence of one or more whitespaces (regex = /\s+/) and add the output array's length to the current count.
    c. If character counting is requested, simple add the number or characters in the line to the running total
5) Create the result object by adding the requested attributes (depending on which counts should be returned) and return this to the caller.

Results

In the second and final function, you will need to iterate over ALL objects in the results array , using:

```
//initialize your totals here

results.forEach(function(res){

	// over here, you would print the results for each file (res.file)

	// and add the file's counts to the running totals

});

// here you can print the GRAND totals, i.e. the toals over all files
```

TEST!