

Lab 1: Introduction to JavaScript, Node.js and API's

1 Purpose of this lab

This lab is split into two parts. In the first part, we introduce JavaScript and Node. JavaScript is the default scripting language used in all standard web browsers (e.g. Chrome, Safari, FireFox, even Internet Explorer) and is therefore the language for implementing the majority of web applications. Node brings the power of JavaScript to the server-side of an application. Consequently, JavaScript can be used on both the client-side and server-side of a web application. In the second part of this lab, we begin to look at how to use Node to implement API's.

This lab looks long, but it's mostly due to the amount of theory you need to be aware of to get started. Subsequent labs will be more practical based and will require less reading.

2 Overview to JavaScript

Taken from: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction>

JavaScript is a cross-platform, object-oriented scripting language. It is a small and lightweight language. Inside a host environment (for example, a web browser), JavaScript can be connected to the objects of its environment to provide programmatic control over them.

JavaScript contains a standard library of objects, such as Array, Date, and Math, and a core set of language elements such as operators, control structures, and statements. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects; for example:

- Client-side JavaScript extends the core language by supplying objects to control a browser and its Document Object Model (DOM). For example, client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation.
- Server-side JavaScript extends the core language by supplying objects relevant to running JavaScript on a server. For example, server-side extensions allow an application to communicate with a database, provide continuity of information from one invocation to another of the application, or perform file manipulations on a server. Node is the primary server-side Javascript implementation used today.

Throughout this course we will be using JavaScript for the majority of the exercises. Specifically, we will be using the ES2015 standard. If you are new to JavaScript and/or the ES2015 standard then you should take some time now to familiarise yourself. There is an abundance of information available online.

Below are some useful resources:

- <https://developer.mozilla.org/en-US/docs/Web/JavaScript>
- <https://leanpub.com/understandings6>
- <https://developers.google.com/web/fundamentals/getting-started/>
- <https://leanpub.com/understandings6>
- <http://exploringjs.com/>
- <https://leanpub.com/es6-in-practice>
- <http://javascript.crockford.com/>
- <https://github.com/ericdouglas/ES6-Learning#articles--tutorials>

3 What is Node.js?

From their website (nodejs.org):

“Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.”

But what does this mean? Let's break it down:

“Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.”

V8 compiles and executes JavaScript source code, handles memory allocation for objects, and garbage collects objects it no longer needs. JavaScript is most commonly used for client-side scripting in a browser, being used to manipulate Document Object Model (DOM) objects for example. The DOM is not, however, typically provided by the JavaScript engine but instead by a browser. So V8, as a pure Javascript engine, does not include any methods for DOM manipulation. V8 does however provide all the data types, operators, objects and functions specified in the ECMA standard. <http://node.green/> shows exactly which ES2015 features are supported by each Node version. It shows that version 6.4.0 and above have the best support.

“Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.”

Event driven: “In computer programming, event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads.” - https://en.wikipedia.org/wiki/Event-driven_programming

Non-blocking: Read a short blog post on the difference between blocking and non-blocking code at: <https://medium.com/@hengkiardo/blocking-versus-non-blocking-code-d3bde835062f>

“Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.”

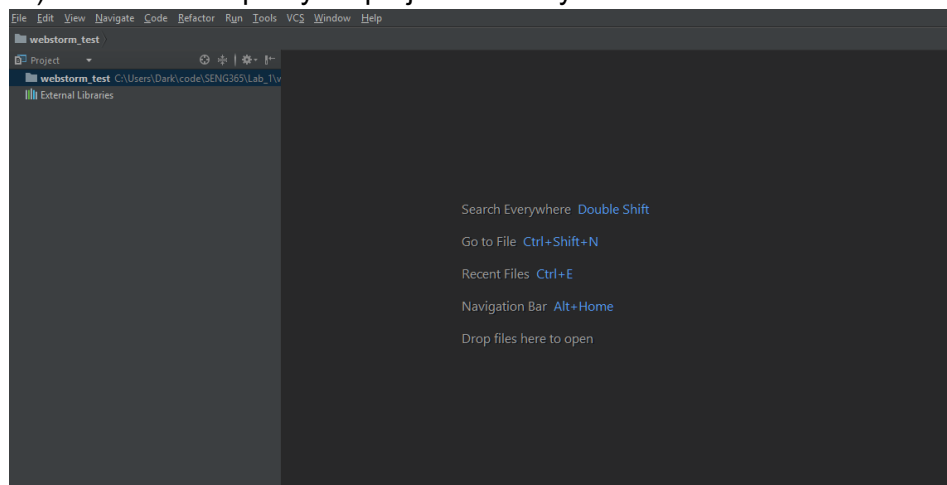
NPM: “npm , short for Node Package Manager, is two things: first and foremost, it is an online repository for the publishing of open-source Node projects; second, it is a command-line utility for interacting with said repository that aids in package installation, version management, and dependency management.” - <https://docs.nodejitsu.com/articles/getting-started/npm/what-is-npm/>

3.1 Before we start: Running scripts in WebStorm

Many of the seasoned JavaScript developers amongst you may already have a preferred workflow for writing web applications. For those of you who do not, then we recommend using the WebStorm IDE. Webstorm is simple to use and already installed in the labs. For any of you wishing to use Webstorm on your personal machines, then you get a free version using your university email account (you may have to reconfirm that you are a student each year).

Running scripts on Webstorm is easy:

1. Open Webstorm
2. Select 'create new project'
3. Create an empty project (make sure to rename your project first and provide a project path). The IDE will open your project directory as shown below.



4. Right click on your project directory and select: New > JavaScript File
5. When prompted for a filename, call it 'test.js'
6. Your new file will open, copy and paste the below code into it:

```
console.log("Hello!");
```

7. Save your file (CTRL + s) and run your script by clicking Run > Run test.js or using the shortcut SHIFT + F10. **Alternatively:** you can run your scripts by opening your terminal (View > Tool Windows > Terminal) and entering the command 'node test.js'

Read more: <https://www.jetbrains.com/help/webstorm/quick-start-guide.html>

Note: You may need to set up Webstorm to validate ES2015 code. To do this go to: File > Settings > Languages & Frameworks > JavaScript and set the JavaScript Language Version

to be ECMAScript 6. (For Mac users, you may need to navigate through a slightly different menu structure.)

3.2 Getting Started: Download and installation (if using a personal computer)

1. Download latest version of Node from <http://nodejs.org/download>.
2. Run the downloaded file and navigate through the wizard.
3. Confirm the installation by opening your terminal and typing 'node.' You should be presented with the node prompt ('>').
4. Confirm that node has installed correctly by moving onto Exercise 1.

3.3 Exercise 1: Hello World

1. Create a file named **exercise1.js** on your machine
2. Open the file and insert the following code

```
/* Hello, World! program in node.js */  
console.log("Hello, World!");
```

3. Open your terminal, navigate to the directory where your file was created and type:
node exercise1.js
4. If node was successfully installed, 'Hello, World!' should be outputted to the console.

3.4 Exercise 2: Write a Web Server

In Node, we load modules into other modules using the 'require' directive. (c.f 'import' in Python).

1. Create a new file called **exercise2.js**
2. Use the 'require' directive to pull in the http module into a variable for later use.

```
const http = require("http");
```

3. Now use the 'http' module's 'createServer' function to create a new server. The example code below ignores the content of any request given and instead will just return a 200 message with the text 'Hello World.'

```
http.createServer(function (request, response) {  
  
  // Send the HTTP header  
  // HTTP Status: 200 : OK  
  // Content Type: text/plain  
  response.writeHead(200, {'Content-Type': 'text/plain'});  
  
  // Send the response body as "Hello World"
```

```
response.end('Hello World\n');
}).listen(8081);

console.log('Server running at http://127.0.0.1:8081/');
```

The documentation for the `createServer` function can be found at:

https://nodejs.org/api/http.html#http_http_createserver_requestlistener

4. Open your terminal and run: **node exercise2.js**
5. Open your browser and navigate to <http://127.0.0.1:8081> or <http://localhost:8081>

3.5 Exercise 3: Handling URL parameters

Next we want to be able to retrieve URL parameters so that we can use them in our applications. Read https://en.wikipedia.org/wiki/Query_string for an explanation of what URL parameters are.

1. Make a copy of your `exercise2.js` file and call it `exercise3.js`
2. Use the `require` directive to pull in the `'url'` module. The `'url'` module allows us to easily parse URLs.

```
const http = require("http"), URL = require('url').URL;
```

3. Next, edit the contents of your `createServer` function. We want to first add a listener on the `end` event of the HTTP request. This event will parse the URL for the parameters and then return a 200 HTTP response with the data in the response body.

```
http.createServer(function (request, response) {
  const parameters = new URL(request.url,
    'http://localhost').searchParams.toString();
  // write the response
  response.writeHead(200, {
    'Content-Type': 'text/plain'
  });

  response.end('Here is your data: ' + parameters);
}).listen(8081);
```

4. Again, run your server and then go to your browser and navigate to the server's URL. Add parameters to your URL, they should be listed in your browser (e.g. <http://localhost:8081?name=Jake&age=35>).

3.6 Exercise 4: Putting it all into practice

In this exercise, we will create a server that contains a shopping list. When a user navigates to the server with the parameter `'item_num'` set, the server will respond with the name of the item at the specified index of `'item_num'`.

1. Create a new file called `exercise4.js`
2. Import the `'http'` and `'url'` modules using the `require` directive
3. Create a server
4. Inside the server, create a variable that contains a list of items that are typical of a shopping basket (e.g. milk, bread, eggs, flour).

5. Attach a listener on the 'end' event of the HTTP request
6. Parse the URL for the parameters
7. Find the value of 'item_num,' you may need to do some research here on how to find a specific parameter.
8. Return a 200 HTTP response that writes out to the browser the item in the list at position 'item_num.'

Now that we have created a simple web application, let's look at creating our first API.

4 Creating an API in Node

4.1 Data formats e.g. JSON

Throughout this course, we will be using JSON objects. JSON stands for JavaScript Object Notation. JSON is a lightweight data-interchange format. W3Schools provide a good introduction and tutorial for getting started with JSON (https://www.w3schools.com/js/js_json_intro.asp).

5 What is ExpressJS?

"Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications." [expressjs.com]

5.1 Exercise 5: Introduction to ExpressJS - Hello World!

Taken from: <https://expressjs.com/en/starter/hello-world.html>

1. First create a directory named lab1_ex5, navigate to this directory in your terminal.
2. Install express using: 'npm install express'

Note: When in the labs, you may need to initialise the 'Python internet enabler' before installing. To do this, search for 'pie' in the application list. You will need to log in and hit 'enable'. Ask a tutor for help.

3. In the lab1_ex5 directory, create a file named app.js and add the following code:

```
const express = require('express');

const app = express();

app.get('/', function(req, res){
  res.send("Hello World!");
});

app.listen(3000, function(){
  console.log("Example app listening on port 3000!");
});
```

```
app.use(function (req, res, next) {  
  res.status(404).send("404 Not Found");  
});
```

4. In your terminal, run 'node app.js'
5. Navigate to <http://127.0.0.1:3000> to see the output

The app starts a server and listens on port 3000 for connections. When receiving a HTTP request for the root ('/') resource, the app responds with "Hello World!" For every other path, it will respond with a '404 Not Found' HTTP response.

Note: you can read more about HTTP ports here:
[https://en.wikipedia.org/wiki/Port_\(computer_networking\)](https://en.wikipedia.org/wiki/Port_(computer_networking))

6 RESTful API's

REpresentational State Transfer (REST) is an architecture that makes use of the HTTP protocol for communicating between different clients and servers on the Web. A REST API provides access to resources that the client can access and modify using the HTTP protocol. Each resource is identified using URI's and unique identifiers. Table 1 below shows how different HTTP methods are used in REST to perform specific actions.

HTTP Method	REST Action	Example URL	Example (A blog)
GET	Retrieve a resource	/articles/1234	Viewing a blog article
POST	Add a new resource	/articles/	Posting a new blog article
PUT	Update an existing resource	/articles/1234	Correcting a spelling mistake in an already posted blog article, by replacing the whole article
DELETE	Delete a resource	/articles/1234	Delete a blog article
PATCH	Partial update to a resource (diff)	/articles/1234	Correcting a word in an already posted blog article, by replacing one word at a given index

Table 1: An example of how different HTTP methods are used within an application

The difference between PUT and PATCH is sometimes confusing. Both are used for updating data, but PUT will update the whole resource while PATCH updates by identifying only the parts the need to change, similar to a diff operation.

6.1 Exercise 6: Basic routing in Express

In Express, we can use these HTTP methods to perform different actions. In this exercise, we will use Express to make a single web page react differently depending on the HTTP method that is used.

1. Create a new directory named lab1_ex6. Inside this directory, use npm to install Express and create an 'app.js' file.
2. As with exercise 5, include the Express module and initiate Express using a variable called 'app'

```
const express = require('express');
const app = express();
```

3. Add a function to manage a user sending a GET request to the applications root resource. This GET request should just return the string 'HTTP request: GET /'

```
app.get('/', function(req, res){
  res.send('HTTP request: GET /');
});
```

4. Next, create similar functions for POST, PUT and DELETE requests.
5. Call the 'listen' function to start a server on port 3000.
6. Run the server by navigating to the directory in the terminal and running 'node app.js'
7. To test your simple API, you can use the following tools:
 - a. POSTMAN (recommended) - <https://www.getpostman.com/> (Postman is installed in the labs. You can also download it from [getpostman.com/app/download/linux64](https://www.getpostman.com/app/download/linux64) and run the Postman binary. Postman will continuously prompt you to create an account but feel free to ignore these. Also, here is a tutorial on how to use Postman: https://www.getpostman.com/docs/postman/sending_api_requests/requests
 - b. CURL - <https://en.wikipedia.org/wiki/CURL> or <https://curl.haxx.se/>

6.2 Exercise 7: Creating an API - Microblogging site

Now that we understand the basic concepts behind Express, we can create our first API. We will create an API for a microblogging site (like Twitter), beginning with the following functionality:

HTTP Method	URL	Action
GET	/users	List all users
GET	/users/:id	List one user
POST	/users	Add a new user

PUT	/users/:id	Update a user
DELETE	/users/:id	Delete a user

1. Create a new directory called lab1_ex7
2. Inside this directory, create a new file called 'app.js' and use npm to install express
3. Inside this file, import the express module and initiate express inside a variable called 'app'
4. Copy and paste the JSON from Appendix A into a file called 'users.json' (**Beware:** copying and pasting directly from this PDF may result in encoding issues).
5. Import the list of JSON users into your application using the 'require' directive. You can confirm that this has worked by printing out to the console.

```
const data = require('./users.json');
const users = data.users;
console.log(users[0]);
```

6. Now we can start to build our API functionality. Create the list all users function that returns the JSON

```
app.get('/users', function (req, res) {
  res.send(users);
});
```

7. Creating the function to list one specific user is more difficult as we have to retrieve this user from the list. Create the function using the code below.

```
app.get('/users/:id', function(req, res){
  let id = req.params.id;
  let res_data = "No user";

  for (let user of users){
    if(id == user.id){
      res_data = user;
      break;
    }
  }

  res.send(res_data);
});
```

8. Now we will write the post request. This time the request will contain a JSON object that contains the data required for the new user. Before doing so, we need to include the 'body-parser' module. Install it using npm and import it into the module.
9. Initiate it as shown below.

```
// Import the module
const bodyParser = require('body-parser');
// Tell the express app to expect json in the body of the request
```

```
app.use(bodyParser.json());
```

10. Now we can write our POST function.

```
app.post('/users', function(req, res){
  let user_data = req.body;

  users.push(user_data);
  res.send(users);
});
```

11. Using the 'list one user' and 'add a new user' functions as a template, we can create a function for the PUT method

```
app.put('/users/:id', function(req, res){
  let id = req.params.id;
  let user_data = req.body;

  for (let user of users){
    if(id === user.id){
      let uid = users.indexOf(user);
      users[uid] = user_data;
      break;
    }
  }

  res.send(user_data);
});
```

12. Finally, to create our DELETE method, we use the JavaScript 'delete' operator.

```
app.delete('/users/:id', function(req, res){
  let id = req.params.id;

  for (let user of users){
    if(id === user.id){
      let uid = users.indexOf(user);
      // remove 1 item at index 'uid'
      users.splice(uid, 1);
    }
  }

  res.send(users);
});
```

13. Make your application listen on port 3000 and test using Postman or CURL.

NOTE: The remaining exercises have been left vague intentionally; how you implement the changes is up to you.

6.3 Exercise 8: Adding followers

For this exercise, make a copy of your solution to exercise 7, renaming it to lab1_ex8. Then add functionality that allows users to follow other users. The JSON for each user should include a 'following' key, the value of which is a list of ID's that the user follows.

Make sure you add functionality for the following:

1. Add the followers feature to the project, by copying and updating the existing functionality.
2. Add a 'follow' function to the API, this function will add a new user ID to a specified users following list.
3. Add an 'unfollow' function.
4. Add a 'view_followers' function, this function will show all the users that a specified user is following.

6.4 Exercise 9: Adding microblog posts

Each user should also have a list of microblog posts. Functionality should exist that:

1. Creates a new post for a user
2. Retrieves all of a user's posts
3. Retrieves a single post of a specified user
4. Updates an existing post for a user
5. Deletes a post
6. Retrieves all the posts from all the followers of a specified user
7. **(Extra challenge)** Implement 'Likes' on posts

We have now written our first API using Node. However, a problem with our application is that the data isn't persisted anywhere. Once the server is stopped (or crashes), we will lose all of our users. In the next lab, we will look at persisting our application data to a database.

Appendix A: users.json

The JSON data below was generated using an automatic online data generator (<http://www.json-generator.com/>). We use this data in the exercises for this lab as an example.

(If you have problems copying and pasting this text, try the users.json file available on learn)

```
{
  "users": [
    {
      "id": "1001",
      "age": 35,
      "first_name": "Burch",
      "last_name": "George",
      "gender": "male",
      "email": "burchgeorge@geofarm.com"
    },
    {
      "id": "1002",
      "age": 31,
      "first_name": "Rachelle",
      "last_name": "Chang",
      "gender": "female",
      "email": "rachellechang@geofarm.com"
    },
    {
      "id": "1003",
      "age": 38,
      "first_name": "Sheri",
      "last_name": "Bennett",
      "gender": "female",
      "email": "sheribennett@geofarm.com"
    },
    {
      "id": "1004",
      "age": 32,
      "first_name": "Fisher",
      "last_name": "Dillard",
      "gender": "male",
      "email": "fisherdillard@geofarm.com"
    },
    {
      "id": "1005",
      "age": 20,
      "first_name": "Pope",
      "last_name": "Bailey",
      "gender": "male",
      "email": "popebailey@geofarm.com"
    }
  ]
}
```