Seneca
SCHOOL OF INFORMATION AND
COMMUNICATIONS TECHNOLOGY

WEB322

Web Programming Tools and Frameworks

Schedule | Graded Work | Resources | Cyclic Guide | MyApps Instructions | Code examples

WEB322 Week 2 Notes

Introduction to Node.js

As we learned last week, Node.js is actually a JavaScript runtime environment based on Chrome's V8 engine. It is a command-line program (written in C++) that you can install on your local machine or on a web-server that will take your JavaScript code and execute it. This means that we don't actually need a web browser to execute JavaScript at all – we just need a JavaScript engine. Why don't we try a short example:

1. If you haven't already, be sure to download and install the current release of Node.js. If you're not sure whether or not you have Node.js installed, open the **Command Prompt** and type **node -v**. If Node.js has been installed, this will output the version.

2. Make sure you have Visual Studio Code installed. As we discussed last week, this is an open-source, cross-platform development environment provided by Microsoft. While it is true that you can write your code in any text editor, Visual Studio Code works very nicely alongside Node.js and all examples going forward will assume that you are using Visual Studio Code. You can download it here

3. On your Local computer, navigate to your desktop and **create a folder** called **Ex1**

4. Open **Visual Studio Code** and select **File -> Open Folder**. Choose your newly created **"Ex1"** Folder and click **"Select Folder"**

5. You should see an "Explorer" pane open on the left side with two items: "Open Editors" and "Ex1". Click to expand "Ex1" and locate the "New File" button ( ). Click this and type **"hello.js".**

6. You should now see your newly created "Hello.js" file in the editor. Enter the following line of code:

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

   and click **File -> Save (Ctrl + S)**

7. Open the **Integrated Terminal** by selecting **View -> Integrated Terminal (Ctrl + `)** and type:

   ```
   node hello.js
   ```

Hello World! This is the most basic example in Node.js – notice how we didn't need to open a web browser, scratchpad, devtools, etc? It's also important to note that the command **"node hello.js"** can be executed in any command prompt as long as the active working directory is set to wherever your **hello.js** file is located (Ex1 in this case). The Integrated Terminal is just a quick, easy way to get a command prompt running in the correct location without leaving the development environment.

Regarding the code that we wrote, it's very simple; however we have made an important assumption: that we have access to a global **"console"** object. In Node.js we have access to a number of global objects / variables in addition to the built-in objects that are built into the JavaScript language. Some of the Node.js Globals that we will be using include:

Console	The console object provides a simple debugging console that is similar to the JavaScript console mechanism provided by web browsers. Some of the key methods that we will be using are: • console.log() • console.time() / console.timeEnd() • console.dir()
Process	The process object is a global instance of the EventEmitter class that provides information about, and control over, the current Node.js process. It exposes many properties, methods and events related to controlling system interactions. Some of the key elements that we will be using are: • Methods: process.on(), process.abort(), process.kill(), process.exit() • Properties: process.stdin, process.stdout, process.stderr, process.pid,

process.env
- Events: beforeExit, Exit, uncaughtException

__dirname

The name of the directory that the currently executing script resides in.

For example: if our .js file is located in /Users/pcrawford/ex1.js:

```
console.log(__dirname);
// outputs /Users/pcrawford
```

__filename

The filename of the code being executed. This is the resolved absolute path of this code file.

For example: if our .js file is located in /Users/pcrawford/ex1.js:

```
console.log(__filename);
// outputs /Users/pcrawford/ex1.js
```

setTimeout()

This function will execute a piece of code (function) after a certain delay. It accepts 3 parameters:

- **callback** Function: The function to call when the timer elapses.
- **delay** number: The number of milliseconds to wait before calling the callback
- **[, ...arg]** Optional arguments to pass when the callback is called.

For example:

```
// outputs "Hello after 1 second" to the console
setTimeout(function(){
    console.log("Hello after 1 second");
}, 1000);
```

setInterval()

This function will execute a piece of code (function) after a certain delay and continue to call it repeatedly. It accepts 3 parameters (below) and returns a timeout object

- **callback** Function: The function to call when the timer elapses.
- **delay** number: The number of milliseconds to wait before calling the callback
- **[, ...arg]** Optional arguments to pass when the callback is called.

Note: Unless you want the interval to continue forever, you need to call clearInterval() with the timeout object as a parameter to halt the interval

For example:

```
var count = 1; // global counter
var maxCount = 5; // global maximum

var myCountInterval = setInterval(function () {
    console.log("Hello after " + (count++) + "
second(s)");
    checkMaximum();
}, 1000);

var checkMaximum = function () {
    if (count > maxCount) {
        clearInterval(myCountInterval);
    }
}
```

require()	The require function is the easiest way to include modules that exist in separate files. The basic functionality of require is that it reads a javascript file, executes the file, and then proceeds to return the exports object. More about modules and the require() function discussed below.

Now that we have written and executed our very first program with JavaScript using Node.js – why don't we try another example. This time, lets add some user input. Recall when we first discussed JavaScript in the previous course, we did not capture user input using JavaScript, but instead relied on explicitly setting test values at the top of our programs. This allowed us to have specific control over the type of input we were testing. However, it might be interesting to deal with "live data" in a pure JS environment and fortunately for us, we have access to the extensive collection of modules that come bundled with Node.js – more specifically, we have access to the readline module, which can be used for this purpose, for example:

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

var rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('Enter Your Name: ', function(answer){
  console.log('Hello ' +  answer);
  rl.close();
});
```

Notice the first line of our code invokes the global **require()** function which returns an object from the core "readline" module, which we store in a variable named "readline". Now we have access to all of the functionality from the **readline** module, including an input/output mechanism via the createInterface() method which creates a new Interface instance (named "r1" from above). This exposes a handy method called question() that we can use to capture and output user data at the command prompt.

Node also includes a number of other extremely useful core "modules", including:

util	A collection of helper utility methods you can use with your applications in node.js
path	The path module provides utilities for working with file and directory paths. This will come in quite handy when working with reading template files or writing uploaded files for example.
events	The events module allows you to register an event 'listener' and act on those events within node. You can emit an event when a certain condition happens and node will automatically call a function connected to the listening event.
fs	This module is used to work directly with the file system to read and write files. All methods of fs have synchronous and asynchronous versions of the methods. Depending on your needs, you can make use of either type but typically it is best to use the async versions to avoid blocking the node event loop.

Modules

We can also create our own Modules that work the same way, by making use of a global "module" object – which isn't truly "global" in the same sense as "console", but instead global to each of your modules, which are located in separate .js files. For example, consider the two following files (modEx1.js: the main file that node will execute, and message.js: the file containing the module):

file ./modEx1.js

```
var message = require("./modules/message.js");

message.writeMessage("Hello World!");

message.readMessage();
```

file: ./modules/message.js

```
// NOTE: Node.js wraps the contents of this file in a function:
// (function (exports, require, module, __filename, __dirname) { ... });
// so that we have access to the working file/directory names as well
// as creating an isolated scope for the module, so that our
// variables are not global.

var localFunction = function () {
  // a function local to this module
}

var localMessage = "";

module.exports.writeMessage = function(msg){
    localMessage = msg;
}

module.exports.readMessage = function () {
  console.log(localMessage + " from " +  __filename);
}
```

Executing the code in modEx1.js (ie: **node modEx1.js**) should output:

“Hello World” from …

where … is the absolute location of the message.js file in your system, for example:
C:\Users\patrick.crawford\ModTest\modules\message.js

Notice how our “message” module uses the exports property of the “module” object to store functions and data that we want to be accessible in the object returned from the require(“./modules/message.js”); function call from modEx1.js. Generally speaking, if you want to add anything to the object returned by “require” for your module, it’s added to the module.exports object from within your module. In this case, we only added two functions (readMessage() and writeMessage()).

Using this methodology, we can safely create reusable code in an isolated way that can easily be added (plugged in) to another .js file.

NPM – Node Package Manager

The Node package manager is a core piece of the module based node ecosystem. The package manager allows us to create reusable modules that can be packaged and put on the npm repository for others to use. We will make heavy use of the Node Package Manager in this course.

Node Package Manager (npm for short), is installed by default when you install node. From the command line you can run ‘npm’ with various commands to download and remove packages for use with your Node applications. When you have installed a package from npm you use it in the same way as using your own modules like above, with the require() function.

All npm packages that you install locally for your application will be installed in a node_modules folder in your project folder.

While there are over 60 “npm” commands available, the ones that we will most commonly use in this course are as follows:

npm install [Module Name] EX: npm install express	install is used to install a package from the npm repository so that you can use it with your application. EX: var express = require("express");
npm uninstall [module name]	uninstall does exactly what you would think, it uninstalls a module from the node_modules folder and your application will no longer be able to require() it.
npm init	create a new package.json file for a fresh application. More on this part later.
npm prune	The prune command will look through your package.json file and remove any npm modules that are installed that are not required for your project. More on this part later.
npm list	Show a list of all packages installed for use by this application.

Globally installing packages

Every so often, you will want to install a package globally. Installing a package globally means you will install it like an application on your computer which you can run from the command line, not use it in your application code. For example, some npm packages are tools that are used as part of your development process on your application:

One example is the migrate package which allows you to write migration scripts for your application that can migrate your data in your database and keep track of which files have been run.

Another example is grunt-cli so that you can run grunt commands from the command line to do things like setup tasks for running unit tests or checking for formatting errors in code before pushing up new code to a repository.

A third example is bower. Bower is a package manager similar to npm but typically used for client side package management. To install a package globally you just add the -g switch to your npm install command. For example:

```
npm install bower -g
```

Globally installed packages do not get install in your node_modules folder and instead are installed in a folder in your user directory. The folder uses for global packages varies for Windows, Mac, and Linux. See the documentation if you need to find globally installed packages on your machine.

package.json explained

The Node Package Manager is great. It provides an easy way to download reusable packages or publish your own for other developers to use. However, there are a few problems with sharing modules and using other modules, once you want to work on an application with someone else. For example:

How are you going to make sure everyone working on your project has all the packages the application requires?

How are you going to make sure everyone has the **same version** of all those packages?

And lastly, how are you going to handle updating a package and making sure everyone else on your project updates as well?

This is where the package.json file comes in.

The package.json file is a listing of all the packages your application requires and also which versions are required. It provides a simple way for newcomers to your project to get started easily and stay up to date when packages get updated.

The npm documentation for the package.json file has all the information you will need as you begin building applications in node.js

Let’s look at a simple package.json example file

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (seneca) seneca
version: (1.0.0)
description:
entry point: (index.js) week2.js
test command:
git repository:
keywords:
author:
license: (ISC) MIT
About to write to C:\seneca\package.json:
```

```
{
  "name": "seneca",
  "version": "1.0.0",
  "description": "",
  "main": "week2.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "MIT"
}

Is this ok? (yes) yes
```

You can start your own package.json file from scratch but it is much easier to run an **npm init** in your project folder, answer a few questions, and your initialized package.json file will be generated for you. Once generated, you can edit it if you decide to change the name or version (for example). Once you decide to add packages to your app you can simply install the package with **npm install**. This will save the package and version into the package.json file for you so that when others want to work on your app, they will have the package.json file and can use **npm install** to install all the required dependencies with the right version. Think of package.json as a checklist for your application for all of its dependencies.

Building a simple web server using Node.js with Express.js

In week 4, your going to learn more about express.js but for now let's introduce it as a module you can install from NPM that has lot's of code that wraps up more complicated code and as a result it makes writing your own web server MUCH easier than writing it from scratch in node.js

First, we will install the package to our app before we even start building our server

```
$ npm install express
seneca@1.0.0 C:\seneca
`-- express@4.14.0
  +-- accepts@1.3.3
  | +-- mime-types@2.1.13
  | | `-- mime-db@1.25.0
  | `-- negotiator@0.6.1
  +-- array-flatten@1.1.1
  +-- content-disposition@0.5.1
  +-- content-type@1.0.2
  +-- cookie@0.3.1
  +-- cookie-signature@1.0.6
  +-- debug@2.2.0
  | `-- ms@0.7.1
  +-- depd@1.1.0
  +-- encodeurl@1.0.1
  +-- escape-html@1.0.3
  +-- etag@1.7.0
  +-- finalhandler@0.5.0
  | +-- statuses@1.3.1
  | `-- unpipe@1.0.0
  +-- fresh@0.3.0
  +-- merge-descriptors@1.0.1
  +-- methods@1.1.2
  +-- on-finished@2.3.0
  | `-- ee-first@1.1.1
  +-- parseurl@1.3.1
  +-- path-to-regexp@0.1.7
  +-- proxy-addr@1.1.2
  | +-- forwarded@0.1.0
  | `-- ipaddr.js@1.1.1
  +-- qs@6.2.0
  +-- range-parser@1.2.0
  +-- send@0.14.1
  | +-- destroy@1.0.4
  | +-- http-errors@1.5.1
  | | +-- inherits@2.0.3
  | | `-- setprototypeof@1.0.2
  | `-- mime@1.3.4
  +-- serve-static@1.11.1
  +-- type-is@1.6.14
  | `-- media-typer@0.3.0
  +-- utils-merge@1.0.0
  `-- vary@1.1.0

npm WARN seneca@1.0.0 No description
npm WARN seneca@1.0.0 No repository field.
```

Now if we take a look at the package.json file you can see that it has a new section that has been added for the express dependency.

```
"dependencies": {
  "express": "^4.14.0"
}
```

This means that if we give our project to someone else now they can just type **npm install** and it will install the dependencies listed here and the app should be able to run and have everything it needs.

Now let's start our week2.js file and create a web server in 13 lines of code! Create a new week2.js that looks like this:

```
var express = require("express");
var app = express();

var HTTP_PORT = process.env.PORT || 8080;

// call this function after the http server starts listening for requests
function onHttpStart() {
  console.log("Express http server listening on: " + HTTP_PORT);
}

// setup a 'route' to listen on the default url path (http://localhost)
app.get("/", function(req,res){
    res.send("Hello World<br /><a href='/about'>Go to the about page</a>");
});

// setup another route to listen on /about
app.get("/about", function(req,res){
    res.send("<h3>About</h3>");
});

// setup http server to listen on HTTP_PORT
app.listen(HTTP_PORT, onHttpStart);
```

You can now run this web server by typing **node week2** from the commandline.

```
$ node week2
Express http server listening on: 8080
```

and visit the website by navigating to **http://localhost:8080**

Sending a HTML page back from a "get" request

Now that we know how to send messages back from our server, it's very simple to extend this functionality to return files (ie, HTML pages) instead.

To begin, we must first create a "views" folder for our HTML files inside the working (open) folder.

```
/node_modules
/views
```

```
week2.js
package.json
```

Next, we must add a new require for the path module at the top of our week2.js file.

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

And most importantly, add the new about.html page inside our new views folder:

```
<!doctype html>
<html>

<head>
  <title>About</title>
</head>

<body>
  <h1>About</h1>
  <p>This is what it's all about.<br /><a href='/'>Go back to home</a></p>
</body>

</html>
```

Your project folder should now look something like the below:

```
/node_modules
/views
  about.html
week2.js
package.json
```

In order to serve this page, we make a small change to our "/about" route (ie; use the **sendFile** method instead of the send method on the response object).

```
// setup another route to listen on /about
app.get("/about", function(req,res){
  res.sendFile(path.join(__dirname,"/views/about.html"));
});
```

To test your server, run **node week2** to see the results on **http://localhost:8080**

Running this example on Cyclic

If we wish to see this code run on Cyclic, we follow the same procedure highlighted in the Getting Started with Cyclic guide.

However, there is one small change that we can make to our package.json file to ensure that the correct .js file is started on Cyclic when we deploy our app. This is necessary if we have multiple server files for debugging and testing; as is the case with the "example code" from github – it contains multiple servers, one for each week!

Fortunately, when you push your code to GitHub, Cyclic will start the automated build process and run the command **npm start** once it is complete. This command (npm start) looks in the package.json file for the "start" property within the "scripts" property and run that command (ie, **node somefile.js**). Let's make sure there is a start property on the scripts property of the package.json file for this example.

```
{
  "name": "seneca",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "scripts": {
    "start": "node week2.js"
  },
  "author": "",
  "license": "MIT",
  "dependencies": {
    "express": "^4.14.0"
  }
}
```

This will ensure that Cyclic runs the week2.js file when you push your code to the new app!

```

## Sources

- https://nodejs.org
- https://developer.mozilla.org
- Package.json documentation

---
```