COMP2068 – JavaScript Frameworks

Lesson 2
Closures & NPM

Lesson Objectives

In this Lesson we will learn about:

1. Closure structures in JavaScript
2. Node Package Manager (NPM)

Intro to Closures

- Last week we looked at some new strange syntax in JS, where we assigned a variable to a function:

```
let food = fs.readFile('food.txt', 'utf8', (err, food) => {
    console.log(food)
})
```

- This structure is called a **Closure** and it’s vital to asynchronous code in Node

What is a Closure?

- A function referencing variables from parent environment

```
function parent() {
   let message = 'Hello World';

   function child() {
      console.log (message);
    }
    child();
}

parent();
```

- child() can access variables from parent()

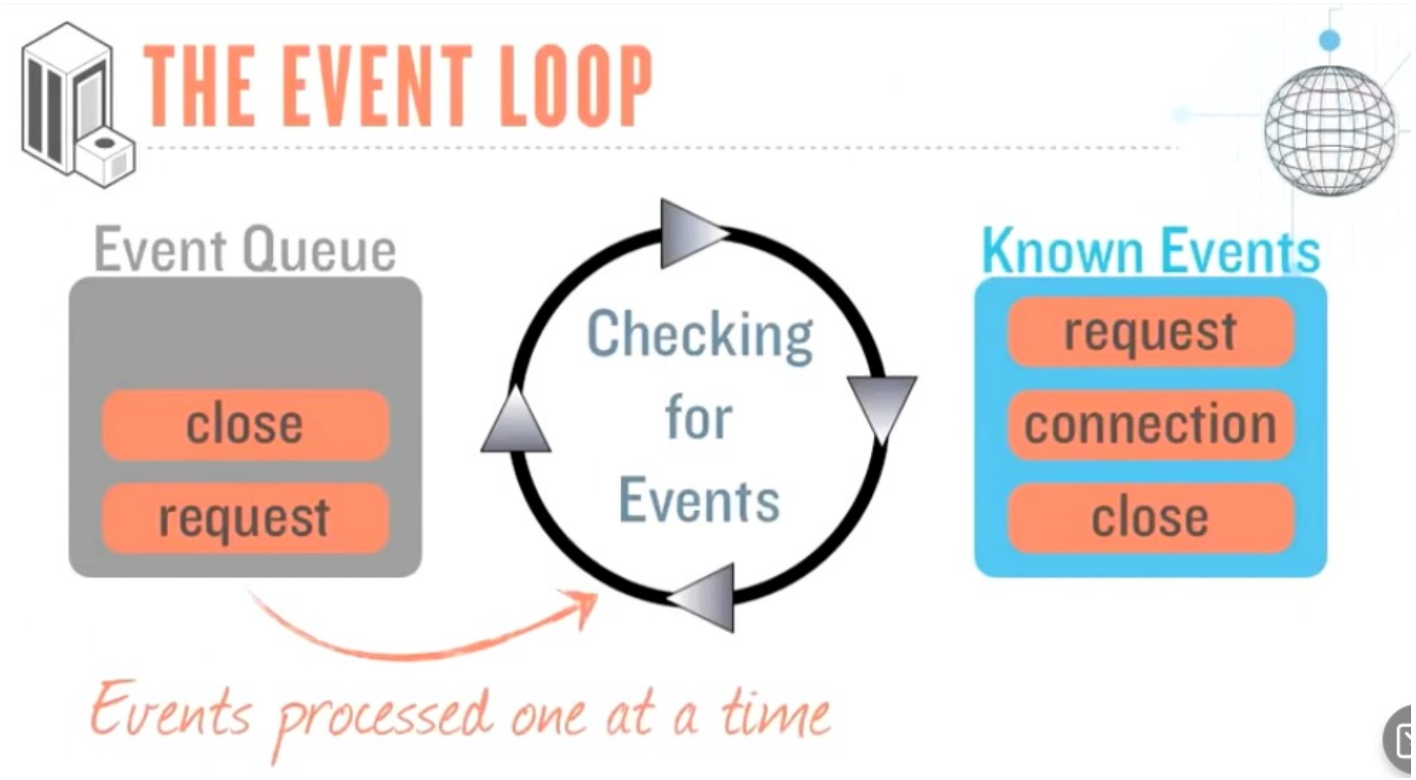
Assigning a Variable to a Closure

```
function parent() {
   let message = 'Hello World'
   function child() {
      console.log (message)
   }
   return child
}

let childFunction = parent()
childFunction()
```

- Closure: a function AND its environment, including its variables
- This allows us to pass functions with their variables as arguments to other functions
- So What?

Closures & the Node Event Loop



Source: https://www.youtube.com/watch?v=GJmFG4ffJZU

Introducing NPM

- **Node.js** is a platform, which means its features and APIs are kept to a minimum.
- To achieve more complex functionality, it uses a module system that allows you to extend the platform.
- The best way to install, update, and remove Node.js modules is using the **NPM** (Node Package Manager).
- **NPM** has the following main features:
  - A registry of packages to browse, download, and install third-party modules
  - A CLI tool to manage local and global packages

Using NPM

Installing a package using NPM

- Once you find the right package, you'll be able to install it using the **`npm i`** command as follows:

```
$ npm i <Package Unique Name>
```

- Installing a module globally is similar to its local counterpart, but you'll have to add he **–g** flag as follows:

```
$ npm i –g <Package Unique Name>
```

- For example, to locally install Express, you'll need to navigate to your application folder and issue the following command:

```
$ npm i express
```

Using NPM (cont’d)

- The preceding command will install the latest stable version of the Express package in your local **node_modules** folder.
- Furthermore, NPM supports a wide range of semantic versioning, so to install a specific version of a package, you can use the **`npm i`** command as follows:

```
$ npm i <Package Unique Name>@<Package Version>
```

- For instance, to install the latest major version of the Express package, you'll need to issue the following command:

```
$ npm i express --save
```

Using NPM (cont’d)

Removing a package using NPM

- To remove an installed package, you'll have to navigate to your application folder and run the following command:

```
$ npm uninstall < Package Unique Name>
```

- NPM will then look for the package and try to remove it from the local **node_modules** folder.
- To remove a global package, you'll need to use the **`-g`** flag as follows:

$ npm uninstall –g < Package Unique Name>

Using NPM (cont’d)

Updating a package using NPM

- To update a package to its latest version, issue the following command:

```
$ npm update < Package Unique Name>
```

- NPM will download and install the latest version of this package even if it doesn’t exist yet.
- To update a global package, use the following command:

```
$ npm update –g < Package Unique Name>
```

Managing dependencies using the package.json file

- Installing a single package is nice, but pretty soon, your application will need to use several packages, and so you'll need a better way to manage these **package dependencies**.
- For this purpose, NPM allows you to use a configuration file named **package.json** in the root folder of your application.
- In your package.json file, you'll be able to define various metadata properties of your application, including properties such as the **name**, **version**, and **author** of your application.
- This is also where you define your **application dependencies**.

Managing dependencies using the package.json file (cont’d)

- The package.json file is basically a JSON file that contains the different **attributes** you'll need to describe your application properties.
- An application using the latest Express and Grunt packages will have a **package.json** file as follows:

```
{
   "name" : "MEAN",
   "version" : "0.0.1",
   "dependencies" : {
      "express" : "latest",
      "grunt" : "latest"
   }
}
```

Managing dependencies using the package.json file (cont'd)

Creating a package.json file

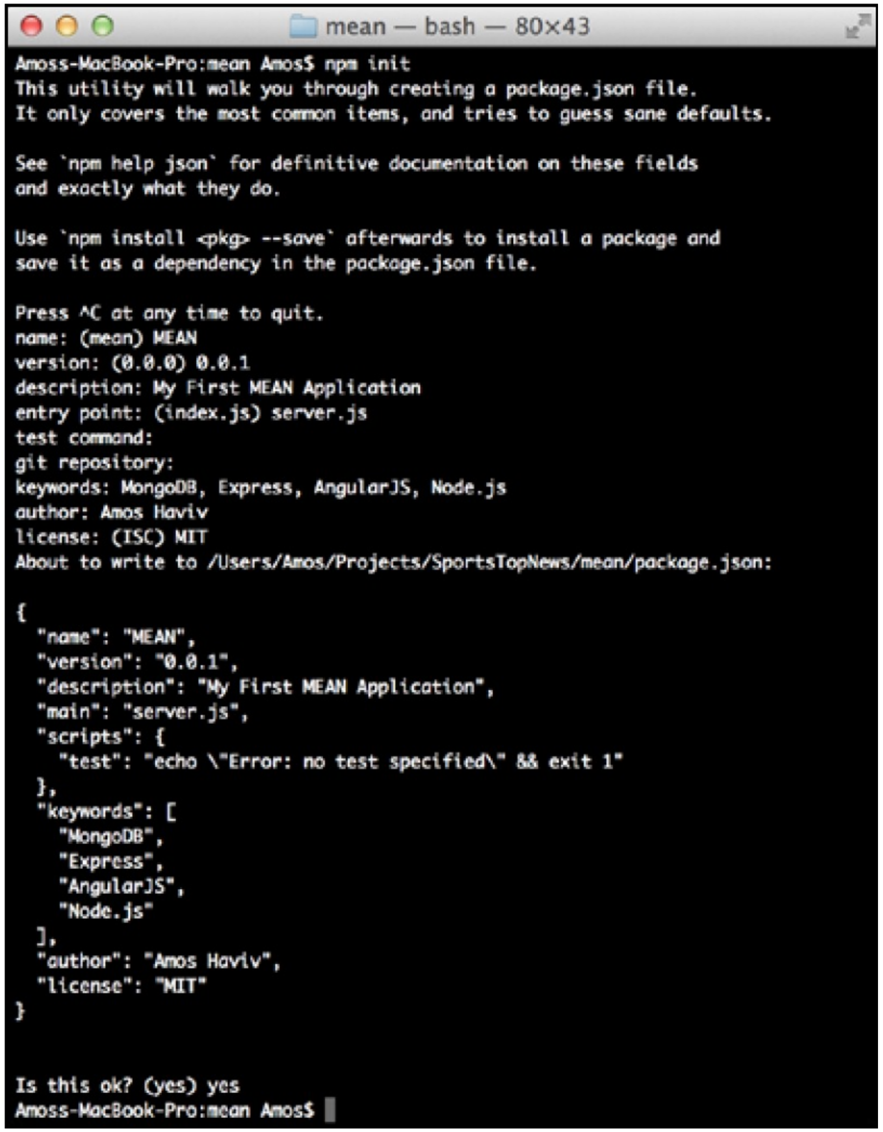
- While you can manually create a package.json file, an easier approach would be to use the npm init command. To do so, use your command-line tool and issue the following command:

```
$ npm init
```

- NPM will ask you a few questions about your application and will automatically create a new **package.json** file for you.
- A sample process should look similar to the following screenshot:

Managing dependencies using the package.json file (cont'd)

- A sample process should look similar to the following screenshot:



```
Amoss-MacBook-Pro:mean Amos$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.

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

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

Press ^C at any time to quit.
name: (mean) MEAN
version: (0.0.0) 0.0.1
description: My First MEAN Application
entry point: (index.js) server.js
test command:
git repository:
keywords: MongoDB, Express, AngularJS, Node.js
author: Amos Haviv
license: (ISC) MIT
About to write to /Users/Amos/Projects/SportsTopNews/mean/package.json:

{
  "name": "MEAN",
  "version": "0.0.1",
  "description": "My First MEAN Application",
  "main": "server.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "MongoDB",
    "Express",
    "AngularJS",
    "Node.js"
  ],
  "author": "Amos Haviv",
  "license": "MIT"
}


Is this ok? (yes) yes
Amoss-MacBook-Pro:mean Amos$
```

Managing dependencies using the package.json file (cont’d)

Installing the package.json dependencies

- After creating your package.json file, you'll be able to install your application
- dependencies by navigating to your application's root folder and using the npm install command as follows:

```
$ npm i
```

- NPM will automatically detect your package.json file and will install all your application dependencies, placing them under a local node_modules folder.
- An alternative and sometimes better approach to install your dependencies is to use the following npm update command:

```
$ npm update
```

- This will install any missing packages and will update all of your existing dependencies to their specified version.

Managing dependencies using the package.json file (cont'd)

Updating the package.json file

- Another robust feature of the **`npm i`** command is the ability to install a new package and save the package information as a dependency in your **package.json** file.
- This can be accomplished using the **`--save`** optional flag when installing a specific package.
- For example, to install the latest version of Express and save it as a dependency, you can issue the following command:

```
$ npm i express --save
```

Node Modules

- JavaScript has turned out to be a powerful language with some unique features that enable efficient yet maintainable programming.
- Its **closure pattern** and **event-driven behavior** have proven to be very helpful in real-life scenarios, but like all programming languages, it isn't perfect, and one of its major design flaws is the sharing of a single **global namespace**.
- This could have been a major threat for Node.js evolution as a platform, but luckily a solution was found in the **CommonJS** modules standard.

CommonJS Modules

- **CommonJS** is a project started in 2009 to standardize the way of working with JavaScript outside the browser.
- The project has evolved since then to support a variety of JavaScript issues, including the global namespace issue, which was solved through a simple specification of how to write and include isolated JavaScript modules.
- The **CommonJS standards** specify the following three key components when
- working with modules:
  - **require():** This method is used to load the module into your code.
  - **exports:** This object is contained in each module and allows you to expose pieces of your code when the module is loaded.
  - **module**: This object was originally used to provide **metadata** information about the module. It also contains the pointer of an exports object as a property. However, the popular implementation of the exports object as a standalone object literally changed the use case of the module object.

CommonJS Modules (cont’d)

- In Node's **CommonJS** module implementation, each module is written in a single JavaScript file and has an isolated scope that holds its own variables.
- The author of the module can expose any functionality through the **exports** object.
- To understand it better, let's say we created a module file named **`hello.js`** that contains the following code snippet:

```
let message = 'Hello';
exports.sayHello = function(){
   console.log(message);
}
```

CommonJS Modules (cont’d)

- Also, let's say we created an application file named **`server.js`**, which contains the following lines of code:

```
let hello = require('./hello');
hello.sayHello();
```

- In the preceding example, you have the **`hello`** module, which contains a variable named **`message`**.
- The **`message`** variable is self-contained in the **`hello`** module, which only exposes the **`sayHello()`** method by defining it as a property of the **`exports`** object.
- Then, the application file loads the hello module using the **`require()`** method, which will allow it to call the **`sayHello()`** method of the **`hello`** module.

Node.js Core Modules

- **Core modules** are modules that were compiled into the Node binary.
- They come **prebundled** with **Node** and are documented in great detail in its documentation.
- The core modules provide most of the basic functionalities of Node, including **filesystem** access, **HTTP** and **HTTPS** interfaces, and much more.
- To load a core module, you just need to use the **require** method in your JavaScript file.

Node.js Core Modules (cont’d)

- An example code, using the **`fs`** core module to read the content of the environment hosts file, would look like the following code snippet:

```
const fs = require('fs');
fs.readFile('/etc/hosts', 'utf8', (err, data) => {
   if (err) {
      return console.log(err);
   }
   console.log(data);
});
```

- When you require the **`fs`** module, Node will find it in the core modules folder.
- You'll then be able to use the **`fs.readFile()`** method to read the file's content and print it in the command-line output.

Developing Node.js web applications

- **Node.js** is a platform that supports various types of applications, but the most popular kind is the development of **web applications**.
- Node's style of coding depends on the community to extend the platform through third-party modules; these modules are then built upon to create new modules, and so on.
- Companies and single developers around the globe are participating in this process by creating modules that wrap the basic Node APIs and deliver a better starting point for application development.