COMP3123 - Full Stack Development I

What is Node js ?

- Node.js is a **server-side** platform built on Google Chrome's JavaScript Engine (V8 Engine).
- Node.js was developed by Ryan Dahl in 2009 and its latest version is v0.10.36.
- The definition of Node.js as supplied by its official documentation is as follows −
- Node.js is
  - ***an open source***
  - ***cross-platform runtime environment***
  - ***for developing server-side***
  - ***and networking applications***
- Node.js applications are written in **JavaScript**, and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.
- Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

Web Reference Link

Node JS : Environment Setup

- Download latest version of Node.js installable archive file from Node.js Downloads. At the time of writing this tutorial, following are the versions available on different OS.

Verify installation: Executing a File

- Create a JavaScript file named **main.js** on your machine (Windows, MacOs or Linux) having the following code.

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

Now execute main.js file using Node.js interpreter to see the result −

```
$ node main.js
```

If everything is fine with your installation, this should produce the following result −

```
Hello, World!
```

Web Reference Link

Node Package Manager (NPM)

- **Node Package Manager (NPM)** provides two main functionalities −
  - Online repositories for node.js packages/modules which are searchable on search.nodejs.org
  - Command line **utility to install Node.js packages**, do version management and dependency management of Node.js packages.

Installing Modules using NPM

- There is a simple syntax to install any Node.js module −

  $ npm install <Module Name>

- For example, following is the command to install a famous Node.js web framework module called express −

  $ npm install express --save

- Now you can use this module in your .js file as following −

  var express = require('express');

Web Reference Link

Semantic Versioning using npm

When you make a new release, you don't just up a number as you please, but you have rules:

- *you up the major version when you make incompatible API changes*
- *you up the minor version when you add functionality in a backward-compatible manner*
- *you up the patch version when you make backward-compatible bug fixes*

Why is that so important?

Because npm set some rules we can use in the **package.json** file to choose which versions it can update our packages to, when we run npm update.

The Semantic Versioning concept is simple: all versions have 3 digits: **x.y.z. for e.g. 1.1.3**

- *the first digit is the major version*
- *the second digit is the minor version*
- *the third digit is the patch version*

The rules use those symbols: **^ ~ > >= < <= = - ||**

Web Reference Link

Semantic Versioning cont..

Let's see those rules in detail:

^	if you write ^0.13.0 when running npm update it can update to patch and minor releases: 0.13.1, 0.14.0 and so on.
~ patch	if you write ~0.13.0, when running npm update it can update to releases: 0.13.1 is ok, but 0.14.0 is not.
>	you accept any version higher than the one you specify
>=	you accept any version equal to or higher than the one you specify
<=	you accept any version equal or lower to the one you specify
<	you accept any version lower to the one you specify
=	you accept that exact version
-	you accept a range of versions. Example: 2.1.0 - 2.6.2
||	you combine sets. Example: < 2.1 || > 2.6

Web Reference Link

What is the file `package.json`?

All npm packages contain a file, usually in the project root, called **package.json**

- this file holds various metadata relevant to the project.
- This file is used to give information to npm that allows it to identify the project as well as handle the project's dependencies.
- It can also contain other metadata such as a project description, the version of the project in a particular distribution, license information, even configuration data –
- all of which can be vital to both npm and to the end users of the package.

The package.json file is normally located at the root directory of a Node.js project.

Node.js itself is only aware of two fields in the package.json:

```
{
  "name" : ”node-sample-app",
  "version" : ”1.0.0",
 }
```

Web Reference Link

Example - package.json?

```
{
  "name" : "underscore",
  "description" : "JavaScript's functional programming helper library.",
  "homepage" : "http://documentcloud.github.com/underscore/",
  "keywords" : ["util", "functional", "server", "client", "browser"],
  "author" : "Jeremy Ashkenas <jeremy@documentcloud.org>",
  "contributors" : [],
  "dependencies" : [],
  "repository" : {"type": "git", "url": "git://github.com/documentcloud/underscore.git"},
  "main" : "underscore.js",
  "version" : "1.1.6"
}
```

Web Reference Link

Node JS : Global Object

- Node.js global objects are global in nature and they are available in all modules.
- We do not need to include these objects in our application, rather we can use them directly. **These objects are modules, functions, strings and object**
- The **__filename** represents the filename of the code being executed. This is the resolved absolute path of this code file. For a main program, this is not necessarily the same filename used in the command line. The value inside a module is the path to that module file.
- The **__dirname** represents the name of the directory that the currently executing script resides in.
- The **setTimeout(cb, ms)** global function is used to run callback cb after at least ms milliseconds. The actual delay depends on external factors like OS timer granularity and system load. This function returns an opaque value that represents the timer which can be used to clear the timer.
- The **clearTimeout(t)** global function is used to stop a timer that was previously created with setTimeout(). Here **t** is the timer returned by the setTimeout() function.

- **Console** - Used to print information on stdout and stderr.
- **Process** - Used to get information on current process.
  Provides multiple events related to process activities.

Web Reference Link

Node.js Module

- Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.
- Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope. Also, each module can be placed in a separate .js file under a separate folder.
- Node.js implements CommonJS modules standard. CommonJS is a group of volunteers who define JavaScript standards for web server, desktop, and console application.

Node.js includes three types of modules:

- Core Modules
- Local Modules
- Third Party Modules

- Node.js is a light weight framework. The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts. However, you need to import the core module first in order to use it in your application.

The following table lists some of the important core modules in Node.js.

Core Module	Description
http	http module includes classes, methods and events to create Node.js http server.
url	url module includes methods for URL resolution and parsing.
querystring	querystring module includes methods to deal with query string.
path	path module includes methods to deal with file paths.
fs	fs module includes classes, methods, and events to work with file I/O.
util	Provides utility functions useful for programmers.
OS Module	Provides basic operating-system related utility functions.
Net Module	Provides both servers and clients as streams. Acts as a network wrapper.
Domain Module	Provides ways to handle multiple different I/O operations as a single group.

Export Module in Node.js

The module.exports is a special object which is included in every JavaScript file in the Node.js application by default.

The module is a variable that represents the current module, and exports is an object that will be exposed as a module. ***So, whatever you assign to module.exports will be exposed as a module.***

Let's see how to expose different types as a module using module.exports.

Export Literals

- As mentioned above, exports is an object. So it exposes whatever you assigned to it as a module. For example, if you assign a string literal then it will expose that string literal as a module.
- The following example exposes simple string message as a module in **Message.js**.

```
module.exports = 'Hello world';
```

Now, import this message module and use it as shown below.

```
var msg = require('./Messages.js');
console.log(msg);
```

Web Reference Link

Require in Node JS

Node.js follows the CommonJS module system, and the built-in 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. An example module:

```
console.log("evaluating example.js");

var invisible = function () {

console.log("invisible");

}

exports.message = "hi";

exports.say = function () {

console.log(exports.message);

}
```

So if you run **var example = require('./example.js'),** then example.js will get evaluated and then example be an object equal to:

```
{
  message: "hi",
  say: [Function]
}
```

Web Reference

Introduction to Events and Event Emitter

In Node many objects emit events,

For e.g. a net.Server emits an event each time a peer connects to it, an fs.readStream emits an event when the file is opened.

- **All objects which emit events are the instances of events.EventEmitter.**
- Node.js has a built-in module, called "Events", where you can **create-, fire-, and listen** for- your own events.
- To include the built-in Events module use the require() method.
- In addition, all event properties and methods are an instance of an EventEmitter object.

To be able to access these properties and methods, create an EventEmitter object:

```
var events = require('events');
var eventEmitter = new events.EventEmitter();
```

Web reference Link

Introduction to Event Emitter

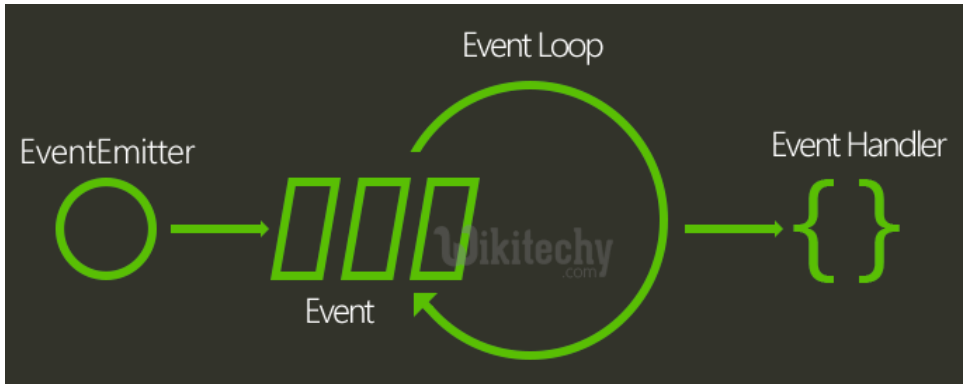
The EventEmitter Object

- You can assign event handlers to your own events with the **EventEmitter** object.
- In the example below we have created a function that will be executed when a "scream" event is fired.
- To fire an event, use the emit() method.

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

//Create an event handler:
var myEventHandler = function () {
 console.log('I hear a scream!');
}
//Assign the eventhandler to an event:
eventEmitter.on('scream', myEventHandler);

//Fire the 'scream' event:
eventEmitter.emit('scream');
```



Web reference Link

Node as a Web Server

- The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.
- Use the createServer() method to create an HTTP server:

```
var http = require('http');
//create a server object:
http.createServer(function (req, res) {
res.write('Hello World!'); //write a response to the client
res.end(); //end the response
}).listen(8080); //the server object listens on port 8080
```

- The function passed into the http.createServer() method, will be executed when someone tries to access the computer on port 8080.
- Save the code above in a file called "demo_http.js", and initiate the file:
- Initiate demo_http.js:

```
node demo_http.js
```

Web reference Link

Add an HTTP Header

- If the response from the HTTP server is supposed to be displayed as HTML, you should include an HTTP header with the correct content type:

```
var http = require('http');
http.createServer(function (req, res) {
 // add a HTTP header:
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('Hello World!');
  res.end();
}).listen(8080);
```

Read the Query String

The function passed into the http.createServer() has a req argument that represents the request from the client, as an object (http.IncomingMessage object).

This object has a property called "url" which holds the part of the url that comes after the domain name:

demo_http_url.js

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write(req.url);
  res.end();
}).listen(8080);
```

Node.js File System Module

- The Node.js file system module allows you to work with the file system on your computer.
- To include the File System module, use the require() method:

var fs = require('fs');

Common use for the File System module:

- Read files
- Create files
- Update files
- Delete files
- Rename files

- The ***fs.readFile()*** method is used to read files on your computer.
- The File System module has methods for creating new files:

fs.appendFile()

fs.open()

fs.writeFile()

- The fs.appendFile() method appends specified content to a file. If the file does not exist, the file will be created:

Node.js File System Module

- The File System module has methods for updating files:

  fs.appendFile()

  fs.writeFile()

- The fs.appendFile() method appends the specified content at the end of the specified file.
- To delete a file with the File System module, use the ***fs.unlink()*** method.
- To rename a file with the File System module, use the ***fs.rename()*** method.

DEMO

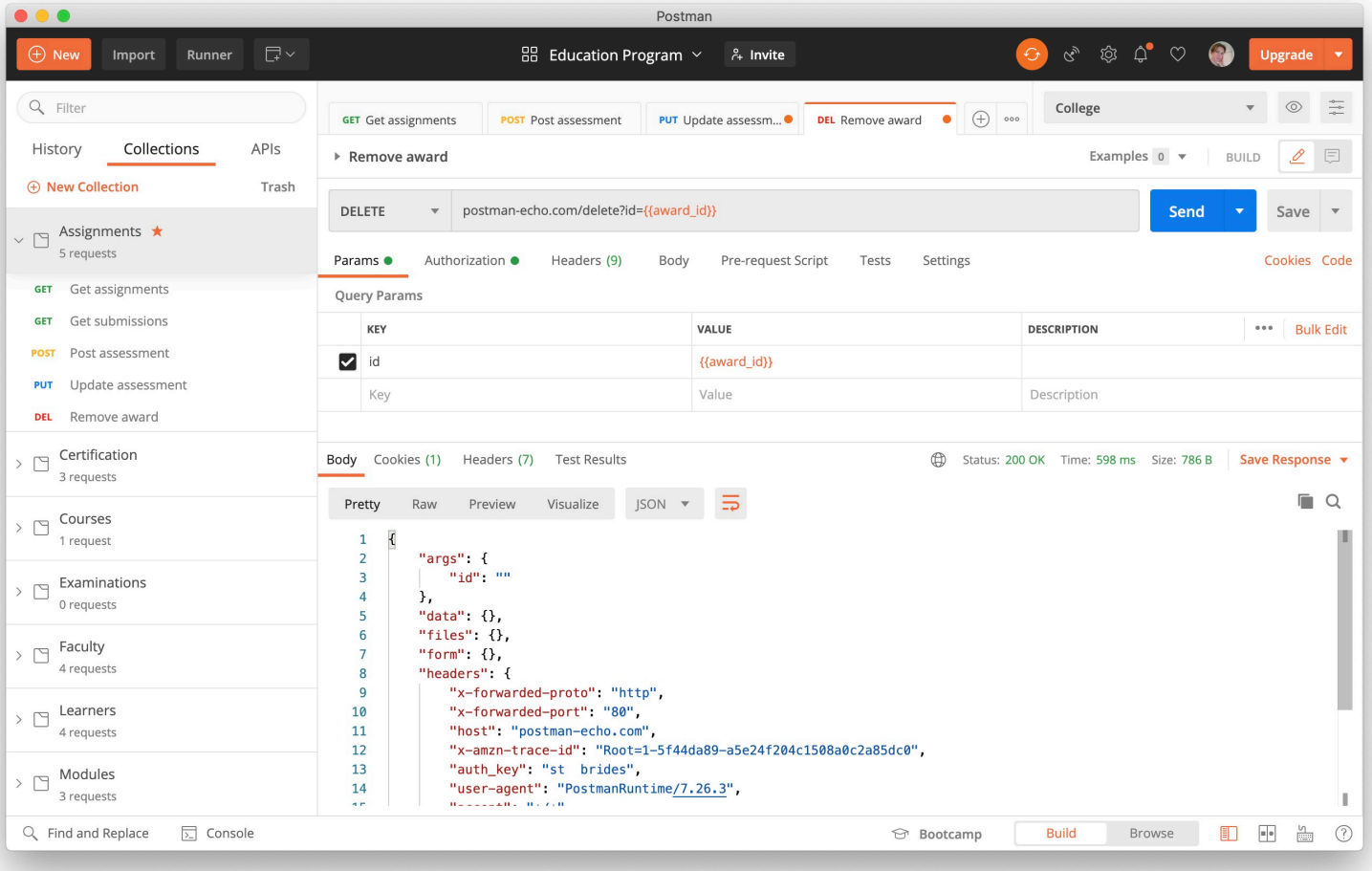
Web Reference Link

Installing Postman

https://www.postman.com/downloads/

- **Postman** is a scalable API testing tool that quickly integrates into CI/CD pipeline.
- It started in 2012 as a side project by Abhinav Asthana to simplify API workflow in testing and development.

- ***What is API ?***
  - *API stands for Application Programming Interface which allows software applications to communicate with each other via API calls.*



Web Reference Link

Thank You