Microservices with Node.js

Objectives

In this module we will discuss:

- Core Node.js concepts
- Node Package Manager (NPM)
- The Express Node.js package
- The MEAN stack

1.1 What is Node.js?

- Node.js® [*https://nodejs.org/*] is a JavaScript runtime built on Chrome's V8 JavaScript engine -- it runs on a wide range of the supported OSes (*ix, OS X and Windows) as a native executable process
- Node.js (or, simply, Node) is used for hosting JavaScript applications running natively (not in a browser-like sandboxed environment) on the target OS
- It uses a single-threaded task dispatcher approach to concurrency
  - There is a lot of system design similarities with the Windows® message loop implemented in the *WinMain* function: *(https://en.wikipedia.org/wiki/Message_loop_in_Microsoft_Windows)*
- For cross-platform asynchronous (non-blocking) I/O operations (file and network), Node.js uses the **libuv** *(http://libuv.org/)* library
- It comes with its own package manager, Node Package Manager (NPM), which manages the full life cycle of Node packages (libraries) and their dependencies
- Node is an excellent platform for building microservices

Notes:

The V8 JavaScript engine, written in C++, also has a built-in Just in Time compiler that translates JavaScript code into high-performance code. V8 can run as a standalone application, or can be embedded into any C++ application. It also supports multiple processor architectures: x86/64, ARM, MIPS, etc.

1.2 Node's Value Proposition

- Node.js is primarily used for rapid application development of web applications and web services
- Node.js is also used as a platform for creating command-line tools
  - Many of the popular build, module dependency management, and testing tools, like Gulp, Grunt, Jasmine, at al, are written as Node.js applications
  - Node.js is also used as the runtime for MongoDB *(https://www.mongodb.com/)* JavaScript-based Admin command console

1.3 Example of a Node.js App: a Simple Web Server

- The code below is all you need to create a very simple (and not very practical) Node app that can be used as a simple web server that accepts an HTTP client request and sends back a *Hello, Node!* response
- The code is contained in a JavaScript file that we called *hello.js*

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

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello, Node!");
  response.end();
}).listen(8088);
```

- You run the app on your "server" machine like this:

```
node  hello.js
```

- You can access the app by either using your browser or the *curl* tool:

```
curl http://<IP address of your Node 'Web Server' app>:8088
```

Notes:

The above code does the following:

The first line imports the *http* Node core module.

Next, we create an HTTP listening end-point on port 8088 and define a callback function that is invoked by the Node event loop process to handle incoming HTTP requests.

In this simple example, we are handling all requests in the same way -- just sending the same "Hello, Node!" response as plain text.

We also do not make any differentiation for HTTP methods or URLs of the incoming client requests.

1.4 Node.js Project Types

- Most Node.js projects fall into two categories:
  - **Executable application** – This code is executed using the *node* command tool
  - **Reusable library** – This code is used by other reusable libraries and/or executable applications
- A simple executable application can be written as a single JavaScript file and executed as follows:

```
node file_containing_your_code.js
```

1.5 Managing Large Applications

- As your Node application grows bigger and bigger, you start facing problems:
  - The larger your application written as a single file becomes, the harder it gets to manage
  - You need to properly encapsulate functionality of your *Node* libraries
    - Make sure to expose only the functions and data relevant to the user of the library and hide the rest !
- Node.js offers a simple and effective module structure to solve these problems

1.6 Core Modules

- Node.js comes with a set of core modules that are pre-compiled into binary form for fast loading and execution:
  - **http / https –** Communicate with a web server
  - **fs** – Read and write files (native file system IO) with asynchronous support
  - **crypto** - Cryptographic functionality support
  - **buffer** – Work with large chunks of data efficiently
- You load a core module by its name:

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

- For the list of core modules of the latest Node.js version, visit: *https://nodejs.org/api/modules.html#modules_core_modules*
- For older Node.js versions' documentation, visit: *https://nodejs.org/docs/*

1.7 Why Node.js uses JavaScript?

- JavaScript is the most widely used programming language in the world !
- It is the *lingua franca* language of the client-side web development
- JavaScript Object Notation (JSON) is the common data interchange format for JavaScript and now, increasingly, in other programming and storage systems as well
- Non-blocking programming model implemented via asynchronous communication patterns and callback functions is common in JavaScript

1.8 The Traditional Concurrency Support Model

- Most web servers, including Java-based application servers, use a multi-threaded concurrency model where
  - The server maintains a pool of threads
  - Each client request is handled in a separate thread from the pool
  - The thread executes (and blocks) until the request is handled
  - After the client request has been processed, the thread is returned to the pool
    - In some cases of unreleased thread local storage structures, the thread pool gets "dirty" threads that contain previous client request's data; this fact poses a number of security and data integrity threats
- Traditional Server-Side I/O (Java, C#, et. al.)
  - Synchronous
  - Multi-Threaded
  - Blocking

1.9 Disadvantages of the Traditional Approach

- Correctly coding multi-threaded apps is complicated
- Poorly implemented thread-management code may lead to *race* or *deadlock* situations
- Client request threads block (wait) for server responses
- Even though blocked threads are put to sleep by the OS scheduler, they still retain useful OS resources, e.g. stack memory

1.10 Event-Driven, Non-Blocking I/O

- By contrast, Node.js uses the event loop mechanism for handling concurrency
  - The event loop process runs in a single thread
  - Client requests are handled in different and isolated from each other threads dispatched by the event loop thread
    - This is a good example of clean concurrency management model
    - It dramatically simplifies the programming model
    - Problems related to deadlocks or race conditions simply disappear (unless you have those threads concurrently accessing the same system resource, e.g. a file)

1.11 The Success Callback Function

- As a Node app developer, you need to specify a *success* callback function that implements your request handling logic and which will be called by Node

- This programming model mirrors the model used in the client-side JavaScript running in the browsers
- For example, the popular jQuery library uses the following programmatic idiom to make an AJAX GET call and process the server response in the anonymous callback function which receives the server response (usually, as a JSON-encoded document) as its input parameter

```
$.get('/YYZ/arrivals/KL691', function( flightData ) {
   // handle flight data here
   ....
});
```

Notes

Calling a function and specifying a callback is a common JavaScript idiom,. One of the things that Java and C# developers often find challenging about this is that the callback function is often an *anonymous* function defined right in the call.

1.12 Using Node Package Manager (NPM)

- Node Package Manager (*NPM*) can be installed separately from Node.js, but usually they are installed together
- NPM functionality is supported by the **npm** tool
- You can run *npm* using this syntax:

```
npm <command> <options>
```

- Basic **npm** commands:
  - **help** – list available commands
  - **install** – install a new package(s) by getting it (them) from a repository
  - **uninstall** – delete the installed package(s)

  - **ls** – list all installed packages
  - **update** – update packages
  - **link** – manage dependencies
  - **publish** – make your packages available publicly for reuse

1.13 NPM Registry (Repository)

- NPM central repository [*https://www.npmjs.com/*] hosts over a quarter million packages of reusable code making it the largest code repository in the world
- The registry supports the following operations:
  - Find
  - Discover
  - Build

1.14 NPM Enterprise

- NPM Enterprise [*https://www.npmjs.com/enterprise*] is, in essence, NPM for enterprise use cases
- Deployed behind the corporate firewall
- Delegates login to your existing single sign-on providers: LDAP, GitHub Enterprise, Atlassian, etc.
- Mirrors the select domains of the public NPM registry and maintains private scopes
- Operate completely in isolation from the public Internet

1.15 Package Life-Cycle Management

- Each unit of distribution in NPM is called a **package**. A package is basically a Node.js module with a **package.json** descriptor file
- NPM manages the full package life-cycle phases:
  - Installation
  - Updating
  - Uninstalling (deleting)

1.16 Local and Global Package Installation Options

- You can install packages in two ways:
  - **Locally** to a module – only that module will be able to use the installed package (you need to run this command from the module directory):
    - ✔ `npm install <package name>`
    - ✔ **Note**: The above command will install the latest version of the package
  - **Globally** – Any module in the machine will be able to use the downloaded packages
    - ✔ Just add the **-g** flag to the *npm install* command
- Generally, modules that are reusable code libraries are installed locally; modules that have executable utilities are installed globally

1.17 Listing and Using Module Versions

- To view the available module versions, use the following *npm* command:

```
npm view <module name> versions
```

- To install a specific version of the module, use the following *npm* command:
  - To install the module globally:

```
npm install -g <module name>@<the version you need>
```

  - To install the module locally to your module:

```
npm install <module name>@<the version you need>
```

1.18 The Express Package

- Express *(http://expressjs.com/)* is a web framework built around Node's *http* core module
- Very useful for rapid provisioning of RESTful web services
- Provides features needed to build large web applications, including:
  - Routing
  - Templating
  - Static file serving
  - Ability to plug in various middleware modules, like caching, security, and logging

1.19 Installing and Using Express

- To globally install the latest version Express module:

```
npm install -g express
```

- Using Express in your Node apps:

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

1.20 Defining Routing Rules in Express

- Each route is a unique combination of HTTP method and path
- You define a new route using a method of the Express application object appropriate for the HTTP method. Example:
  - **get()** - For GET method
  - **put()** - For PUT method
  - **post()** - For POST method, and so on
- These route definition methods take a path as the first argument and a request (success) handler function as the second argument

```
app.post('/customer', function(req, res){});
```

- A request handler function is called when a request matches the HTTP method and path of its route. It takes two arguments:
  - **Request** – Same as the request object defined in the *http* module
  - **Response** – An extension of the response object defined in the *http* module
- Routing rules is a very important capability used in building RESTful endpoints

1.21 Route Path

- Path is provided as the first argument to a route definition method (*get(), post()* etc). It can be a string or a regular expression
- Fully qualified path:

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

- Regular expression in a string. The following matches "/planets/earth" and "/planets/saturn":

```
app.get('/planet/*', function(req, res){});
```

- Proper regular expression. Following matches "/BMW.car", "/FORD.car" but not "/BMW.cars" or "/BMW.car/specials"

```
app.get('/.*car$/', function(req, res){});
```

1.22 The Response Object

- In addition to the methods in the response object as defined in the *http* module, the following methods are added. All of them terminate the response document and no more data can be sent
  - **send(data)** – Sends a string, a buffer or an object. If you supply an object, it will be converted to JSON and the *Content-type* header will be set to "*application/json; charset=utf-8*". The response document is marked as completed after a call to send()
  - **redirect(path)** – Responds with an HTTP 302 redirect code

1.23 A Simple Web Service with Express Example

```
var express = require("express");
var app = express();
var planets = [
    {name: "earth", hoursInDay: 24},
    {name: "jupiter", hoursInDay: 9.9}];

app.get('/planets', function (req, res) {
  res.send(planets);
})
.get('/planets/:name', function (req, res) {
  var result = planets.filter(function(p) {
    return p.name === req.params.name;
  });

  if (result.length === 0) {
      res.statusCode = 404;
      res.end();
  } else {
      res.send(result[0]);
  }
})
.listen(3000);
```

1.24 The MEAN Stack

- The M.E.A.N. (*MEAN*, for short) technology stack (*http://mean.io/*) is comprised of (and gets its name from) the following technologies:
  - **M**ongoDB, **E**xpress.js, **A**ngularJS, and **N**ode.js
- which are mostly open-source JavaScript-based technologies
  - MongoDB *(https://www.mongodb.com/)* is written, similar to Node.js, in C++; however, it also provides JavaScript-based Admin CLI
- MEAN is used for building dynamic web sites and microservices

- MongoDB, Node, and Express are used for building the server back-end; Angular is used for creating the front-end to the server
- You can install the MEAN stack by running this *npm* command:

```
npm install -g mean-cli
```

1.25 Summary

- In this module we discussed Node.js concepts
  - Core Node concepts
  - Event-Driven I/O
  - Concurrency
  - Simple HTTP server with Node.js
- We also covered the basics of the Node Package Manager (NPM) and the Express package
- We provided a quick overview of the MEAN stack