COMP2068 – JavaScript Frameworks

Lesson 2
Managing Packages with NPM

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:

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:

```
var 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:

```
var 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:

```
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.

Developing Node.js web applications (cont'd)

- There are many modules to support web application development but none as popular as the **Connect** module.
- The **Connect** module delivers a set of wrappers around the **Node.js** low-level APIs to enable the development of rich web application frameworks.

Developing Node.js web applications (cont'd)

- To understand what Connect is all about, let's begin with a basic example of a basic Node web server.
- In your working folder, create a file named **`server.js`**, which contains the following code snippet:

```
var http = require('http');
http.createServer((req, res) => {
   res.writeHead(200, {
   'Content-Type': 'text/plain'
   });
   res.end('Hello World');
}).listen(3000);
console.log('Server running at http://localhost:3000/');
```

Developing Node.js web applications (cont'd)

- To start your web server, use your command-line tool, and navigate to your working folder.
- Then, run the node CLI tool and run the server.js file as follows:

```
$ node server
```

- Now open **`http://localhost:3000`** in your browser, and you'll see the **Hello World** response.

Developing Node.js web applications (cont'd)

So how does this work?

- In this example, the **`http`** module is used to create a small **web server** listening to the 3000 port.
- You begin by requiring the **`http`** module and use the **`createServer()`** method to return a new server object.
- The **`listen()`** method is then used to listen to the 3000 port.
- Notice the callback function (anonymous function) that is passed as an argument to the **`createServer()`** method.

Developing Node.js web applications (cont'd)

- The callback function gets called whenever there's an **HTTP request** sent to the web server.
- The server object will then pass the **`req`** and **`res`** arguments, which contain the information and functionality needed to send back an HTTP response.
- The callback function will then do the following two steps:
    1. First, it will call the **`writeHead()`** method of the response object. This method is used to set the **response HTTP headers**. In this example, it will set the **Content-Type** header value to **text/plain**. For instance, when responding with HTML, you just need to replace **text/plain** with **html/plain**.
    2. Then, it will call the **`end()`** method of the response object. This method is used to finalize the response. The **`end()`** method takes a single string argument that it will use as the **HTTP response body**. Another common way of writing this is to add a **`write()`** method before the **`end()`** method and then call the **`end()`** method, as follows:

```
res.write('Hello World');
res.end();
```

Meet the Connect module

- **Connect** is a module built to support interception of requests in a more modular approach.
- In the first web server example, you learned how to build a simple web server using the **`http`** module.
- If you wish to extend this example, you'd have to write code that manages the different HTTP requests sent to your server, handles them properly, and responds to each request with the correct response.

Meet the Connect module (cont'd)

- **Connect** creates an API exactly for that purpose – it uses a modular component called **middleware**, which allows you to simply register your application logic to **predefined** HTTP request scenarios.
- Connect **middleware** are basically callback functions, which get executed when an HTTP request occurs.
- The **middleware** can then **perform some logic**, **return a response**, or call the next registered middleware.
- While you will mostly write **custom middleware** to support your application needs, Connect also includes some common middleware to support **logging**, **static file serving**, and more.

Meet the Connect module (cont'd)

- The way a Connect application works is by using an object called **`dispatcher`**.
- The **`dispatcher`** object handles each HTTP request received by the server and then decides, in a cascading way, the order of middleware execution.

Meet the Connect module (cont’d)

- In the next section, you'll create your first **Express** application, but **Express** is based on **Connect's** approach, so in order to understand how **Express** works, we'll begin with creating a **Connect** application.
- In your working folder, create a file named **`server.js`** that contains the following code snippet:

```
var connect = require('connect');
var app = connect();
app.listen(3000);
console.log('Server running at
http://localhost:3000/');
```

Meet the Connect module (cont'd)

- As you can see, your application file is using the **`connect`** module to create a new **web server**.
- However, **Connect** isn't a core module, so you'll have to install it using **NPM**.
- To do so, use your command-line tool, and navigate to your working folder. Then execute the following command:

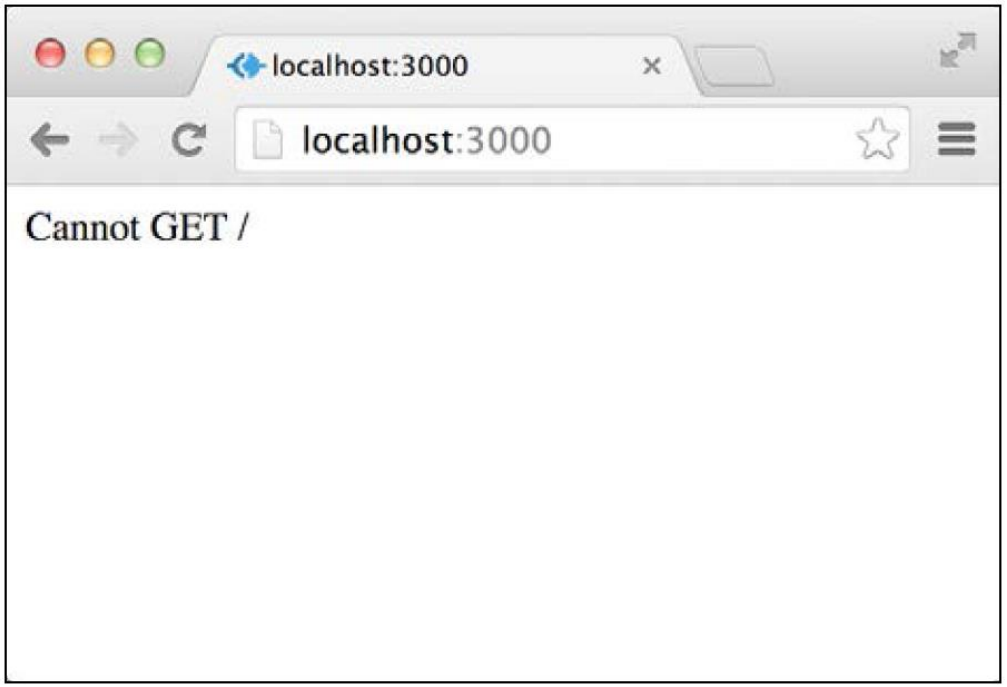
```
$ npm i --save connect
```

- **NPM** will install the connect module inside a **node_modules** folder, which will enable you to **require** it in your application file.
- To run your Connect web server, just use Node's CLI and execute the following command:

```
$ node server
```

Meet the Connect module (cont'd)

- Node will run your application, reporting the server status using the **`console.log()`** method.
- You can try reaching your application in the browser by visiting **`http:// localhost:3000`**.
- However, you should get a response similar to what is shown in the following screenshot:



- What this response means is that there isn't any **middleware** registered to handle the **GET HTTP** request.

Connect Middleware

- Connect **middleware** is just **JavaScript function** with a unique signature.
- Each middleware function is defined with the following three arguments:
  - **req:** This is an object that holds the HTTP **request** information
  - **res:** This is an object that holds the HTTP **response** information and allows you to set the response properties
  - **next:** This is the next middleware function defined in the ordered set of Connect middleware

Connect Middleware (cont'd)

- When you have a **middleware** defined, you'll just have to register it with the **Connect** application using the **`app.use()`** method.
- Let's revise the previous example to include your first **middleware**.
- Change your **`server.js`** file to look like the following code snippet:

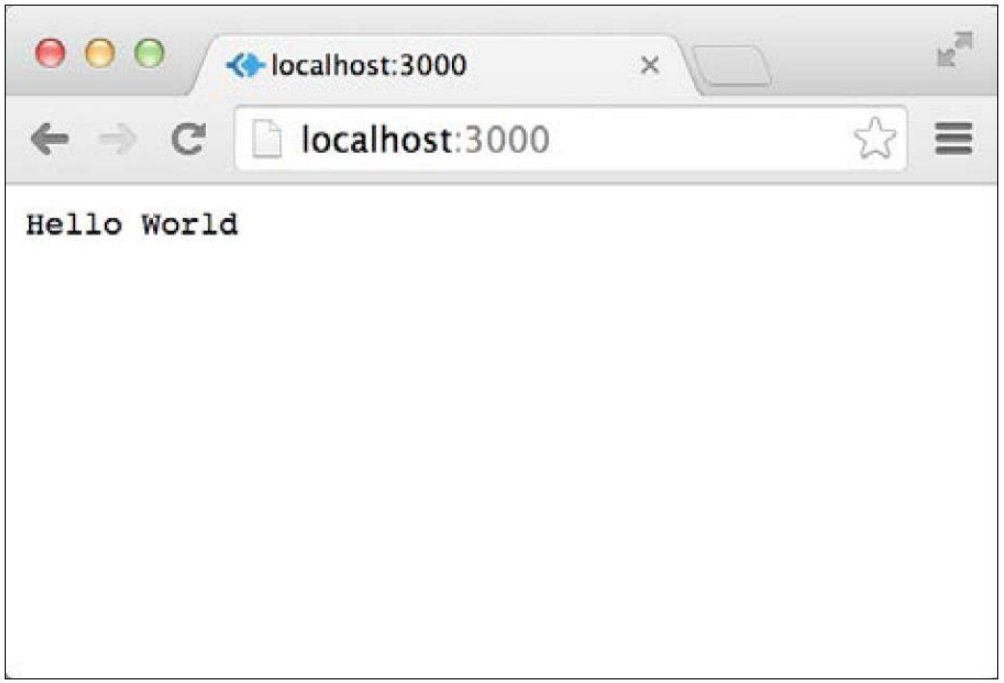
```
var connect = require('connect');
var app = connect();
var helloWorld = (req, res, next) => {
   res.setHeader('Content-Type', 'text/plain');
   res.end('Hello World');
};
app.use(helloWorld);
app.listen(3000);
console.log('Server running at http://localhost:3000/');
```

Connect Middleware (cont'd)

- Then, start your **connect** server again by issuing the following command in your command-line tool:

```
$ node server
```

- Try visiting **`http://localhost:3000`** again.
- You will now get a response similar to that in the following screenshot:



Mounting Connect middleware

- As you may have noticed, the **middleware** you registered responds to **any request** regardless of the request path.
- This does not comply with modern web application development because responding to different paths is an integral part of all web applications.
- Fortunately, Connect middleware supports a feature called **mounting**, which enables you to determine which request path is required for the middleware function to get executed.
- **Mounting** is done by adding the **`path`** argument to the **`app.use()`** method.
- To understand this better, let's revisit our previous example.

Mounting Connect middleware (cont'd)

- To understand this better, let's revisit our previous example -- modify your **`server.js`** file to look like the following code snippet:

```
var connect = require('connect');
var app = connect();
var logger = (req, res, next) => {
   console.log(req.method, req.url);
   next();
};
var helloWorld = (req, res, next) => {
   res.setHeader('Content-Type', 'text/plain');
   res.end('Hello World');
};
var goodbyeWorld = (req, res, next) => {
   res.setHeader('Content-Type', 'text/plain');
   res.end('Goodbye World');
};
app.use(logger);
app.use('/hello', helloWorld);
app.use('/goodbye', goodbyeWorld);
app.listen(3000);
console.log('Server running at http://localhost:3000/');
```

Intro to Express.js

Building an Express Web Application

- The **Express framework** is a small set of common web application features, kept to a minimum in order to maintain the **Node.js** style.
- It is built on top of **Connect** and makes use of its middleware architecture.
- Its features extend **Connect** to allow a variety of common web applications' use cases, such as the inclusion of **modular HTML template engines**, extending the response object to support various data format outputs, a routing system, and much more.

Creating your first Express application

- After creating your **`package.json`** file and installing express as one of your dependencies, you can now create your first **Express** application by adding your already familiar **`server.js`** file with the following lines of code:

```
var express = require('express');
var app = express();
app.use('/', function(req, res) {
   res.send('Hello World');
});
app.listen(3000);
console.log('Server running at http://localhost:3000/');
module.exports = app;
```

The application, request, and response objects

- **Express** presents three major objects that you'll frequently use.
  - The **application object** is the instance of an Express application you created in the first example and is usually used to configure your application.
  - The **request object** is a wrapper of Node's HTTP request object and is used to extract information about the currently handled HTTP request.
  - The **response object** is a wrapper of Node's HTTP response object and is used to set the response data and headers.

The application, request, and response objects (cont'd)

The application object

- The application object contains the following methods to help you configure your application:
  - **`app.set(name, value)`**: This is used to set environment variables that Express will use in its configuration.
  - **`app.get(name):`** This is used to get environment variables that Express is using in its configuration.
  - **`app.engine(ext, callback):`** This is used to define a given template engine to render certain file types, for example, you can tell the **EJS template engine** to use HTML files as templates like this: **`app.engine('html', require('ejs').renderFile).`**
  - **`app.locals:`** This is used to send application-level variables to all rendered templates.

The application, request, and response objects (cont'd)

The application object (cont'd)

- **`app.use([path], callback):`** This is used to create an Express middleware to handle HTTP requests sent to the server. Optionally, you'll be able to mount middleware to respond to certain paths.
- **`app.VERB(path, [callback...], callback):`** This is used to define one or more middleware functions to respond to HTTP requests made to a certain path in conjunction with the HTTP verb declared. For instance, when you want to respond to requests that are using the GET verb, then you can just assign the middleware using the app.get() method. For POST requests you'll use **`app.post()`**, and so on.
- **`app.route(path).VERB([callback...], callback):`** This is used to define one or more middleware functions to respond to HTTP requests made to a certain unified path in conjunction with multiple HTTP verbs. For instance, when you want to respond to requests that are using the GET and POST verbs, you can just assign the appropriate middleware functions using **`app.route(path).get(callback).post(callback)`**.
- **`app.param([name], callback):`** This is used to attach a certain functionality to any request made to a path that includes a certain routing parameter. For instance, you can map logic to any request that includes the userId parameter using **`app.param('userId', callback)`**.