Lesson 2: Module System, Timers, Exports object

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 Module Types

Node.js includes three types of modules:

1. Core Modules
2. Local Modules
3. Third Party Modules

Node.js Core 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	util module includes utility functions useful for programmers.

Loading Core Modules

In order to use Node.js core or NPM modules, you first need to import it using require() function as shown below.

```
var module = require('module_name');
```

As per above syntax, specify the module name in the require() function. The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.

The following example demonstrates how to use Node.js http module to create a web server.

Example: Load and Use Core http Module

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

var server = http.createServer(function(req, res){

  //write code here

});

server.listen(5000);
```

In the above example, require() function returns an object because http module returns its functionality as an object, you can then use its properties and methods using dot notation e.g. http.createServer().

Node.js Local Module

Local modules are modules created locally in your Node.js application. These modules include different functionalities of your application in separate files and folders. You can also package it and distribute it via NPM, so that Node.js community can use it. For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

Writing Simple Module

Let's write simple logging module which logs the information, warning or error to the console.

In Node.js, module should be placed in a separate JavaScript file. So, create a Log.js file and write the following code in it.

Log.js

```
var log = {
        info: function (info) {
           console.log('Info: ' + info);
        },
        warning:function (warning) {
           console.log('Warning: ' + warning);
        },
        error:function (error) {
           console.log('Error: ' + error);
        }
   };

module.exports = log
```

In the above example of logging module, we have created an object with three functions - info(), warning() and error(). At the end, we have assigned this object to **module.exports**. The module.exports in the above example exposes a log object as a module.

The *module.exports* is a special object which is included in every JS file in the Node.js application by default. Use **module.exports** or **exports** to expose a function, object or variable as a module in Node.js.

Now, let's see how to use the above logging module in our application.

Loading Local Module

To use local modules in your application, you need to load it using require() function in the same way as core module. However, you need to specify the path of JavaScript file of the module.

The following example demonstrates how to use the above logging module contained in Log.js.

app.js

```
var myLogModule = require('./Log.js');

myLogModule.info('Node.js started');
```

In the above example, app.js is using log module. First, it loads the logging module using require() function and specified path where logging module is stored. Logging module is contained in Log.js file in the root folder. So, we have specified the path './Log.js' in the require() function. The '.' denotes a root folder.

The require() function returns a log object because logging module exposes an object in Log.js using module.exports. So now you can use logging module as an object and call any of its function using dot notation e.g myLogModule.info() or myLogModule.warning() or myLogModule.error()

Thus, you can create a local module using module.exports and use it in your application.

Export Module in Node.js

How to expose different types as a module using module.exports?

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.

Message.js

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

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

app.js

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

console.log(msg);
```

You must specify ./ as a path of root folder to import a local module. However, you do not need to specify the path to import Node.js core modules or NPM modules in the require() function.

Export Object

The exports is an object. So, you can attach properties or methods to it. The following example exposes an object with a string property in Message.js file.

Message.js
```
exports.SimpleMessage = 'Hello world';

//or

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

In the above example, we have attached a property SimpleMessage to the exports object. Now, import and use this module, as shown below.

app.js

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

console.log(msg.SimpleMessage);
```

In the above example, the require() function will return an object { SimpleMessage : 'Hello World'} and assign it to the msg variable. So, now you can use msg.SimpleMessage.

In the same way as above, you can expose an object with function. The following example exposes an object with the log function as a module.

Log.js

```
module.exports.log = function (msg) {
    console.log(msg);
};
```

The above module will expose an object- { log : function(msg){ console.log(msg); } } . Use the above module as shown below.

app.js

```
var msg = require('./Log.js');

msg.log('Hello World');
```

You can also attach an object to module.exports, as shown below.

data.js
```
module.exports = {
    firstName: 'James',
    lastName: 'Bond'
}
```

app.js

```
var person = require('./data.js');
console.log(person.firstName + ' ' + person.lastName);
```

Export Function

You can attach an anonymous function to exports object as shown below.

Log.js

```
module.exports = function (msg) {
   console.log(msg);
};
```

Now, you can use the above module, as shown below.

app.js

```
var msg = require('./Log.js');

msg('Hello World');
```

The msg variable becomes a function expression in the above example. So, you can invoke the function using parenthesis ().

Export Function as a Class

In JavaScript, a function can be treated like a class. The following example exposes a function that can be used like a class.

Person.js

```
module.exports = function (firstName, lastName) {
   this.firstName = firstName;
   this.lastName = lastName;
   this.fullName = function () {
      return this.firstName + ' ' + this.lastName;
   }
}
```

app.js

```
var person = require('./Person.js');

var person1 = new person('James', 'Bond');

console.log(person1.fullName());
```

As you can see, we have created a person object using the new keyword.

In this way, you can export and import a local module created in a separate file under root folder.

Node.js also allows you to create modules in sub folders. Let's see how to load module from sub folders.

Load Module from the Separate Folder

Use the full path of a module file where you have exported it using module.exports. For example, if the log module in the log.js is stored under the utility folder under the root folder of your application, then import it, as shown below.

app.js

```
var log = require('./utility/log.js');
```

In the above example, . is for the root folder, and then specify the exact path of your module file. Node.js also allows us to specify the path to the folder without specifying the file name. For example, you can specify only the utility folder without specifying log.js, as shown below.

app.js

```
var log = require('./utility');
```

In the above example, Node.js will search for a package definition file called package.json inside the utility folder. This is because Node assumes that this folder is a package and will try to look for a package definition. The package.json file should be in a module directory. The package.json under utility folder specifies the file name using the main key, as shown below.

./utility/package.json

```
{
   "name" : "log",
   "main" : "./log.js"
}
```

Now, Node.js will find the log.js file using the main entry in package.json and import it. If the package.json file does not exist, then it will look for index.js file as a module file by default.