Node.js Modules

Modules in Node.js are blocks of encapsulated code that can be reused throughout your application. These modules can include functions, objects, and variables that are exported from the code files.

What are Modules in Node.js?

Modules in Node.js are like JavaScript libraries — a set of related functions that you can include in your application. Every Node.js file can be considered a module that can export code for reuse in other parts of the application. This modular approach allows developers to separate concerns, reuse code, and maintain a clean architecture.

How Node.js Modules Work?

Each module in Node.js is executed within its own module scope. This means variables, functions, and classes defined in one module are private to that module unless explicitly exported for use by other modules. This behaviour avoids global namespace pollution and improves code maintainability.

At its core, a Node.js module is an object that contains the following key properties:

- **exports:** The object that a module can export for use in other modules.
- **require():** A function that is used to import modules into other modules.
- **module:** The object that represents the current module.

When a module is loaded, Node.js wraps the module code in a function to provide this module system.

Different types of Node.js Modules: Core Modules, Local Modules, Third-party modules

1. Core Modules

Node.js has many built-in modules that are part of the platform and come with Node.js installation. These modules can be loaded into the program by using the **require** function.

Syntax: const module = require('module_name');

The require() function will return a JavaScript type depending on what the particular module returns. The following example demonstrates how to use the Node.js http module to create a web server.

```
const http = require('http');
http.createServer(function (req, res) {
   res.writeHead(200, { 'Content-Type': 'text/html' });
   res.write('Welcome to this page!');
   res.end();
}).listen(3000);
```

In the above example, the require() function returns an object because the Http module returns its functionality as an object. The function http.createServer() method will be executed when someone tries to access the computer on port 3000. The res.writeHead() method is the status code where 200 means it is OK, while the second argument is an object containing the response headers. The following list contains some of the important core modules in Node.js:

Core Modules	**Description**
http	creates an HTTP server in Node.js.
assert	set of assertion functions useful for testing.
fs	used to handle file system.
path	includes methods to deal with file paths.
process	provides information and control about the current Node.js process.
os	provides information about the operating system.
querystring	utility used for parsing and formatting URL query strings.

Core Modules	Description
url	module provides utilit

2. Local Modules

Unlike built-in and external modules, local modules are created locally in your Node.js application. Let’s create a simple calculating module that calculates various operations. Create a calc.js file that has the following code:

```
// Filename: calc.js

exports.add = function (x, y) {
   return x + y;
};
exports.sub = function (x, y) {
   return x - y;
};
exports.mult = function (x, y) {
   return x * y;
};
exports.div = function (x, y) {
   return x / y;
};
```

Since this file provides attributes to the outer world via exports, another file can use its exported functionality using the require() function.

```
// Filename: index.js

const calculator = require('./calc');
let x = 50, y = 10;
console.log("Addition of 50 and 10 is "
            + calculator.add(x, y));
console.log("Subtraction of 50 and 10 is "
            + calculator.sub(x, y));
console.log("Multiplication of 50 and 10 is "
            + calculator.mult(x, y));
console.log("Division of 50 and 10 is "
            + calculator.div(x, y));
```

Step to run this program: Run the **index.js** file using the following command: node index.js

3. Third-party modules

Third-party modules are modules that are available online using the Node Package Manager(NPM). These modules can be installed in the project folder or globally. Some of the popular third-party modules are Mongoose, Express, Angular, and React.

Example:

- npm install express
- npm install mongoose

Create a Simple Express Server:

```
const express = require('express');
const app = express();
const port = 3000;
app.get('/', (req, res) => {
  res.send('Hello World!');
});
app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`);
});
```

Run the Server: Use this command to run the node application.

```
node server.js
```

When you navigate to http://localhost:3000 in your browser, you should see **Hello World!**.

Exporting and Importing Modules

Node.js allows developers to define the module’s public API by exporting functions, objects, or classes using module.exports. The code that uses the module can then import these exported entities using require().

1. Default Export:

You can export a single function or object using module.exports:

```
// logger.js
module.exports = function log(message) {
  console.log(message);
};
```

2. Named Exports:

You can export multiple functions or objects by attaching them to the exports object:

```
// math.js
exports.add = function (a, b) {
  return a + b;
};
exports.multiply = function (a, b) {
  return a * b;
};
```

You can then import them as:

```
// app.js
const math = require('./math');
console.log(math.add(2, 3));  // Output: 5
console.log(math.multiply(2, 3));  // Output: 6
```

Core Modules:

1. **Fs Module:** The Node.js file system module allows you to work with the file system on your computer.

Common use for the File System module:

- Read files
- Create files
- Update files

- Delete files
- Rename files

2. **Path Module:** The Path module provides a way of working with directories and file paths.

Method	Description
basename()	Returns the last part of a path
delimiter	Returns the delimiter specified for the platform
dirname()	Returns the directories of a path
extname()	Returns the file extension of a path
format()	Formats a path object into a path string
isAbsolute()	Returns true if a path is an absolute path, otherwise false
join()	Joins the specified paths into one
normalize()	Normalizes the specified path
parse()	Formats a path string into a path object
posix	Returns an object containing POSIX specific properties and methods
relative()	Returns the relative path from one specified path to another specified path
resolve()	Resolves the specified paths into an absolute path
sep	Returns the segment separator specified for the platform

win32	Returns an object containing Windows specific properties and methods

3. **OS Module**: It provides functions to interact with the operating system. It provides the hostname of the operating system and returns the amount of free system memory in bytes.

- **os.arch():** Returns the CPU architecture of the operating system (e.g., ‘x64’, ‘arm’).
- **os.cpus():** Provides an array of objects describing each CPU/core installed.
- **os.freemem():** Returns the amount of free system memory in bytes.
- **os.homedir():** Returns the path to the current user’s home directory.
- **os.hostname():** Returns the hostname of the operating system.
- **os.networkInterfaces():** Returns a list of network interfaces and their details.
- **os.platform():** Returns the operating system platform (e.g., ‘linux’, ‘darwin’).
- **os.release():** Returns the operating system release.
- **os.totalmem():** Returns the total amount of system memory in bytes.
- **os.uptime():** Returns the system uptime in seconds.

4. **Assert Module:** The assert module provides a way of testing expressions. If the expression evaluates to 0, or false, an assertion failure is being caused, and the program is terminated.

Assert Methods

Method	**Description**
assert()	Checks if a value is true. Same as assert.ok()
deepEqual()	Checks if two values are equal

deepStrictEqual()	Checks if two values are equal, using the strict equal operator (===)
equal()	Checks if two values are equal, using the equal operator (==)
fail()	Throws an Assertion Error
ifError()	Throws a specified error if the specified error evaluates to true
notDeepEqual()	Checks if two values are not equal
notDeepStrictEqual()	Checks if two values are not equal, using the strict not equal operator (!==)
notEqual()	Checks if two values are not equal, using the not equal operator (!=)
notStrictEqual()	Checks if two values are not equal, using the strict not equal operator (!==)
ok()	Checks if a value is true
strictEqual()	Checks if two values are equal, using the strict equal operator (===)

NodeJS NPM

NPM (Node Package Manager) is a package manager for Node.js modules. It helps developers manage project dependencies, scripts, and third-party libraries. By installing Node.js on your system, NPM is automatically installed, and ready to use.

- It is primarily used to manage packages or modules—these are pre-built pieces of code that extend the functionality of your Node.js application.
- The NPM registry hosts millions of free packages that you can download and use in your project.
- NPM is installed automatically when you install Node.js, so you don't need to set it up manually.

How to Use NPM with Node.js?

To start using NPM in your project, follow these simple steps

Step 1: Install Node.js and NPM

First, you need to install Node.js. NPM is bundled with the Node.js installation. You can follow our article to Install the Node and NPM- How to install Node on your system

Step 2: Verify the Installation

After installation, verify Node.js and NPM are installed by running the following commands in your terminal:

```
node -v
```

```
npm -v
```

These commands will show the installed versions of Node.js and NPM.

Step 3: Initialize a New Node.js Project

In the terminal, navigate to your project directory and run:

```
npm init -y
```

This will create a package.json file, which stores metadata about your project, including dependencies and scripts.

Step 4: Install Packages with NPM

To install a package, use the following command

```
npm install <package-name>
```

For example, to install the Express.js framework

```
npm install express
```

This will add express to the node_modules folder and automatically update the package.json file with the installed package information.

Step 5: Install Packages Globally

To install packages that you want to use across multiple projects, use the -g flag:

```
npm install -g <package-name>
```

Step 6: Run Scripts

Using NPM Package in the project

Create a file named **app.js** in the project directory to use the package

```
//app.js

const express = require('express');//import the required package
const app = express();

app.get('/', (req, res) => {
  res.send('Hello, World!');
});

app.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

- **express()** creates an instance of the Express app.
- **app.get()** defines a route handler for HTTP GET requests to the root (/) URL.
- **res.send()** sends the response "Hello, World!" to the client.
- **app.listen(3000)** starts the server on port 3000, and console.log() outputs the server URL.

Now run the application with

```
node app.js
```

Visit **http://localhost:3000** in your browser, and you should see the message: **Hello, World!**

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

console.log(colors.green('hello')); // outputs green text
console.log(colors.red.underline('i like cake and pies')) // outputs red underlined text
console.log(colors.inverse('inverse the color')); // inverses the color
console.log(colors.rainbow('OMG Rainbows!')); // rainbow
console.log(colors.trap('Run the trap')); // Drops the bass

colors.setTheme({
   custom: ['red', 'underline']
 });

 console.log('test'.custom);
```

Steps to Create Node Application (Web-Based)

Step 1: Import required modules

Load Node modules using the required directive. Load the http module and store the returned HTTP instance into a variable.

Syntax:

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

Step 2: Creating a server in Node

Create a server to listen to the client's requests. Create a server instance using the **createServer() method**. Bind the server to port 8080 using the listen method associated with the server instance.

Syntax:

```
http.createServer().listen(8080);
```

Step 3: Read request and return response in Node:

Read the client request made using the browser or console and return the response. A function with request and response parameters is used to read client requests and return responses.

Syntax:

```
http.createServer(function (request, response) {...}).listen(8080);
```

First Node Application

After combining all these techniques, you can create a Node application. This code below creates a **Hello World web-based application using Node.js.**

```
// Require http header
var http = require('http');
 // Create server
http.createServer(function (req, res) {
   // HTTP Status: 200 : OK
   // Content Type: text/html
   res.writeHead(200, {'Content-Type': 'text/html'});
    // Send the response body as &quot;Hello World!&quot;
   res.end('Hello World!');
}).listen(8080);
```

Step to Run Node Application:

Step 1: To run the program type the following command in terminal

```
node firstprogram.js
```

Step 2: Then type the following URL in the browser

http://127.0.0.1:8080/