Modules in Node.js

Understanding Module

What is a Module?

A module is a self-contained code that can be easily reused across different parts of an application. It helps in organizing code and makes it easier to maintain and understand. In Node.js, there are two syntaxes available to use modules: CommonJS and ES6 module syntax.

CommonJS Module

CommonJS is the default module system in Node.js that uses the ‘require’ function to import modules and the ‘module.exports’ object to export them.

Here is an example of a CommonJS module:

```
// utils.js
const add = (a, b) => a + b;
const subtract = (a, b) => a - b;

module.exports = { add, subtract };
```

In the above example, a module called utils.js exports two functions, add and subtract, using the module.exports object. They can be used in another file as follows:

```
// app.js
const { add, subtract } = require('./utils');

console.log(add(2, 3)); // output: 5
```

```
console.log(subtract(5, 2)); // output: 3
```

ES6 Module

The ES6 module syntax is a more modern approach that is supported by modern JavaScript environments, and it employs the "import" and "export" keywords.

Here is an example of an ES6 module:

```
// utils.mjs
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
```

In the above example, a module called utils.mjs exports two functions add and subtract using the export keyword. They can be used in another file as follows:

```
// app.mjs

import { add, subtract } from './utils.mjs';
//import * as utility from './utils.mjs'

console.log(add(2, 3)); // output: 5
console.log(subtract(5, 2)); // output: 3

// console.log(utility.add(2, 3));   output: 5;
// console.log(utility.subtract(5, 2));  output: 3;
```

In the above example, the add and subtract functions from the utils.mjs module are imported into the app.mjs file using destructuring and then invoked with arguments 2, 3 and 5, 2 respectively. This would output 5 and 3 to the console.

Note: You must use the .mjs extension for ES6 module files.

Types of Modules

There are three main types of modules in Node.js:

1. **Core modules:** These are built-in modules in Node.js that provide basic functionality for tasks like file I/O, networking, and more. They can be imported and used in your application without installation or configuration. Here are some common core modules:
    - 'fs': The 'fs' module works with the file system, like reading and writing files.
    - 'http': The 'http' module is used for creating HTTP servers and clients.
    - 'path': The 'path' module is used for working with file paths.
2. **Internal/User-defined/local modules:** These are custom modules you create for your own application. They are created as separate files in your application and can be imported and used in other parts of your application using either the CommonJS or ES6 module syntax.
3. **Third-party modules:** These are modules created by other developers and published on package registries like npm. They can be installed using a package manager like npm or yarn and then imported and used in your application.

Package Managers and NPM

Package

A package is a collection of reusable code that can be easily shared and installed in projects. It usually contains one or more modules, along with other files like documentation, license information, and configuration files. Packages in Node.js are usually distributed via the npm registry.

Package Manager

A package manager is a tool that simplifies managing, installing, and updating packages. In the Node.js ecosystem, the most popular package manager is NPM, which unofficially stands for Node Package Manager.
NPM allows:

- Installation of packages
- Version Management
- Managing dependencies
- Publishing packages

Note: When we install Node.js, NPM is installed by default.

Here is the command to the npm version:

```
npm -v
```

Nodemon

Nodemon is a popular utility for Node.js development that automatically restarts your application when file changes are detected. This can save you much time during development, as you won't have to manually restart your server each time you make changes.

To install “Nodemon” globally, you can use the following command in the terminal:

```
npm install -g nodemon
```

This will download and install “Nodemon” globally on your system. Now, you can use nodemon to start your application by running:

```
nodemon app.js
```

Here, app.js is the entry point of your application. Nodemon will watch for changes in your code and automatically restart the server when any changes are detected. This can be a huge time-saver during development.

Understanding Package.json file

Package.json file is a crucial part of any Node.js project and contains metadata about your project such as its name, version, description, author information, dependencies, scripts, and other configuration options.
Dependencies listed in package.json can be installed by running the command `npm install` in the project directory.

Here's an example of a package.json file for a Node.js project that includes nodemon as a dev dependency:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "A cool Node.js project!",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "go": "nodemon server.js"
  },
  "author": "Coding Ninjas",
  "dependencies": {
    "mongodb": "^5.1.0"
  },
  "devDependencies": {
    "nodemon": "^2.0.22"
  }
}
```

In this example, we have added "nodemon": "^2.0.22" as a dev dependency by including it in the devDependencies section of the package.json file. We have also added a new script called "go" that runs nodemon with the server.js file.

This means that during development, we can use `npm run go` to start the server with nodemon, which will automatically restart the server whenever changes are made to the code.

Dependencies and devDependencies

In a Node.js project, the package.json file lists all of the external dependencies that the project needs to run. Two types of external dependencies can be included in the file:

- **dependencies:** These are the dependencies that are required for the project to run in a production environment. An example of a dependency could be MongoDB, which is a popular database system used with Node.js. You can install MongoDB as a dependency using the following command: `npm install mongodb`

- **devDependencies:** These dependencies are only needed for development purposes, such as testing or building the project. They are installed with the command npm install and can be listed with the --save-dev or -D flag. For example, to install Nodemon as a dev dependency, you can run the following command: `npm install nodemon --save-dev` or `npm install -D nodemon`

Listing dependencies separately can help make it clear which dependencies are required for the project to run in production, and which are only needed for development. This can also help with managing dependencies and reducing the size of the project's node_modules directory.

Understanding package-lock.json

- The package-lock.json file is automatically generated by npm when packages are installed.

- It contains information about the exact versions of all installed packages and their dependencies.
- The purpose of package-lock.json is to ensure the same versions of packages are installed on every environment your project runs in, avoiding unexpected issues caused by different package versions.
- Without package-lock.json, developers or environments might end up with different package versions, leading to inconsistent behavior and bugs that are difficult to reproduce and fix.

Differences between package.json and package-lock.json

The package.json file contains high-level information about a Node.js project and its direct dependencies, while the package-lock.json file stores the exact versions of all installed packages, including nested dependencies. In other words, the package.json file is a metadata file that lists the project's dependencies, scripts, and other configuration options, while the package-lock.json file is a detailed record of the exact versions of each dependency, ensuring consistency across different environments.

Understanding Node Version Manager (NVM)

Node Version Manager (NVM) is a tool that allows the management of multiple Node.js versions on a computer. As a developer, managing multiple projects with different Node.js versions can be difficult, but NVM helps to switch between them easily, ensuring consistency and smooth operation.

How to use NVM

Installing NVM

1. Open a terminal window.
2. Install NVM using one of the following commands, depending on your operating system:

- **For Mac:**
    1. Install homebrew using:
       `/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"`
    2. Verify that homebrew is installed correctly by running `brew`
    3. Use `brew install nvm` to install nvm.
    4. Use `source $(brew --prefix nvm)/nvm.sh` to configure nvm with the shell system.
- **For Windows:** You can download and run the installer from the NVM Windows repository on GitHub: https://github.com/coreybutler/nvm-windows/releases

3. Verify that NVM is installed correctly by running `nvm --version`

Some basic commands to use NVM:

1. To list all the available Node.js versions:
```
nvm ls-remote
```
2. To install a specific Node.js version:
```
nvm install <version>
```
3. To switch to a different Node.js version:
```
nvm use <version>
```
4. To set a default Node.js version to use in new shells:
```
nvm alias default <version>
```

Reading Data from Console

Reading user input from the console is crucial for building interactive CLI apps and collecting user preferences in command-line tools. This feature enables

command-line applications to interact with users and process their input in the application. Other use cases include creating a chat application, a command-line calculator, or using git and npm commands in the terminal.

Readline module

To read input from the console in Node.js, we can use the built-in Readline module. Readline is a module that provides an interface for reading data from a Readable stream (such as process.stdin) on a line-by-line basis.

To use the Readline module, we need to require it at the beginning of our file:

```
const readline = require('readline');
```

Here's an example code snippet that takes two inputs from the command line using Readline module and returns their sum:

```
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('Enter the first number: ', (num1) => {
  rl.question('Enter the second number: ', (num2) => {
    const sum = parseInt(num1) + parseInt(num2);
    console.log(`The sum is: ${sum}`);
    rl.close();
  });
});
```

The readline module exports several functions that we can use to interact with the console. The most commonly used are:

- readline.createInterface(): This function creates a new Readline interface, which provides methods for reading input from the console.
- rl.question(): This function displays a prompt to the user and waits for them to enter a response. The response is then passed to a callback function that we provide.
- rl.close(): This function closes the Readline interface and frees up resources.

File System Module

The FileSystem module, also known as 'fs', is a built-in module in Node.js that provides a variety of methods for interacting with the file system. It enables you to perform various operations on files and directories, including reading, writing, deleting, and updating them.

Types of “fs” methods

There are two types of fs methods available: blocking (or synchronous) and non-blocking (asynchronous).

Using Synchronous Methods

1. Reading a file: To read a file, you can use the fs.readFileSync() method, which returns a buffer. You can convert the buffer to a string, or you can set the encoding property to 'utf8' to get the content directly as a string.

```
const fs = require('fs');
const path = 'example.txt'; // A file at path exists

// Reading a file synchronously
const data = fs.readFileSync(path,{encoding :'utf8'});
console.log(data);
```

2. Writing to a file: To write data to a file, use the fs.writeFileSync() method. It creates a new file or overwrites an existing one.

```
const fs = require('fs');
const path = 'example.txt';

// Writing to a file synchronously
fs.writeFileSync(path, 'Hello World!');
console.log('File written successfully.');
```

3. Updating a file: If you want to append content to an existing file, use the fs.appendFileSync() method.

```
const fs = require('fs');
const path = 'example.txt';

// Updating a file synchronously
fs.appendFileSync(path, '\nThis is an update.');
console.log('File updated successfully.');
```

4. Deleting a file: To delete a file, use the fs.unlinkSync() method.

```
const fs = require('fs');
const path = 'example.txt';

// Deleting a file synchronously using fs.unlinkSync()
fs.unlinkSync(path);
console.log('File deleted successfully.');
```

Using Asynchronous Methods

We learned about performing CRUD operations using blocking code with the File System module, which means while file operations are being performed, our main thread is blocked, and it can't perform any other task. Blocking code can make your app less responsive and slow. To avoid this, we can use non-blocking or asynchronous methods available in the FileSystem module.

1. Reading a file: To read a file asynchronously, use fs.readFile().

```
//Reading data
fs.readFile('data.txt', (err, data) => {
  if (err) {
    console.log(err)
  } else {
    console.log(data.toString())
  }
})
```

2. Writing to a file: To write data to a file asynchronously, use fs.writeFile().

```
// Write data
fs.writeFile('employee.txt', 'New Employee', (err) => {
     if(err)
          console.log(err)
     else
          console.log('File is written')

     });
```

3. Updating a file: To append content to a file asynchronously, use fs.appendFile().

```
fs.appendFile('employee.txt', '\n Another Employee',
     (err) => {
         if(err)
             console.log(err)
         else
             console.log('File is updated')
     });
```

4. Deleting a file: To delete a file asynchronously, use fs.unlink().

```
fs.unlink('employee.txt', (err) => {
     if(err)
```

```
        console.log(err)
    else
        console.log('File is deleted')
    });
```

Path module

The Path module in Node.js is a built-in module that provides various methods to work with file paths. It can be used to normalize, join, resolve, and manipulate file and directory paths.

1. path.join(): This method joins two or more path segments using the platform-specific separator and returns the combined path.
   Here is an example:

```
const path = require('path');
const filePath = path.join('folder', 'file.txt');
console.log(filePath);
// output: folder/file.txt  (on Unix-based systems)

// output: folder\file.txt (on Windows systems)
```

2. path.resolve(): This method resolves the given sequence of paths or path segments into an absolute path. It takes multiple arguments as input and returns the resolved absolute path.

   Here is an example:

```
const path = require('path');
const absPath = path.resolve('folder', 'file.txt');
console.log(absPath);
// output:
```

```
 /home/user/folder/file.txt (on Unix-based systems)

//output:
 \home\user\folder\file.txt (on Windows systems)
```

3. path.extname(): This method returns the extension of the given file path. It takes a file path as input, and returns the extension (including the dot). If there is no extension, an empty string is returned.

   Here is an example:

```
const path = require('path');
const ext = path.extname('file.txt');
console.log(ext); // output: .txt
```

Summarising it

Let's summarise what we have learned in this module:

- Types of modules and how to use them.
- Package managers and NPM.
- Nodemon and how it can be used for automatic server restarts.
- package.json and package-lock.json files.
- dependencies and devDependencies
- NVM to manage multiple Node.js versions on the same machine.
- Reading data from the console using the readline module.
- File System module, including synchronous and asynchronous methods for file operations.
- Path module and its usage.

Some Additional Resources:

- **https://betterprogramming.pub/use-nvm-to-manage-node-js-and-npm-versions-2bd0d0875f9f**
- **https://nodejs.org/api/readline.html**
- **https://nodejs.org/api/fs.html**
- **https://nodejs.org/api/path.html**