upGrad

Lecture notes: Fundamentals of Node.js

1. Introduction and Installation of Node.js

1.1 What is Node.js?

Node.js is a JavaScript runtime environment built on Chrome's V8 JavaScript engine. All types of browsers are capable of running the JavaScript code, making JavaScript usable as a client-side scripting language; however, JavaScript can also be used as a server-side scripting language.

Modern browsers utilise the Just-In-Time (JIT) compilation technique, which compiles the entire code and converts it into the low-level machine code. Thus, an interpreter is not needed as an intermediary to run the code. This reduces the execution time significantly and leads to better performance.

In the client-side JavaScript, the code is detected by a browser and handed over to the browser's JavaScript engine. In contrast, the server-side JavaScript does not use any browser and is directly executed by the engine. This is how Node uses the browser's (Google Chrome's) engine V8 without using the browser.

There are certain differences between the client-side JavaScript and Node.js, which are as follows:

1. Node.js is a modified version of Chrome's V8 JavaScript engine.
2. Node offers additional features such as local file system management.
3. DOM manipulation is dependent on the browser, and because the Node does not use any browser, such features are not supported in Node.

1.2 Environment Set-Up

In this module, you set up your local machine based on its operating system.

2. Core Concepts of Node.js

2.1 Session Overview

In this session, you learnt about I/O models and how Node.js uses the non-blocking I/O model. You also gained an in-depth understanding of what happens behind the scenes in JavaScript runtime. You learnt how to use the global-looking variables in each module to check how they remain local to each 'module'. Finally, you learnt how to export the constructs declared in one module and import them into another module in order to use them.

2.2 REPL

In this module, you learnt how to invoke the REPL environment in which you can execute your Node.js code. You can write the code in it the same way you used to write code in Chrome's console.

Some of the important points regarding REPL can be summarised as follows:.

1. REPL (Read, Evaluate, Print, Loop) is an interactive environment where you can read input from the user, evaluate it, print the result(s) of the evaluation and loop these three commands until termination.
2. REPL enables you to view the results of your input code immediately without going through the compilation phase in the cycle. You can directly execute the code without compiling it.
3. The console tab in Chrome where we used to execute our JavaScript code is also based on REPL.

2.3 Alternative to REPL

The content of the memory heap is kept only for the current session in the REPL environment. We used the VS Code to write the code and learnt how to execute the code that we had written.

2.4 JavaScript: Runtime Environment

In this session, you learnt the following about JavaScript Runtime:

1. Memory Heap

- This is a data structure that contains the memory allocated to all the variables and functions used in a program.

2. Execution/Call Stack
    - This is a data structure that indicates where in the program you are.

3. Web APIs Container
    - This contains all the long-running tasks such as event listeners, HTTP/AJAX requests and HTML Timer APIs, etc. Each long-running task must ideally have a callback associated with it, which must be invoked after the completion of the long-running task. Now, when an event is triggered or the long-running task finishes its execution, the associated callback is sent to the callback queue.

4. Callback Queue
    - The callback queue stores all the callback functions in the order in which they are added. It follows the traditional approach of FIFO (First-In-First-Out). The first callback added to the queue is the one that will be executed first. When the execution (or call) stack is empty, the callback is sent to the stack to be executed.

5. Event Loop
    - This is a loop that constantly monitors the execution (or call) stack and the callback queue. If the callback queue consists of some code to be executed, but the stack is not empty, the event loop waits for it to get empty. Whenever it finds the execution stack empty (and the callback queue contains a callback method), it sends the callback at the beginning of the queue to the top of the stack. A callback is executed when it is pushed to the stack. In another scenario, when both the execution stack and the queue are empty, the event loop waits for some callback to be added to the callback queue.

2.5 I/O Model

In this session, you learnt about the I/O model and the difference between the blocking I/O model and the non-blocking I/O model.

1. Node.js uses an event-driven, non-blocking I/O model.
2. The I/O (Input/Output) model not only refers to the input/output operations but also to other operations ranging from writing/reading files on your local

system to making an API request to read/write data to a remote system to routing the requests.

3. In the blocking I/O model, the subsequent code is blocked by the preceding code. Until an operation finishes its execution, the next operation cannot be performed.
4. In the non-blocking I/O model, the code is not blocked by the preceding code.
5. JavaScript follows the non-blocking I/O model.

2.6 module Keyword

- In Node.js, each file is treated as a separate module.

- Before executing the code written in a module, Node.js takes this code and converts it into a function wrapper, which has the following syntax:

```
(function(exports, require, module, __filename, __dirname) {
  // Module code actually lives in here
});
```

- The function wrapper ensures that the code written inside a module is private to it unless explicitly stated otherwise (exported). The parameters 'exports', 'require', 'module __filename' and '__dirname' act as the variables that are global to the entire code in a module.

- The **module** keyword refers to the object representing the current module. **Module.exports** is the property of a module object that is used to define what will be exported by a module and made available to another module. It is empty by default.
    - There are two methods that can be used to export functions, objects, or primitive values from any module: **. (dot) notation** and **[] (square bracket) notation**.
        - **circle Module (Dot Notation):**
          Example:

```
const PI = 3.14;
const calculateArea = r => PI * r * r;
const calculateCircumference = r => 2 * PI * r;

module.exports.calculateArea = calculateArea;
module.exports.calculateCircumference =
calculateCircumference;
```

- **circle Module (Bracket Notation):**
  Example:

```
const PI = 3.14;
const calculateArea = r => PI * r * r;
const calculateCircumference = r => 2 * PI * r;

module.exports['calculateArea'] = calculateArea;
module.exports['calculateCircumference'] =
calculateCircumference;
```

2.7 require Keyword

In this segment, you learnt the following:

1. The **require** keyword refers to a function that is used to import all the variables and functions exported using module.exports in another module.
   Syntax:

```
require('pathToFile')
```

2. A path can be represented in two formats: **absolute** path and **relative** path. An **absolute** path is an exact path followed, whereas a **relative** path is a path relative to the current file.

3. The **require** function returns the module.exports object. While exporting something from a module, you can use any valid identifier.

4. If you do not export any value from the module, its value will be undefined.

You can read more about the **module** and **require** keywords from the following link.

2.8 Other Methods of Exporting Using module.exports

You learnt another way to export constructs from a module, which is an alternative to the traditional method. You just need to use a single module.exports statement and define all the things (functions, objects, or primitive values) to be exported inside it.

```
// using object literal notation to export everything at once
    module.exports = {
```

```
  givenAlias: functionName,
  givenAlias1: functionName1
}
```

You can also use the module.exports while defining the function so that you do not have to export them explicitly.
Syntax:

```
module.exports = functionName = () => {
        // function code
}
```

2.9 exports Keyword

Before executing the code written in a module, Node.js converts the code from the module into a function wrapper, which has the following syntax:

```
(function(exports, require, module, __filename, __dirname) {
        // Module code actually lives in here
});
```

1. The **exports** keyword is a shortcut to using *module.exports*.

2. **exports** is a variable that references the *module.exports* object.

3. Before the code in a module is evaluated, the exports variable is assigned the reference to the *module.exports*.

4. At the end of each module, **module.exports** is returned. Thus, the require function in some other module loads the object returned by the module.exports keyword.

5. If you make any changes in a property using the exports variable, the changes will be reflected in the module.exports object because it is pass by reference.

6. When the exports variable is assigned a value, which is a new object, the object to which module.exports was pointing is lost. The module.exports object does not change, i.e., it remains as it was. So, now the exports and module.exports objects are different.

upGrad

Methods of Exporting Constructs Using exports

In this session, you learnt about various ways in which you can use exports with some more examples.

circle Module (Method 1):

```
const PI = 3.14;

const calculateArea = r => PI * r * r;
const calculateCircumference = r => 2 * PI * r;

// using individual exports statement to export each construct one by one
exports.calculateArea = calculateArea;
exports.calculateCircumference = calculateCircumference;
```

circle Module (Method 2):

```
const PI = 3.14;

// using exports statement while defining a function
exports.calculateArea = r => PI * r * r;
exports.calculateCircumference = r => 2 * PI * r;
```

app Module:

```
const circle = require('./circle.js');

const area = circle.calculateArea(8);
const circumference = circle.calculateCircumference(8);

console.log(`Area = ${area}, Circumference = ${circumference}`);
```

Output:

```
Area = 200.96, Circumference = 50.24
```

circle Module (Method 3):

```
const PI = 3.14;

const calculateArea = r => PI * r * r;
```

```
const calculateCircumference = r => 2 * PI * r;

// using a single exports statement to export all constructs at once
exports = {
  calculateArea: calculateArea,
  calculateCircumference: calculateCircumference
}

// fixing the reference by assigning an object inside exports to module.exports
module.exports = exports;
```

app Module:

```
const circle = require('./circle.js');
const area = circle.calculateArea(8);
const circumference = circle.calculateCircumference(8);

console.log(`Area = ${area}, Circumference = ${circumference}`);
```

Output:

```
Area = 200.96, Circumference = 50.24
```

__filename vs. __dirname

The keyword ***__filename*** contains the filename of the current module along with the absolute path of the file.

The keyword ***__dirname*** returns all the directories in the hierarchical order in which the current module file is stored.

upGrad

Modules & Packages

Session Overview

In this module, you learnt how to initialise a project as a Node package. You can automatically run your code using a package named Nodemon. You then learnt about the different types of packages in Node as well as the applications of these packages to understand how they make our life, as a developer, easy. Lastly, you learnt about JSON and some of its important methods and its usage.

Initializing a Project as Node Package

NPM stands for Node Package Manager. It is a CLI tool that helps you import and use packages written in Node.js. It is one of the largest repositories consisting of different packages and modules.

You leant how you can use NPM to initialise a Node project. To initialise the npm project, you need to use the command ***npm init*** and provide all the requested information.
You also learnt about the parameters of the ***package.json*** file.

Nodemon: Third-Party Package

You can leverage NPM to install and use the required packages that are already created by others.
To install any package, we use the command ***npm install packageName***.

The different types of flags that can be specified while installing a package are as follows:

- ***-g*** : This flag is used to install the package globally.
- ***--save-prod/-p***: This is the default option used when we do not specify any flag. It installs the dependency on the production server.
- **--save-dev/*-D*:** This flag is used when a package is not necessary for the server but makes the developer's job easier.
- ***--save-optional/-O:*** This flag is used to install dependencies that are optional.

- ***--no-save :*** *This flag is used to prevent the dependency* from being saved in the *dependency key* in the *package.json* file.

You then learnt about the **package-lock.json** file. This file is automatically generated when the npm modifies either the *node_modules* tree or the *package.json* file. It also contains the name of the common dependencies that all Node projects contain. It guarantees that whenever anyone clones the project, they will get the identical dependency tree.

Custom Script in Node Project

In this session, you learnt about the Nodemon package. Nodemon is a tool that helps you develop Node.js based applications by automatically restarting the Node application when file changes in the directory are detected.

This is not a project dependency; it is provided to make the developer's job easier.

Can you recall which flag you need to use while installing this?

chalk: Third-Party Package

In this session, you learnt about the **chalk** package. You first installed this package using the following command:

npm install chalk

Next, you imported this package using the **'require'** function as follows:

const chalk = require('chalk');

Further, using the properties present in this package, you printed the text "Hello World" in blue using the following command:

console.log(chalk.blue('Hello world!'));

Types of Modules/Packages

In this session, you learnt about the core module in Node. To view core modules, we used the following command:

console.log(require('module').builtinModules);

You then learnt about the fs module, which stands for file-system. This module provides the local file system management capability to Node.js, which is one of the factors that differentiates Node (server-side JavaScript) from the client-side JavaScript.

There are three types of modules/packages in Node.js, which are as follows:

1. **Custom modules/packages:** These modules/packages are created and defined by the user; you need not install a custom module. However, you need to use the require function to import a custom module by providing its path to start using it.

2. **Third-party modules/packages:** These modules/packages are provided by Node Package Manager; they have already been created by someone else for you. You are required to install a third-party module via NPM and import it via the require function while providing the name of the module as the ID to start using it.

3. **Core modules:** These modules are provided by Node by default; you need not define them. Also, these are not required to be installed prior to their usage. You can just use the require function to import a core module to start using it. Examples: *buffer, os, events, http, https, module, path, url, fs, v8, etc.*

fs: Core Module

All the file system operations in Node.js follow synchronous as well as asynchronous forms.

The **asynchronous** form takes a completion callback as its last argument, which is invoked when the asynchronous file operation finishes its operation. The first argument of this completion callback is reserved for the exception, which is assigned the value null or undefined when the file operation is successful.

The **synchronous** form is written inside the try-catch block, and if an error or exception occurs, it is handled by the catch-block.

If you need to execute a busy process, you need to use the asynchronous form because the synchronous form blocks the entire process until they complete, halting all the operations.

You then learnt about the fs module and its two APIs, writeFile and writeFileSync. Let's first take a look at the writeFile API in action.
Syntax:

```
const fs = require('fs');
const data = 'Hello students! Let\'s learn Node.js';
const options = {
  encoding: 'utf8',
  flag: 'w'
}

fs.writeFile('data.txt', data, options, (err) => {
  if (err) throw err;
  console.log('The file has been saved!');
});
```

Output:

The file has been saved!

Content written to data.txt file:

Hello students! Let's learn Node.js

If the file data.txt does not exist, it is first created, and then, the specified text inside the data variable is added to it.
To learn more about the **writeFile API**, you can visit this link.

Next, you learnt about the **synchronous** form of the file operation. For this, you can use the **writeFileSync** API. Its syntax is the same as that for the asynchronous form, but there is no callback option. So, to catch the error, you need to write it in the **try & catch** block.
You can read more about the writeFileSync API here.

events: Core Module

In this session, you learnt how events are created and listened to in Node.js.
For this, the steps that you need to follow are as follows:

1. Import the events module using the require keyword.
2. Create an object of the EventEmitter class.
3. Bind an event with its eventHandler using the on property.
4. Fire an event using the emit property.

When the eventEmitter object emits an event, all the functions attached to this event are fired synchronously and any value retired by this eventListeners are ignored and discarded.

JSON vs JavaScript Objects

JSON stands for **JavaScript Object Notation**. It is a lightweight **data-interchange** format. It is independent of the server-side language.

A valid JSON can be written in either of the following two formats, or a mixture of both:

- A collection of name-value pairs, which is known as an object in many programming languages, and/or
- An ordered list of values, which is known as an array in many programming languages.

JavaScript Object literals are different from JSON in the following ways:

1. In JavaScript Object literals, keys can be written with or without quotes, but in JSON, all keys should be written with double quotes.
2. In JavaScript Object literals, we are allowed to use earlier defined constructs, but in JSON, we are not allowed to do so. You cannot have anything at the root level, except a JSON object or an array in JSON.
3. In JavaScript Object literals, we can have a function corresponding to the key, but in JSON, the value corresponding to a key can only be string, boolean, number, null, object and array.
4. In JavaScript Object literals, comments are allowed, but we cannot add comments in JSON.

Every JSON file is a valid JavaScript Object literal, but every JavaScript Object literal is not a valid JSON.

To read more about this, you can visit this link.

JSON Methods

You learnt about two important JSON methods, *JSON.stringify()* and *JSON.parse()*.

JSON.stringify()

This method accepts an argument as an object or a value and returns JSON of the type string.

Example:

```
const obj = {
  firstName: "Alan",
  lastName: "Cooper"
}

const str = JSON.stringify(obj);
console.log(str);
```

Output:

```
'{"firstName":"Alan","lastName":"Cooper"}'
```

JSON.parse()

This method accepts an argument as JSON in the string format and returns JSON in the object/array format.

Example:

```
const arr = [
  {
    firstName: "Alan",
    lastName: "Cooper"
  },
  {
    firstName: "Bob",
    lastName: "Cooper"
  }
]
const str = JSON.stringify(arr);
```

```
const parsedArr = JSON.parse(str);
console.log(parsedArr);
```

Output:

```
[{firstName: "Alan", lastName: "Cooper"}, {firstName: "Bob", lastName: "Cooper"}]
```

You also learnt about the use of these two functions. JSON is now a widely used data-interchange format, which is language-independent, especially in transmitting data over the web. So, you can use JSON to transmit your request data to a server and receive the response data from the server.

This requires JSON to be converted into a string so that it can be transmitted with a web request. The *JSON.stringify()* method is used to do this. This method is mostly used when a request is to be sent to a server.

When a request is sent to a server, the server responds by sending data, which is known as the response data. This data is in the form of string because, as mentioned earlier, the data is transmitted in the form of text over the web. Now, in order to use the response data, you need to convert this string back into JSON so that you can extract the required information from it. This is when the *JSON.parse()* method is used to convert a string into the JSON format.