Introduction to Node JS & Node JS Modules

Submission Deadline: 09th July 2024

(UKI STU 875 – Ahnaf)

Node.js Basics

1. **What is Node.js and what is it used for?**

   - ✓ Node.js is an open-source platform that lets you run JavaScript on the server.
   - ✓ It is primarily used for building server-side applications, enabling developers to create scalable and high-performance network applications.

2. **Explain the main differences between Node.js and traditional web server environments like Apache or Nginx.**

   - ✓ **Node.js** differs from traditional web servers like Apache and Nginx because it uses a single-threaded, event-driven model, making it efficient for real-time applications. **Apache and Nginx** use multiple threads or processes to handle connections, which is good for serving static content and traditional web tasks. **Node.js** excels in handling many simultaneous connections with low resource usage, while **Apache and Nginx** are better for heavy, CPU-intensive operations. **Node.js** is ideal for dynamic applications like chat apps and APIs, with a vast library of modules available through npm.

3. **What is the V8 engine and how does Node.js utilize it?**

   - ✓ The V8 engine is Google's open-source JavaScript engine, originally developed for the Chrome browser. It compiles JavaScript directly to machine code, enabling fast execution.
   - ✓ Node.js utilizes the V8 engine to run JavaScript code on the server side. By embedding V8, Node.js can execute JavaScript outside of the browser, taking advantage of V8's performance optimizations.

4. **Describe the event-driven architecture of Node.js.**

   - ✓ The event-driven architecture of Node.js revolves around an event loop that continuously listens for and processes events. When an event occurs, such as a user request, Node.js places a callback function on the event queue. The event loop then picks up these callbacks and executes them one by one, ensuring non-blocking and efficient handling of tasks. This approach allows Node.js to handle many simultaneous connections without creating multiple threads, making it highly scalable and efficient for I/O-intensive operations.

5. **What are some common use cases for Node.js?**

   - ✓ **API Servers**: Node.js is great for building fast and scalable backend APIs
   - ✓ **Single-page Applications (SPAs)**: Node.js can power the server-side of JavaScript-based SPAs.
   - ✓ **Streaming Applications**: Node.js handles streaming data well, such as in video or audio services.
   - ✓ **Real-time Applications**: It's ideal for apps needing live updates, like chat or gaming platforms

6. **How does Node.js handle asynchronous operations?**

   - ✓ Node.js handles asynchronous operations by executing tasks in the background and using callback functions to process results once tasks are complete. This non-blocking approach allows Node.js to efficiently handle multiple tasks simultaneously without waiting for each one to finish before moving on to the next.

7. **What is the purpose of the `package.json` file in a Node.js project?**

   - ✓ **Package.json** file acts as a central configuration file that facilitates dependency management, script execution, metadata storage, and module configuration within a Node.js project

8. **Explain the role of the Node Package Manager (NPM).**

   - ✓ **NPM** (Node Package Manager) is used in Node.js for installing, managing, and sharing code packages. It helps developers add functionality, manage dependencies, and automate tasks like testing and deployment.

9. **What is the `node_modules` folder and why is it important?**

   - ✓ The `node_modules` folder stores all the packages and libraries a Node.js project needs. It's important because it keeps all the dependencies in one place, ensuring the project has everything it needs to run correctly. It also helps manage different versions of packages for different projects without conflicts.

10. **How can you check the version of Node.js and NPM installed on your system?**

    - ✓ To check the **version of Node.js** : node -v or node --version
    - ✓ To check **NPM installed** : npm -v or npm –version

11. **How does Node.js handle concurrency and what are the benefits of this approach?**

    - ✓ Node.js handles concurrency using an event-driven, non-blocking I/O model. It operates on a single thread with an event loop that manages incoming requests. When an I/O operation occurs, Node.js registers a callback and continues processing other tasks, only executing the callback when the I/O operation completes.
    - ✓ The benefits of this approach include improved efficiency, as Node.js can handle many connections simultaneously with minimal overhead. It also offers better scalability, making it suitable for high-concurrency applications like real-time web apps. Additionally, this model uses fewer system resources, leading to better performance under heavy loads. Lastly, it simplifies development by allowing developers to manage asynchronous workflows with callbacks, promises, or async/await.

12. **How does Node.js handle file I/O? Provide an example of reading a file asynchronously.**

- ✓ Node.js handles file I/O using its built-in `fs` (file system) module, which provides both synchronous and asynchronous methods for interacting with the file system. The asynchronous methods are preferred for non-blocking operations.

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

// Path to the file you want to read
const filePath = 'example.txt';

// Read the file asynchronously
fs.readFile(filePath, 'utf8', (err, data) => {
  if (err) {
    // Handle error
    console.error('Error reading file:', err);
    return;
  }
  // Handle the file content
  console.log('File content:', data);
});
```

13. **What are streams in Node.js and how are they useful?**

- ✓ Streams in Node.js are a powerful way to handle reading or writing data continuously. They allow data to be processed piece by piece, without having to load the entire dataset into memory, which is particularly useful for handling large files or data streams.

Benefits

- ✓ **Efficient Memory Usage**: Streams process data in chunks, reducing memory consumption and allowing Node.js to handle large datasets or files efficiently.
- ✓ **Composability:** Streams can be chained together to perform complex data manipulations, making your code modular and reusable.
- ✓ **Time Efficiency**: Streams can start processing data as soon as it begins arriving, without waiting for the entire dataset to be available, thus improving performance.

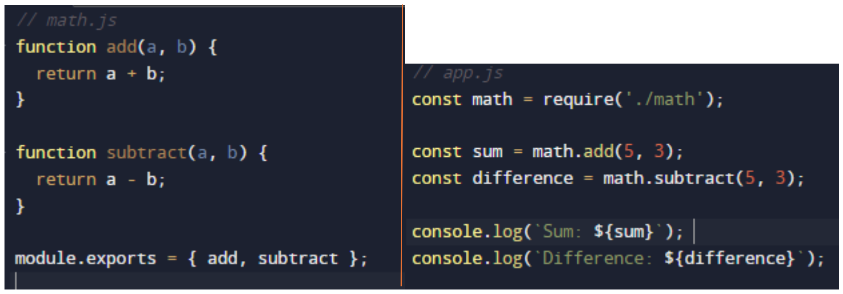
Node.js Modules

1. **What are modules in Node.js and why are they important?**

    - ✓ Modules in Node.js are reusable pieces of code that encapsulate specific functionality. They are important because they help organize code, promote code reuse, and maintain a clear separation of concerns. This modular approach makes development more manageable, scalable, and maintainable.

2. **How do you create a module in Node.js? Provide a simple example.**

   ✓ To create a module in Node.js, write your code in a separate file and export the functions or variables you want to use elsewhere. For example, in a file named `math.js`, you can define functions like `add` and `subtract` and export them using `module.exports`. Then, in another file like `app.js`, you import the module using `require('./math')` and call the exported functions. This way, you can reuse your code across different parts of your application.



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

function subtract(a, b) {
  return a - b;
}

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

```
// app.js
const math = require('./math');

const sum = math.add(5, 3);
const difference = math.subtract(5, 3);

console.log(`Sum: ${sum}`);
console.log(`Difference: ${difference}`);
```

3. **Explain the difference between `require` and `import` statements in Node.js.**

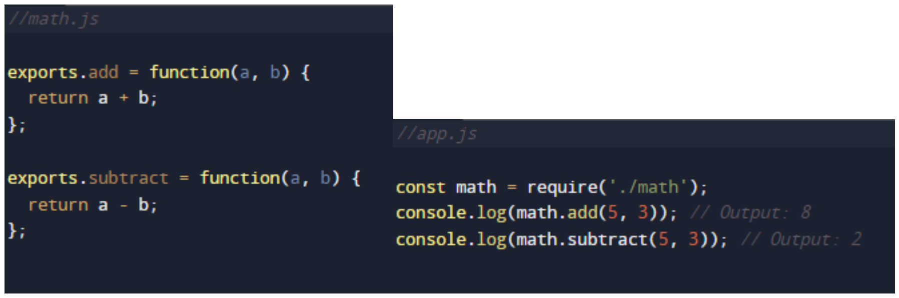
Require	Import
Syntax: const module = require('module-name');	**Syntax:** import { functionName } from 'module-name';
Node.js traditionally uses the CommonJS module system, where **require** is used to load modules.	The **import** statement is part of the ES Module (ESM) system, which is now supported in Node.js.
require is synchronous, meaning it loads modules immediately and blocks execution until the module is fully loaded.	**import** is asynchronous and supports static analysis, making it suitable for tree shaking and better optimization.
Suitable for Node.js applications and widely used in existing codebases.	Requires enabling ESM in Node.js using **.mjs** file extension or **"type": "module"** in package.json.

4. **What is the `module.exports` object and how is it used?**

   ✓ The **module.exports** object in Node.js is used to define what a module exports so other files can import it using require. It allows you to specify which functions, objects, or values should be available when the module is imported.

5. **Describe how you can use the `exports` shorthand to export module contents.**

   ✓ We can use the exports shorthand to export module contents by directly attaching properties or methods to the exports object.



```
//math.js

exports.add = function(a, b) {
  return a + b;
};

exports.subtract = function(a, b) {
  return a - b;
};
```

```
//app.js

const math = require('./math');
console.log(math.add(5, 3)); // Output: 8
console.log(math.subtract(5, 3)); // Output: 2
```

6. **What is the CommonJS module system?**

   ✓ The CommonJS module system is essential for Node.js as it provides a clear and efficient way to organize and reuse code by exporting and importing modules using **module.expor**ts and **require**.

7. **How can you import a module installed via NPM in your Node.js application?**

   ✓ To import a module installed via NPM in your Node.js application, follow these steps:

   1. **Install the Module**: Use the **NPM** command to install the module. For example, to install the **lodash** module, run:

      ```
      npm install lodash
      ```

   2. **Import the Module in Your Code:**

```
// Import the lodash module
const  lodash = require('lodash');

// Use a function from the lodash module
const array = [1, 2, 3, 4];
const reversedArray = lodash.reverse(array.slice());

console.log(reversedArray); // Output: [4, 3, 2, 1]
```

8. **Explain how the `path` module works in Node.js. Provide an example of using it.**

   ✓ The path module in Node.js helps manage and manipulate file paths. It provides functions to join paths, resolve them into absolute paths, and extract details like directory names, file names, and extensions.

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

const filePath = path.join(__dirname, ' files', 'data.txt');

console.log(filePath); // Outputs the absolute path to file.txt
```

9. **How do you handle circular dependencies in Node.js modules?**

   ✓ circular dependencies occur when two or more modules depend on each other directly or indirectly. This can lead to issues during module loading in Node.js, as a module might not be fully loaded yet when another tries to import it.

10. **What is a built-in module in Node.js? Name a few and explain their purposes.**

   ✓ Built-in modules in Node.js are essential functionalities that come pre-installed with Node.js. They provide a variety of core functionalities for common tasks, saving you the effort of writing them from scratch.

- **Fs**: Handles file system operations (e.g., reading/writing files).
- **Http**: Creates HTTP servers and handles requests/responses.
- **Path**: Manages and manipulates file and directory paths.
- **Os**: Provides operating system-related utility methods and properties

11. **What is the difference between relative and absolute module paths in Node.js?**

Relative Module Paths	**Absolute Module Paths**
Specified relative to the current module file (`./` or `../`), used for modules within the project.	Specified from the root of the file system (`/` or drive letter), used for globally installed or fixed location modules.

12. **What is a module wrapper function in Node.js?**

    - ✓ In Node.js, every module is wrapped in a module wrapper function before it is executed. This wrapper function provides a scope for the module, which prevents variable and function names from leaking into the global scope and allows Node.js to inject useful variables into the module.

13. **Describe the `buffer` module and its use in Node.js.**

    - ✓ The buffer module in Node.js is used to handle binary data directly. This module is particularly useful for dealing with file I/O operations, networking, and other scenarios where data needs to be manipulated at a low level.

Starting an HTTP Server in Node.js

1. **How do you create a simple HTTP server in Node.js? Provide a code example.**

2. **Explain the purpose of the `http` module in Node.js.**
3. **What method do you use to start the HTTP server and make it listen on a specific port?**
4. **How can you send a response to the client in an HTTP server created with Node.js?**
5. **Explain the `request` and `response` objects in the context of an HTTP server.**
6. **How do you handle different HTTP methods (GET, POST, etc.) in a Node.js HTTP server?**
7. **What is middleware in the context of a Node.js HTTP server?**
8. **How can you serve static files using an HTTP server in Node.js?**
9. **Explain how to handle errors in an HTTP server created with Node.js.**
10. **How can you implement routing in a Node.js HTTP server without using external libraries?**