Nodejs

Reading Material

PW SKILLS

Topics Covered

- Understanding the Concept
  - Introduction to Nodejs
    - V8 Engine
    - Architecture
  - Node Package Manager (NPM)
  - Module System
  - Global Object
    - Common Properties and Methods
    - Custom Global Variables
  - Path Module
  - OS Module
  - File System Module
  - HTTP Module
  - Events Module
    - Listening to Events
    - Removing Event Listeners
    - Special Events
    - Asynchronous Event Handling
  - Stream and Buffers
    - Types of Streams
    - Buffers in Streams

Conceptual Understanding

Introduction to Nodejs

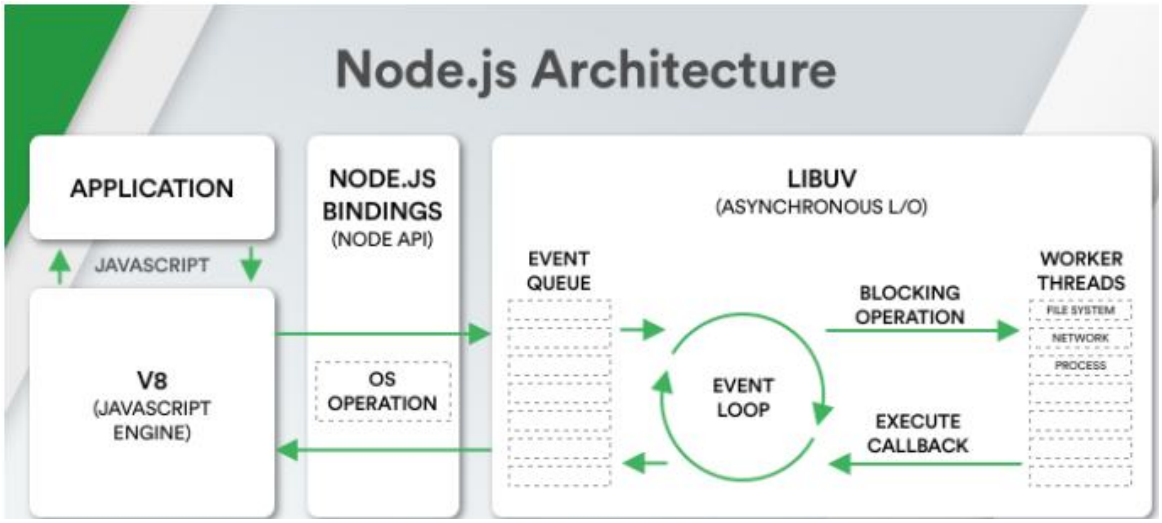
Node.js is a server-side JavaScript runtime environment, enabling developers to use JavaScript for both server-side and client-side scripting. It's event-driven, asynchronous, and single-threaded, making it ideal for web servers, APIs, and real-time applications. Its scalability, fast execution, and npm package manager make development easier, with an active developer community providing resources.

V8 Engine

The V8 engine is an open-source JavaScript engine developed by Google, designed to execute JavaScript code with high performance and efficiency. It consists of five steps: initialize the environment, compile JavaScript codes, generate bytecodes, interpret and execute bytecodes, and optimize some bytecodes for better performance.

Architecture

Node.js is a JavaScript architecture that uses a "Single Threaded Event Loop" design to manage multiple concurrent clients. This design uses an asynchronous model and non-blocking of I/O operations, ensuring efficient resource utilization. Node.js architecture consists of components like requests, servers, event queues, event loops, thread pools, and external resources. This scalable, efficient, and non-blocking architecture is ideal for web applications and software development, improving system performance by optimizing resource utilization.



Node Package Manager (NPM)

Node Package Manager (NPM) is a powerful package manager for JavaScript, primarily used for managing packages and dependencies in Node.js projects. It simplifies the process of installing, sharing, and managing third-party libraries and tools required for Node.js development. It is the default package manager for JavaScript's runtime Node.js.

Key Features of NPM:

Package Management: NPM allows for easy installation, updating, and removal of packages. It maintains a large registry of packages, making it easy to find and use community-developed libraries and tools.

Dependency Management: NPM tracks and manages project dependencies, creating a package.json file to store project metadata and dependencies, ensuring consistent builds across environments.

Version Control: NPM provides robust versioning, allowing developers to specify version ranges or exact versions of packages in package.json, aiding in dependency resolution.

Scripts: Custom scripts can be defined in package.json and executed using npm run, automating tasks like building, testing, and deployment.

Publishing: Developers can publish their packages to the NPM registry, promoting code sharing and collaboration within the Node.js community.

Scoped Packages: Scoped packages, prefixed by a namespace, help prevent naming conflicts and organize related packages.

Security: NPM includes security features that scan for vulnerabilities in dependencies and provide advisories to help keep projects secure.

Usage of NPM:

Installation: Bundled with Node.js, NPM is automatically installed. Packages can be installed locally in a project or globally using the npm install command.

Initialization: The npm init command creates a new Node.js project, generating a package.json file with project metadata and dependencies.

Installing Dependencies: Specified in package.json, dependencies can be installed using npm install, which downloads and installs the packages and their dependencies in the node_modules directory.

Running Scripts: Custom scripts defined in package.json under the "scripts" field can be executed using npm run followed by the script name.

Publishing Packages: Developers can publish packages to the NPM registry using the npm publish command after creating an NPM account. They can also manage access to scoped packages and publish them privately if needed.

Module System

Node.js is a modular programming framework that divides complex applications into manageable modules, enhancing code reusability, maintainability, and collaboration among developers. The system provides a built-in module system to organize code into modules, making it easier to manage dependencies and scale applications. Modules can be categorized into built-in modules, core modules, and local modules. Built-in modules offer essential functionalities for tasks like file system operations, HTTP requests, and cryptography.

Core modules are automatically loaded when Node.js starts and can be accessed using their module names without specifying the file path. Local modules are user-defined components created by developers to encapsulate specific functionalities or components within their applications. Exporting modules allows for easy access to functions, variables, or objects outside the module.

Exporting a Single Function or Object:

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

Exporting Multiple Functions or Objects:

```
// math.js
function add(a, b) {
    return a + b;
}
function subtract(a, b) {
    return a - b;
}
module.exports = {
    add,
    subtract
};
```

Importing Modules:

Once a module is exported, you can import it into other modules using the 'require' function.

Importing a Single Function or Object:

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

Importing Multiple Functions or Objects:

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

Global Object

Node.js' Global object, similar to browser window objects, provides global properties and functions accessible anywhere in the application. However, it's not entirely global, meaning they're only available in the current module scope.

Common Properties and Methods

1. **global:** The global keyword serves as a reference to the global object itself. It can be used to access global variables and functions.
2. **process:** The process object provides information and controls over the current Node.js process. It includes properties and methods related to process management, environment variables, and standard I/O.
3. **console:** The console object is used for printing messages to the console, including log messages, warnings, errors, and debugging information.
4. **require():** The require() function is used to import modules in Node.js. It searches for modules in the node_modules directory and other specified paths.
5. **module and exports:** In Node.js, each file is treated as a separate module. The module object represents the current module, while the exports object is used to export variables, functions, and objects from the module.
6. **__dirname and __filename:** These variables represent the directory path and file name of the current module, respectively.

Custom Global Variables

In addition to the built-in global objects and methods, you can also define custom global variables in Node.js. However, it's generally considered a best practice to avoid polluting the global namespace with too many variables to prevent naming conflicts.

```
global.myVariable = "This is a custom global variable";
console.log(myVariable); // This is a custom global
variable
```

Although the Global object in Node.js provides a convenient way to access commonly used functionalities, it's important to be cautious when using global variables and methods, as they can lead to unexpected behavior and bugs, especially in large-scale applications. Whenever possible, it's recommended to encapsulate your code into modules and only expose the necessary functionalities via exports.

Path Module

Node.js includes a path module that simplifies interaction with file paths. This module, a core component of Node, offers various properties and methods for navigating and manipulating file paths within the file system.

Among its properties, the path module provides:

sep: A property representing the platform-specific path separator.
delimiter: A property representing the path delimiter.
These properties are essential for handling paths across different operating systems.

The path module also offers several useful methods, including:

path.basename(path[, ext]): Returns the last portion of a path, optionally stripping a provided extension.

path.dirname(path): Returns the directory name of a path.

path.extname(path): Returns the extension of the provided path.

path.format(pathObj): Constructs a path string from an object.

path.isAbsolute(path): Checks if a given path is absolute.

path.join(...paths): Joins path segments into a single path.

path.normalize(path): Normalizes a path, resolving '..' and '.' segments.

path.parse(path): Parses a path into an object with directory, root, base, name, and extension properties.

path.relative(from, to): Generates a relative path between two provided paths.

path.resolve(...paths): Resolves a sequence of paths or path segments into an absolute path.
These methods facilitate efficient file path management, aiding developers in navigating and manipulating the file system within Node.js applications.

OS Module

The OS module in Node.js serves as a valuable tool for retrieving crucial information about a computer's operating system. It offers a range of commands and methods to gather various system details, such as:

os.arch(): Retrieves the CPU architecture.
os.freemem(): Retrieves the amount of free memory.
os.totalmem(): Retrieves the total amount of memory.
os.networkInterfaces(): Retrieves information about network interfaces.
os.tmpdir(): Retrieves the default directory path for temporary files.
Additionally, the OS module provides methods to fetch additional system attributes, including:

os.endianness(): Determines the endianness of the system, indicating how bytes are stored in memory.
os.hostname(): Retrieves the hostname, typically used to identify the computer on a network.
os.type(): Retrieves the operating system name, which can vary based on the platform.
os.platform(): Retrieves the platform on which the Node.js process is running.
os.release(): Retrieves the operating system release, which varies based on the specific operating system.
In essence, the OS module plays a crucial role in obtaining comprehensive information about the computer's operating system, aiding in tasks such as system monitoring, resource management, and platform-specific operations within Node.js applications.

Path Module

1. **File System Operations:** The fs module in Node.js provides functionality to perform file system operations, such as reading from and writing to files, creating and deleting files, and manipulating file metadata.

2. **Asynchronous and Synchronous Methods:** fs module offers both asynchronous and synchronous methods for file operations. Asynchronous methods are non-blocking and use callbacks or Promises for handling results, while synchronous methods block the execution until the operation completes.

3. **Common Operations:** Some common operations provided by the fs module include reading files (fs.readFile()), writing to files (fs.writeFile()), appending data to files (fs.appendFile()), deleting files (fs.unlink()), checking file existence (fs.existsSync()), and manipulating directories (fs.mkdir(), fs.readdir()).

4. **Error Handling:** When using asynchronous methods, error handling is crucial. Node.js follows the convention of passing errors as the first argument to the callback function or rejecting the Promise in case of an error.

5. **Buffer and Streams:** Many fs methods can work with either Buffers or Streams. Buffers are used for small amounts of data, while Streams are more efficient for handling large files or continuous data processing.

6. **Path Module Integration:** The path module often complements fs operations by providing utilities for working with file paths. It's commonly used alongside fs for tasks like joining path segments (path.join()), resolving relative paths (path.resolve()), and extracting file extensions (path.extname()).

7. **Permissions and Ownership:** The fs module allows manipulation of file permissions and ownership on POSIX systems using methods like fs.chmod() and fs.chown().

8. **Event Emitters:** Some fs methods return instances of EventEmitter, allowing developers to listen for events such as file open, close, or error, providing a more granular control over file operations.

9. **Performance Considerations:** While synchronous methods may be easier to use, they can block the event loop and degrade performance, especially in I/O-bound applications. Asynchronous methods are generally recommended for better scalability and responsiveness.

10. **Cross-Platform Compatibility:** Node.js fs module abstracts away platform-specific differences, ensuring consistent behavior across different operating systems. However, developers should still be aware of platform-specific limitations and differences in file system behavior.

HTTP Module

Node.js is a core module that provides the HTTP module, allowing users to create HTTP servers and make HTTP requests. This module allows data transfer over HTTP to other users and servers, handling HTTP methods, status codes, headers, and more. To use the HTTP module, import it using the require() function and use the http.createServer() method to create an HTTP server instance. The request and response objects (req and res) are provided by the HTTP server to handle incoming requests and send responses. Different HTTP methods include GET, POST, PUT, and DELETE.

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

const server = http.createServer((req, res) => {
  if (req.url === '/') {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write('<h1>Hello from HTTP server!!!</h1>');
    res.end();
  }
});

const PORT = 3000;
server.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}/
`);
});
```

Events Module

Introduction to Events: Events are described as signals indicating that something has happened or is about to happen in programming. In Node.js, events are crucial for building responsive and efficient applications, especially considering its non-blocking, asynchronous nature.

Listening to Events:

The EventEmitter class from the Event Module is introduced as a central hub for emitting and handling events. The process of listening to events involves creating an instance of EventEmitter, registering listener functions for specific events, and emitting those events.

```
// Import the Event Module
const EventEmitter = require('events');

// Create an instance of EventEmitter
const myEmitter = new EventEmitter();

// Register a listener for the 'hello' event
myEmitter.on('hello', () => {
  console.log('Hello, world!');
});

// Emit the 'hello' event
myEmitter.emit('hello');
```

Removing Event Listeners:

Sometimes, it's necessary to remove listeners for certain events. This can be achieved using the removeListener method, ensuring that specific functions no longer respond to emitted events.

```
// Import the Event Module
const EventEmitter = require('events');

// Create an instance of EventEmitter
const myEmitter = new EventEmitter();

// Define the listener function
const helloListener = () ⇒ {
  console.log('Hello, world!');
};

// Register the listener for the 'hello' event
myEmitter.on('hello', helloListener);

// Emit the 'hello' event
myEmitter.emit('hello');

// Remove the listener for the 'hello' event
myEmitter.removeListener('hello', helloListener);
// Emit the 'hello' event again
myEmitter.emit('hello'); // This will not trigger any
listener
```

Special Events:

Node.js includes special events emitted by certain objects, such as the 'error' event. Handling such events, like errors, is critical to prevent application crashes and ensure robustness.

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

// Create an instance of EventEmitter
const myEmitter = new EventEmitter();

// Register a listener for the 'error' event
myEmitter.on('error', (error) ⇒ {
  console.error('An error occurred:', error.message);
});

// Attempt to read a file that does not exist
fs.readFile('nonexistentfile.txt', (err, data) ⇒ {
  if (err) {
    // Emit the 'error' event if an error occurs
    myEmitter.emit('error', err);
  } else {
    console.log('File contents:', data);
  }
});
```

Asynchronous Event Handling:

Events in Node.js are handled asynchronously, allowing code execution to continue without waiting for event listeners to finish. An example demonstrates how events are emitted asynchronously using setTimeout, showcasing how code execution isn't blocked by event handling.

```
const EventEmitter = require('events');

// Create an instance of EventEmitter
const myEmitter = new EventEmitter();
// Register a listener for the 'hello' event
myEmitter.on('hello', () => {
  console.log('Hello, world!');
});

// Emit the 'hello' event asynchronously after 1 second
setTimeout(() => {
  myEmitter.emit('hello');
}, 1000);
console.log('Event emitted asynchronously');
```

Stream and Buffers

Streams

To handle and manipulate streaming data like a video, a large file, etc., we need streams in Node. The streams module in Node.js manages all streams.

Types of Streams

In Node, there are four different types of streams:
Readable streams → To create a stream of data for reading (say, reading a large file in chunks).
Writable streams → To create a stream of data for writing (say, writing a large amount of data to a file).
Duplex streams → To create a stream that is both readable and writable at the same time. We can read and write to a duplex stream (say, a socket connection between a client and a server).
Transform streams → To create a stream that is readable and writable, but the data can be modified while reading and writing to the stream (say, compressing data by the client and server before requesting).

Buffers in Streams

Streams work on a concept called buffer.

A buffer is a temporary memory that a stream takes to hold some data until it is consumed.

In a stream, the buffer size is decided by the highWatermark property on the stream instance which is a number denoting the size of the buffer in bytes.

A buffer memory in Node by default works on String and Buffer. We can also make the buffer memory work on JavaScript objects. To do so, we need to set the property objectMode on the stream object to true.

If we try to push some data into the stream, the data is pushed into the stream buffer. The pushed data sits in the buffer until the data is consumed.

If the buffer is full and we try to push data to a stream, the stream does not accept that data and returns with a false value for the push action.