Contents

Introduction to Node.js

What is Node.js?

Node.js is an **open-source, cross-platform JavaScript runtime environment** that allows developers to execute JavaScript code outside of a browser. It is widely used for building scalable and high-performance applications.

Key Features of Node.js:

- **V8 Engine**: Node.js runs on the V8 JavaScript engine (also used in Google Chrome), which makes it fast and efficient.
- **Asynchronous and Event-Driven**: Node.js handles I/O operations (network requests, database access, file system operations) asynchronously, meaning it doesn't block the execution of other code.
- **Single-Threaded, Non-Blocking I/O**: Unlike traditional multi-threaded models, Node.js operates on a single thread but can handle multiple requests concurrently.
- **Rich Package Ecosystem**: Node.js has a vast collection of open-source modules available via npm (Node Package Manager), making development faster and easier.

What is a Module in Node.js?

A **module** in Node.js is a reusable block of code that can be imported into other files. It helps in organizing the code and improving maintainability.

Types of Modules in Node.js:

1. **Core Modules**: Built-in modules provided by Node.js, such as http, fs, path, os.
2. **Local Modules**: Custom modules created by developers within a project.
3. **Third-Party Modules**: Modules installed via npm, like express, mongoose, dotenv.

The HTTP Module in Node.js

The **http module** in Node.js allows the creation of web servers and handling HTTP requests and responses.

Creating a Basic Web Server:

```
const { createServer } = require('node:http'); // Importing the HTTP module

const hostname = '127.0.0.1'; // Localhost IP

const port = 3000; // Port number

const server = createServer((req, res) => {

  res.statusCode = 200; // HTTP status code for success

  res.setHeader('Content-Type', 'text/plain'); // Setting response header

  res.end('Hello World'); // Sending response body

});
```

```
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Explanation:

1. ``: Loads the built-in HTTP module.
2. ``: Creates an HTTP server that listens for incoming requests.
3. **Request and Response Objects**:
    - req: Represents the HTTP request (contains request headers, method, URL, etc.).
    - res: Represents the HTTP response (used to send data back to the client).
4. ``: Binds the server to a specific port and hostname.
5. ``: Ends the response and sends data to the client.

Running the Server:

1. Save the file as server.js.
2. Run the command:
3. node server.js
4. Open a browser and visit http://127.0.0.1:3000/ to see the output.

---

1. **Asynchronous programming and callbacks**
2. **Timers**
3. **Promises**
4. **Async and Await**
5. **Closures**
6. **The Event Loop**

Asynchronous Programming and Callbacks JavaScript is single-threaded, meaning it executes one operation at a time. However, asynchronous programming allows tasks like I/O operations, network requests, and timers to run in the background without blocking the main thread. This is crucial in Node.js since it handles multiple requests efficiently.

Callbacks**:** A callback is a function passed as an argument to another function and executed later. Callbacks are commonly used in asynchronous operations. Callback functions are a way to ensure certain code runs only after another code has already finished execution.

Example:

```
function greet(name, callback) {
  console.log('Hello ' + name);
  callback();
}
function logEnd() {
  console.log('Function execution ended.');
}
// Passing 'logEnd' as a callback to the 'greet' function
```

```
greet('Alice', logEnd);
// Output:
// Hello Alice
// Function execution ended.
```

Another Example:

```
function processUserData(userId, callback) {
    // Simulate fetching user data with a 1.5-second delay
    setTimeout(() => {
        if (userId) {
            callback(null, { id: userId, name: "John Doe" }); // Success case
        } else {
            callback("User ID not provided", null); // Error case
        }
    }, 1500);
}
```

- **processUserData:** This function takes two parameters: userId and callback.
    - userId: The ID of the user whose data is to be fetched.
    - callback: A function that will be executed after the data fetching operation completes.
- **setTimeout:** Simulates a delay of 1.5 seconds to mimic an asynchronous operation (like fetching data from a server).
    - After 1.5 seconds, it checks if userId is provided.
    - If userId is provided, it calls the callback function with null as the first argument (indicating no error) and a user object { id: userId, name: "John Doe" } as the second argument (indicating success).
    - If userId is not provided, it calls the callback function with an error message "User ID not provided" as the first argument and null as the second argument.

2. Function Call

javascript

```
processUserData(1, (error, user) => {
    if (error) {
        console.log("Error:", error); // Logs error message if userId is not provided
    } else {
        console.log("User Data:", user); // Logs user data if successful
    }
```

});

- **processUserData(1, callback):** Calls the processUserData function with 1 as the userId and an anonymous function as the callback.
    - Inside the callback function:
        - It checks if there is an error.
            - If there is an error, it logs the error message to the console.
            - If there is no error, it logs the user data to the console.

Timers Timers allow executing code after a specified delay or at intervals. JavaScript provides three main timer functions:

- setTimeout(fn, delay): Executes fn after delay milliseconds.
- setInterval(fn, delay): Repeats execution of fn every delay milliseconds.
- clearTimeout and clearInterval: Stop the execution of timers.

Example:

```
setTimeout(() => console.log("Executed after 2 seconds"), 2000);

let interval = setInterval(() => console.log("Repeating every 1 second"), 1000);

setTimeout(() => clearInterval(interval), 5000); // Stops after 5 seconds
```

A Promise in JavaScript is an object representing the eventual completion (or failure) of an asynchronous operation and its resulting value. It provides a cleaner, more robust way to handle asynchronous operations compared to traditional callback functions, which can lead to callback hell.

Key Concepts

- **Pending:** The initial state; neither fulfilled nor rejected.
- **Fulfilled:** The operation completed successfully.
- **Rejected:** The operation failed.

Creating a Promise

```
const myPromise = new Promise((resolve, reject) => {
  const success = true; // You can change this to `false` to simulate failure
  if (success) {
    resolve("Operation succeeded!"); // If successful, resolve the promise
  } else {
    reject("Operation failed."); // If failed, reject the promise
```

```
  }
});
```

Handling Promises

You can handle promises using then and catch methods:

The then and catch methods are used to handle the outcome of a Promise in JavaScript.
then Method
The then method is used to handle the resolved value of a Promise. It takes up to two arguments:

1. A callback function for the onFulfilled case (when the Promise is resolved successfully).
2. An optional callback function for the onRejected case (when the Promise is rejected).

catch Method
The catch method is used to handle the rejected value of a Promise. It takes one argument, a callback function for the onRejected case.

```
myPromise
  .then((message) => {
    console.log("Success:", message); // Runs if the promise is resolved
  })
  .catch((error) => {
    console.log("Error:", error); // Runs if the promise is rejected
  });
```

Using Async/Await

Async/await syntax provides an even cleaner way to work with promises:

Asynchronous in JavaScript
In JavaScript, asynchronous operations don't block the execution of other code. This is achieved using constructs like callbacks, Promises, and the async/await syntax.
async Keyword
The async keyword is used to declare an asynchronous function. This function returns a Promise implicitly, and you can use the await keyword within it.
await Keyword
The await keyword can only be used inside an async function. It pauses the execution of the async function until the Promise is resolved or rejected.
Handling Errors
When using async and await, it's important to handle errors using try and catch blocks. If the awaited Promise is rejected, the code in the catch block will execute.

```
async function handleOperations() {
  try {
    const result1 = await asyncOperation1;
```

```
    console.log(result1);
    const result2 = await asyncOperation2;
    console.log(result2);
  } catch (error) {
    console.error(error);
  }
}
handleOperations();
```

In this example, async keyword is used to define an asynchronous function, and await pauses the function execution until the promise settles.

Summary

- **Promises** simplify asynchronous code, making it more readable and manageable.
- **Then** and **catch** methods handle success and error cases.
- **Chaining** allows for sequential asynchronous operations.
- **Async/await** provides a syntactically cleaner approach for dealing with promises.

Async and Await async and await make asynchronous code look synchronous and are used with Promises.

Example:

```
function fetchData() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Data fetched successfully!");
    }, 2000);
  });
}
async function getData() {
  console.log("Fetching data...");
  try {
    const data = await fetchData(); // Pauses here until fetchData() is resolved
    console.log(data); // Logs "Data fetched successfully!" after 2 seconds
  } catch (error) {
```

```
    console.error("Error:", error); // Catches and logs any error from fetchData()
  }
  console.log("Done fetching data.");
}
getData();
```

Breakdown of the Example

1. **fetchData Function:**
    - Returns a Promise that resolves with the message "Data fetched successfully!" after a 2-second delay.
2. **getData Function:**
    - Declared as an async function.
    - Logs "Fetching data..." to the console.
    - Uses await to wait for fetchData to resolve and assigns the resolved value to the data variable.
    - Logs the data to the console.
    - Logs "Done fetching data." to the console after the asynchronous operation is complete.

Closures

A closure is a function that remembers the environment in which it was created. It allows a function to access variables from its outer (enclosing) scope even after that outer function has finished executing.

Here's an example to illustrate the concept:

```
function outerFunction() {
  const outerVariable = 'I am from the outer scope';
  function innerFunction() {
    console.log(outerVariable); // Accesses outerVariable even after outerFunction is done
  }
  return innerFunction;
}
const myClosure = outerFunction();
myClosure(); // Logs: 'I am from the outer scope'
```

Explanation:

- outerFunction creates a local variable outerVariable and defines innerFunction.
- innerFunction has access to outerVariable even after outerFunction has returned.
- myClosure holds the innerFunction, which still has access to outerVariable due to closure.

The Event Loop The Event Loop handles JavaScript's asynchronous operations, ensuring that non-blocking code is executed efficiently. It processes the **Call Stack**, **Web APIs**, **Callback Queue**, and **Microtask Queue (Promises)** in a cyclic manner.

Example:

```
console.log("Start");

setTimeout(() => console.log("Timeout callback"), 0);

Promise.resolve().then(() => console.log("Promise resolved"));

console.log("End");
```

Output:

Start

End

Promise resolved

Timeout callback

Promises are executed before setTimeout because they are in the **Microtask Queue**, which has higher priority than the **Callback Queue**.

V8 JavaScript Engine

V8 is the JavaScript engine that makes Chrome and Node.js run JavaScript fast. Let's break it down:

1. **What is V8?**
    - It is the engine that runs JavaScript code inside Google Chrome.
    - Node.js also uses it to run JavaScript outside the browser.
2. **What does it do?**
    - It **parses** (reads) and **executes** JavaScript code.
    - The browser provides extra features like the **DOM (Document Object Model)** and other Web APIs.
3. **Other JavaScript Engines:**
    - **SpiderMonkey** → Used in Mozilla Firefox
    - **JavaScriptCore (Nitro)** → Used in Safari
    - ◇ **V8** → Used in Chrome & Edge (Edge now uses Chromium)

How JavaScript Runs in V8 (Compilation & Performance)

1. **JavaScript was originally "interpreted"** (executed line by line).

2. **Now, V8 compiles JavaScript** before running it (this makes it faster).
3. **JIT (Just-In-Time) Compilation**
    - Instead of running directly, JavaScript is first **compiled into machine code** (fast format for computers).
    - This speeds up execution a lot, which is crucial for big applications like Google Maps.

---

npm (Node Package Manager) – Your JavaScript Toolbox

What is npm?

- **npm = Node.js package manager** (used to install and manage code libraries).
- It has **millions of reusable packages** for JavaScript developers.

Installing All Dependencies

- If a project has a package.json file, run:
- npm install
    - This installs all required libraries in the **node_modules** folder.

Common npm Flags

Flag	Meaning
--save	Adds package to dependencies (default in npm 5+)
--save-dev	Adds package to development dependencies
--no-save	Installs but does not add to package.json
--save-optional	Adds to optional dependencies
-S	Shortcut for --save
-D	Shortcut for --save-dev

Updating Packages

- npm update
- Update a specific package:
- npm update <package-name>

Versioning with npm

- You can install a specific version of a package:
- npm install <package-name>@<version>

Understanding ECMAScript 2015 (ES6) and Beyond in Node.js

1. What is ECMAScript (ES6)?

- ES6 (ECMAScript 2015) introduced many modern JavaScript features.
- Node.js follows the latest updates of this standard by using the **V8 engine**.

2. Features in Node.js

There are three types of JavaScript features in Node.js:

1. **Shipping features** – Fully stable and enabled by default.
2. **Staged features** – Almost ready but require a special flag (--harmony).
3. **In-progress features** – Still being developed, risky to use.

👉 **To check available in-progress features:**
Run this command in your terminal:

```
node --v8-options | grep "in progress"
```

Development vs Production in Node.js

- Node.js itself has no special settings for "development" or "production."
- However, some libraries check the **NODE_ENV** variable to adjust their settings.

Best Practice: Set NODE_ENV to Production

```
NODE_ENV=production node app.js
```

- This improves performance by disabling extra debugging tools.

Why NODE_ENV Can Be a Bad Practice?

- Developers sometimes use it to change how the code works in different environments:

```
if (process.env.NODE_ENV === 'development') {
  console.log("Development mode!");
}
if (process.env.NODE_ENV === 'production') {
  console.log("Production mode!");
}
```

- This can **cause unexpected bugs** because **staging and production behave differently**.
- Instead, use **feature flags or config files** to handle environment-specific behavior.

WebAssembly in Node.js

1. What is WebAssembly?

- A super-fast, low-level language that runs in **browsers and Node.js**.
- You can write WebAssembly (.wasm files) in languages like **C, C++, Rust, or AssemblyScript**.

2. WebAssembly Key Concepts

- **Module** – The compiled WebAssembly file (.wasm).
- **Memory** – A resizable memory buffer.

- **Table** – A list of references (like function pointers).
- **Instance** – A running version of a WebAssembly module.

☐What is Undici?

- **Undici** is an HTTP client library for **Node.js** that powers the fetch API.
- It is built **from scratch** and does **not** use Node.js's built-in HTTP module.
- It is **high-performance** and good for handling many requests efficiently.

2☐Basic Usage of Fetch API with Undici

📌 GET Request (Fetching Data)

Here's how you can make a GET request to fetch data in Node.js using the node-fetch library.

```
// Import the node-fetch library
const fetch = require('node-fetch');
// URL of the API you want to fetch data from
const apiURL = 'https://jsonplaceholder.typicode.com/posts';
// Function to fetch data
async function fetchData() {
    try {
        const response = await fetch(apiURL);
        if (!response.ok) {
            throw new Error('Network response was not ok ' + response.statusText);
        }
        const data = await response.json();
        displayData(data);
    } catch (error) {
        console.error('There was a problem with the fetch operation:', error);
    }
}
// Function to display data
function displayData(data) {
    data.forEach(item => {
        console.log(`ID: ${item.id}, Title: ${item.title}`);
```

```
    });
}
// Call fetchData
fetchData();
```

Explanation

- **API URL**: Replace 'https://jsonplaceholder.typicode.com/posts' with the URL of the API you want to fetch data from.
- **Import node-fetch**: This line imports the node-fetch library, which allows you to use the fetch function in Node.js.
- **fetchData Function**: Uses fetch to make a GET request to the API.
    - await fetch(apiURL): Waits for the response from the API.
    - Checks if the response is OK. If not, throws an error.
    - await response.json(): Parses the response as JSON.
- **displayData Function**: Logs the fetched data to the console.
    - Iterates over the data and logs each item's ID and title.

📌 POST Request (Sending Data)

```
// Import the node-fetch library
const fetch = require('node-fetch');

// URL of the API you want to post data to
const apiURL = 'https://jsonplaceholder.typicode.com/posts';

// Data to be sent in the POST request
const data = {
    title: 'foo',
    body: 'bar',
    userId: 1
};

// Function to make a POST request
async function postData() {
    try {
```

```
        const response = await fetch(apiURL, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        });
        if (!response.ok) {
            throw new Error('Network response was not ok ' + response.statusText);
        }
        const responseData = await response.json();
        console.log('Response:', responseData);
    } catch (error) {
        console.error('There was a problem with the fetch operation:', error);
    }
}

// Call postData
postData();
```

Explanation

- **API URL**: Replace 'https://jsonplaceholder.typicode.com/posts' with the URL of the API you want to post data to.
- **Data**: This is the data you want to send in the POST request. It is converted to a JSON string using JSON.stringify(data).
- **postData Function**: Uses fetch to make a POST request to the API.
    - method: 'POST': Specifies that this is a POST request.
    - headers: Sets the Content-Type to application/json to indicate that the request body contains JSON data.
    - body: Contains the data to be sent, converted to a JSON string.
    - await fetch(apiURL, { method, headers, body }): Sends the request and waits for the response.
    - Checks if the response is OK. If not, throws an error.
    - await response.json(): Parses the response as JSON and logs it to the console.

Node.js Module System

Node.js follows a **module system** to organize code into separate files. This helps make the code reusable, maintainable, and easy to manage.

1️⃣ Global Object

- In Node.js, the **global object** is an object that is accessible from anywhere in your application.
- It provides built-in functions and variables.
- Unlike in browsers (where the global object is window), in Node.js, the global object is **global**.

☑ **Example:**

```
console.log(global); // Prints all global properties

console.log(global.setTimeout); // Shows that setTimeout is a global function
```

2️⃣ Modules

- A **module** is just a JavaScript file that contains some code.
- Node.js organizes code into different files (modules) to keep things clean.
- There are **three types** of modules:
    1. **Built-in modules** (provided by Node.js, like fs, http, path)
    2. **User-defined modules** (files you create)
    3. **Third-party modules** (installed via npm, like express)

3️⃣ Creating a Module

- To create a module, you write some code inside a separate file and **export** it using module.exports.

📄 **math.js**

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

4️⃣ Loading a Module

- To use a module in another file, we **import** it using require().

app.js

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

---

5 Module Wrapper Function

- Every module in Node.js is **wrapped** inside a function before execution.
- This function provides the following arguments:
  - exports → Shortcut for exporting
  - require → Function to import modules
  - module → Object representing the current module
  - __filename → Path of the current file
  - __dirname → Directory of the current file

Example:

```
console.log(__filename); // Prints the full path of this file
console.log(__dirname);  // Prints the directory where this file is located
```

---

6 Path Module

- **Joining Paths:** Combines multiple path segments into one.
- **Resolving Paths:** Converts a sequence of paths into an absolute path.
- **Extracting Parts:** Retrieves information like directory name, base name, extension, etc.
- **Normalization:** Normalizes a path, resolving .. and . segments.

```
// Import the path module from Node.js
const path = require('path');

// Join multiple path segments into a single path
const fullPath = path.join('/users', 'md', 'projects');
// Log the full joined path
console.log('Full Path:', fullPath);

// Resolve a sequence of paths into an absolute path
const absolutePath = path.resolve('users', 'md', 'projects');
// Log the absolute path
```

```
console.log('Absolute Path:', absolutePath);

// Get the last portion (file name) of a path
const fileName = path.basename('/users/md/projects/index.html');
// Log the file name
console.log('File Name:', fileName);

// Get the directory name of a path
const dirName = path.dirname('/users/md/projects/index.html');
// Log the directory name
console.log('Directory Name:', dirName);

// Get the extension of the path
const extName = path.extname('/users/md/projects/index.html');
// Log the extension name
console.log('Extension Name:', extName);
```

OS Module

- The os module is a **built-in module** that provides system-related information.

Example:

```
const os = require('os');
console.log(os.type());        // OS type (Windows/Linux/Mac)
console.log(os.freemem());     // Free memory in bytes
console.log(os.totalmem());    // Total memory in bytes
console.log(os.platform());    // OS platform (win32, linux, darwin)
```

Node.js Module System

Node.js provides **built-in modules** that help developers perform different tasks easily, such as working with files, handling events, and creating servers.

File System (FS) Module

- The **fs (File System) module** is used to interact with files and directories.

- It allows you to:
    - Read files
    - Write files
    - Delete files
    - Rename files
    - Create directories, etc.

☑ **Example: Reading a File**

📄 **app.js**

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

// Read file asynchronously
fs.readFile('example.txt', 'utf8', (err, data) => {
    if (err) {
        console.error(err);
        return;
    }
    console.log(data); // Prints the content of example.txt
});
```

☑ **Example: Writing to a File**

```
fs.writeFile('output.txt', 'Hello, Node.js!', (err) => {
    if (err) throw err;
    console.log('File written successfully!');
});
```

---

2 Events Module

- The **events module** allows Node.js to handle and trigger events.
- It follows the **Observer pattern** (one part of the code listens for events while another part triggers them).
- We use the EventEmitter class to create and manage events.

☑ **Example: Creating and Emitting an Event**

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

// Define an event listener
```

```
emitter.on('greet', () => {
    console.log('Hello! Event triggered.');
});
// Emit (trigger) the event
emitter.emit('greet');
```

3️⃣ Event Arguments

- We can **pass data (arguments)** while emitting an event.
- This helps send information along with an event.

☑ **Example: Passing Arguments in Events**

```
emitter.on('userLoggedIn', (username) => {
    console.log(`User ${username} has logged in.`);
});
emitter.emit('userLoggedIn', 'Luffy');
// Output: User Luffy has logged in.
```

4️⃣ Extending EventEmitter

- Instead of using EventEmitter directly, we can create our own **custom class** that extends it.
- This is useful when we need custom event-handling logic inside a class.

☑ **Example: Creating a Custom EventEmitter Class**

```
const EventEmitter = require('events');
class Logger extends EventEmitter {
    log(message) {
        console.log(message);
        this.emit('messageLogged', { id: 1, text: message });
    }
}
const logger = new Logger();
logger.on('messageLogged', (eventData) => {
    console.log('Listener received:', eventData);
});
logger.log('Hello, World!');
```

Explanation:

1. We create a class Logger that **extends** EventEmitter.
2. The log() method prints a message and emits an event.
3. When the event is emitted, a listener handles it.

5️⃣ HTTP Module

- The http module helps create web servers and handle requests.
- It is used to build backend applications in Node.js.

☑ **Example: Creating a Simple Web Server**

```
const http = require('http');
const server = http.createServer((req, res) => {
   res.writeHead(200, { 'Content-Type': 'text/plain' });
   res.end('Hello, this is a Node.js server!');
});
server.listen(3000, () => {
   console.log('Server is running on port 3000...');
});
```

Explanation:

1. We import the http module.
2. We create a server that sends "Hello, this is a Node.js server!" as a response.
3. The server listens on **port 3000**.

Code With Mosh:

Absolutely! Let's break down what's happening in your code.

app.js

javascript

```
const EventEmitter = require('events');
//Raise an event
emitter.emit('messageLogged', { id: 1, url: "http://" }); //emit means making a noise, produce - signalling that an event has happened
const Logger = require('./logger');
const logger = new Logger();
```

```
// Register a listener
logger.on('messageLogged', (arg) => {
  console.log('Listener called', arg);
});

logger.log('message');
```

logger.js

javascript

```
const EventEmitter = require('events');
var url = 'http://mylogger.io/log';

class Logger extends EventEmitter {
  log(message) {
    // Send an HTTP request
    console.log(message);

    this.emit('messageLogged', { id: 1, url: "http://" }); //emit means making a noise, produce - signalling that an event has happened
  }
}

module.exports = log; //we can change exports variables name here
```

Explanation

1. **EventEmitter:**
    - The EventEmitter class in Node.js allows us to create, manage, and handle custom events. Think of it as a way to signal different parts of your application when something significant happens.
2. **Emitting an Event:**
    - In app.js, we see emitter.emit('messageLogged', { id: 1, url: "http://" }). This line is responsible for emitting or signaling that the messageLogged event has occurred. The object { id: 1, url: "http://"} is passed as the data accompanying the event.
3. **Creating Logger Class:**

- In logger.js, a Logger class is defined that extends EventEmitter. This means that Logger inherits all the functionality of the EventEmitter class.

4. **Logging a Message:**
   - Inside the Logger class, there is a log method that takes a message as an argument. This method prints the message to the console and then emits the messageLogged event, providing an object with id and url.
5. **Module Export:**
   - The Logger class is exported from logger.js using module.exports. This makes it available to other files that require logger.js.
6. **Using Logger Class:**
   - In app.js, the Logger class is imported, and an instance of it is created (const logger = new Logger()).
   - A listener is registered for the messageLogged event using logger.on('messageLogged', (arg) => { ... }). This listener will be called whenever the messageLogged event is emitted, and arg will contain the accompanying data.
   - The log method is called on the logger instance, which prints the message and triggers the messageLogged event.

PDF, DOCX, PPT, TEXT file read using library in node

PDF File Parsing

Package Installation Command:

bash

```
npm install pdf-parse
```

Example Code:

javascript

```
// Import the built-in 'fs' (file system) module
const fs = require('fs');


// Import the 'pdf-parse' library for parsing PDF files
const pdf = require('pdf-parse');


// Specify the path to the PDF file
const filePath = '../Introduction to Node Note-01.pdf';


// Read the PDF file as a buffer
```

```
fs.readFile(filePath, (err, data) => {
    // Handle any errors that occur while reading the file
    if (err) throw err;

    // Parse the PDF data
    pdf(data).then(pdfData => {
        // Log the text content of the PDF to the console
        console.log(pdfData.text);
    }).catch(error => {
        // Handle any errors that occur while parsing the PDF
        console.error("Error reading file", error);
    });
});
```

Parsing Different Types of Files in Node.js

1. PDF Files

- **Library:** pdf-parse
- **Usage:** Extracts text content from PDF files.
- **Example:** The code snippet above demonstrates how to read and parse a PDF file.

2. DOCX Files

- **Package Installation Command:**

bash

```
npm install mammoth
```

- **Example Code:**

javascript

```
// Import the built-in 'fs' (file system) module
const fs = require('fs');

// Import the 'mammoth' library for parsing DOCX files
const mammoth = require('mammoth');

// Specify the path to the DOCX file
const filePath = 'example.docx';
```

```
// Read the DOCX file as a buffer
fs.readFile(filePath, (err, data) => {
    // Handle any errors that occur while reading the file
    if (err) throw err;

    // Parse the DOCX data to extract raw text
    mammoth.extractRawText({ buffer: data })
        .then(result => {
            // Log the text content of the DOCX to the console
            console.log(result.value);
        })
        .catch(error => {
            // Handle any errors that occur while parsing the DOCX
            console.error("Error reading file", error);
        });
});
```

3. PPT Files

- **Package Installation Command:**

```
bash
npm install pptx2json
```

- **Example Code:**

```
javascript
// Import the built-in 'fs' (file system) module
const fs = require('fs');

// Import the 'pptx2json' library for parsing PPTX files
const pptx2json = require('pptx2json');

// Specify the path to the PPTX file
const filePath = 'example.pptx';
```

```
// Parse the PPTX data to extract content
pptx2json(filePath).then(result => {
    // Log the parsed data from the PPTX file to the console
    console.log(result);
}).catch(error => {
    // Handle any errors that occur while parsing the PPTX
    console.error("Error reading file", error);
});
```

4. Text Files

- **Library:** Built-in fs module (no need to install)
- **Usage:** Reads plain text files.
- **Example Code:**

```
javascript
// Import the built-in 'fs' (file system) module
const fs = require('fs');

// Specify the path to the text file
const filePath = 'example.txt';

// Read the text file with UTF-8 encoding
fs.readFile(filePath, 'utf8', (err, data) => {
    // Handle any errors that occur while reading the file
    if (err) throw err;
```