Node.js

Module 4

ADC502 : Web Development

What is Node.js

- **Node.js** is an open-source, cross-platform runtime environment that allows developers to build server-side and network applications using JavaScript.
- It is designed to be event-driven and non-blocking, making it particularly well-suited for building scalable and high-performance applications

History

- Node.js was created by Ryan Dahl and was first released in 2009. It is built on the V8 JavaScript engine developed by Google for use in their Chrome web browser.
- Dahl's motivation for creating Node.js stemmed from the need for a more efficient way to handle I/O operations in web applications. Traditional server-side technologies often suffered from blocking I/O, which limited their ability to handle a large number of concurrent connections.
- Node.js introduced an event-driven, non-blocking architecture, making it possible to handle many concurrent connections with relatively low resource consumption. This innovation quickly gained popularity in the developer community.

Significance in Modern Web Development

Node.js has become a critical technology in modern web development for several reasons:

1. **Efficient I/O Handling:** Node.js excels at handling I/O operations, such as reading/writing to files and making network requests, without blocking the execution of other code. This non-blocking approach allows applications to be highly responsive and handle many connections simultaneously.
2. **JavaScript Everywhere:** Node.js enables developers to use JavaScript on both the server and client sides of web applications. This unification of the programming language streamlines development, as developers can work with a single language for the entire stack.

Significance in Modern Web Development

3. **Rich Package Ecosystem:** Node.js comes with a vast ecosystem of open-source packages available through the Node Package Manager (npm). This package ecosystem accelerates development by providing readily available solutions for various tasks and functionalities.
4. **Scalability:** Node.js is designed with scalability in mind, making it suitable for building real-time applications like chat applications, online gaming platforms, and streaming services.
5. **Community and Support:** Node.js has a large and active community of developers, which means abundant resources, libraries, and support for troubleshooting issues and finding solutions.

Installation and Setup

Installation steps

1. Choose a Version:

- Node.js has two primary versions: LTS (Long-Term Support) and Current. For most use cases, it's recommended to install the LTS version as it provides stability and long-term support. Visit the official Node.js website (https://nodejs.org/) to see which version is currently LTS.

2. Download the Installer:

- Visit the official Node.js website (https://nodejs.org/) in your web browser.
- On the homepage, you'll see the LTS version prominently displayed. Click on it to download the installer for your operating system (Windows, macOS, or Linux).

Installation steps

3. Install Node.js on Windows:

- For Windows, download the MSI installer. Double-click the downloaded MSI file, follow the installation wizard, and accept the default settings. Node.js and npm (Node Package Manager) will be installed.

4. Install Node.js on macOS:

- For macOS, download the macOS installer. Double-click the downloaded package file, follow the installation prompts, and Node.js and npm will be installed.

Installation steps

5. Install Node.js on Linux (Debian/Ubuntu):

- For Debian-based Linux distributions like Ubuntu, you can use the package manager to install Node.js:

```
sudo apt update
sudo apt install nodejs npm
```

- This will install both Node.js and npm.

6. Verify the Installation:

- To ensure that Node.js and npm were installed successfully, open your computer's command prompt or terminal and run the following commands:

```
node -v
npm -v
```

- These commands should display the installed Node.js and npm versions, respectively.

Node.js : Hello World

"Hello World"

Step 1: Create a JavaScript File

- Open your text editor (such as Visual Studio Code, Sublime Text, or Notepad) and create a new file. Save it with a .js extension, like hello.js.

Step 2: Write the "Hello World" Code

- Inside the hello.js file, write the following code:

```
console.log("Hello, World!");
```

- This code uses the `console.log` function to print "Hello, World!" to the console.

"Hello World"

Step 3: To run the Node.js script, follow these steps:

- Open your computer's command prompt or terminal.
- Navigate to the directory where you saved hello.js using the cd command. For example:

  ```
  cd path/to/your/directory
  ```

- Once you're in the correct directory, run the Node.js script using the node command followed by the filename (without the .js extension). In this case:

  ```
  node hello
  ```

- After running the command, you should see the output "Hello, World!" displayed in the console.
- P.N. Code is executed in a non-browser environment.

Event Driven Programming

Event Driven Programming

- The concept of event-driven programming is fundamental to Node.js and is a key reason for its popularity and success.
- Event-driven programming in Node.js is based on the idea of handling events and asynchronous operations efficiently, allowing applications to remain responsive and scalable.

Event Driven Programming : Key Concepts

1. Event and Event Emitters:

- At the core of event-driven programming in Node.js are events and event emitters. An event is a signal that something has happened.
- It can be anything from a user clicking a button to a file being read or a network request completing.
- Event emitters are objects that can emit (produce) events.
- In Node.js, the EventEmitter class is a built-in module that provides the capability to create objects that emit events.

Event Driven Programming : Key Concepts

2. Event Listeners:

- Event-driven programming involves attaching event listeners (also known as subscribers or handlers) to event emitters. Event listeners are functions that are executed when a specific event occurs.
- These listeners "listen" for events of interest and respond to them when they happen.

Event Driven Programming : Key Concepts

3. Non-Blocking and Asynchronous:

- Node.js is designed to be non-blocking and asynchronous.
- This means that while one part of the code is waiting for an I/O operation (e.g., reading a file or making a network request), the program can continue executing other tasks instead of blocking and waiting for the operation to finish.
- Asynchronous code in Node.js is typically written using callback functions, Promises, or async/await, which allow you to define what should happen once the asynchronous operation is complete.

Event Driven Programming : Key Concepts

4. Event Loop:

- Node.js utilizes an event loop to manage the execution of asynchronous code and event handling.
- The event loop is the core of Node.js, responsible for handling I/O operations, timers, and events efficiently.
- The event loop continuously checks for pending events and executes their associated event listeners.

Example of Event-Driven Code

```
const EventEmitter = require('events');
// Create an event emitter instance
const myEmitter = new EventEmitter();
// Define an event listener
myEmitter.on('myEvent', () => {
  console.log('Event occurred!');
});
// Emit the event
myEmitter.emit('myEvent');
```

- In this example, we create an event emitter (myEmitter) and define an event listener for the event 'myEvent’.
- When we emit 'myEvent', the associated listener executes, and "Event occurred!" is logged to the console.

Event Driven Programming : Use Cases

- Event-driven programming is particularly useful for building real-time applications, web servers, chat applications, IoT devices, and applications that need to handle a large number of concurrent connections efficiently.
- Event-driven programming is a powerful paradigm that allows Node.js to excel in handling I/O-intensive tasks and building scalable, responsive, and high-performance applications.
- Understanding how to work with events, event emitters, and asynchronous code is essential when developing with Node.js.

Event Driven Programming : Examples

- eventExample.js
- notificationSystemClass.js
- notificationSystemSubclass.js

Asynchronous Programming

Asynchronous programming

- Asynchronous programming in Node.js is a fundamental concept that allows you to perform non-blocking I/O operations, handle concurrency, and build responsive, high-performance applications.
- Asynchronous programming is at the heart of Node.js, and it's achieved through various techniques and patterns

Asynchronous programming : Use cases

- Reading a File Asynchronously:
  - fs.readFile('file.txt', 'utf8', (err, data) => { /* Handle file contents */ });
- Making an HTTP Request:
  - http.get('http://example.com', (response) => { /* Handle HTTP response */ });
- Querying a Database:
  - db.query('SELECT * FROM users', (results) => { /* Handle database query results */ });
- Sending Email:
  - mailer.sendEmail('recipient@example.com', 'Hello!', (result) => { /* Handle email sending status */ });
- Fetching Data from an API:
  - fetch('https://api.example.com/data').then((data) => { /* Handle API response */ });

Asynchronous programming : Use cases

- Handling User Input in Web Applications:
  - app.post('/submit', (req, res) => { /* Handle form submission asynchronously */ });
- Real-time Chat Application:
  - socket.on('message', (message) => { /* Handle real-time chat messages */ });
- Scheduling Tasks:
  - setTimeout(() => { /* Execute a task after a delay */ }, 2000);
- Streaming Data Processing:
  - stream.pipe(destinationStream); // Stream data asynchronously between sources and destinations.
- Parallel Processing:
  - Promise.all([asyncTask1(), asyncTask2()]).then((results) => { /* Handle results of multiple async tasks */ });

Asynchronous programming : Techniques

- **Callbacks:** Callback functions are a common way to handle asynchronous operations in Node.js. A callback is a function passed as an argument to another function. It gets executed when the operation it's associated with is completed. Callbacks are commonly used for I/O operations like reading files, making HTTP requests, or querying databases.
- Example:

```
fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }
  console.log('File contents:', data);
});
```

Asynchronous programming : Techniques

- **Promises:** Promises provide a more structured and readable way to work with asynchronous code.
- They represent a value (or error) that might be available in the future.
- Promises allow you to chain operations and handle errors in a more organized manner.
- Example:

```
const fs = require('fs').promises;
fs.readFile('file.txt', 'utf8')
  .then((data) => {
    console.log('File contents:', data);
  })
  .catch((err) => {
    console.error('Error reading file:', err);
  });
```

Asynchronous programming : Techniques

- **Async/Await:** The async/await syntax is built on top of Promises and provides a more synchronous-like way of writing asynchronous code.
- It allows you to write asynchronous code that looks similar to traditional synchronous code, making it easier to reason about and maintain.
- Example:

```
async function readFileAsync() {
  try {
    const data = await fs.readFile('file.txt', 'utf8');
    console.log('File contents:', data);
  } catch (err) {
    console.error('Error reading file:', err);
  }
}
readFileAsync();
```

Asynchronous programming : Techniques

- **Event Emitters:** Node.js's event-driven architecture is based on Event Emitters, which allow you to create and handle custom events.
- You can attach event listeners to an emitter, and when an event is emitted, the associated listeners are executed asynchronously.
- Example:

```
const EventEmitter = require('events');
const emitter = new EventEmitter();
emitter.on('customEvent', (data) => {
  console.log('Custom event data:', data);
});
emitter.emit('customEvent', { message: 'Hello, world!' });
```

Asynchronous programming : Techniques

- **Timers:** Node.js provides functions like setTimeout and setInterval for scheduling code to run after a specified delay or at regular intervals. These functions are asynchronous and allow you to perform tasks at specific times without blocking the main event loop.
- Example:

```
setTimeout(() => {
  console.log('Delayed operation');
}, 2000); // Execute after 2 seconds
```

Asynchronous programming

- Concurrency Control
  - Asynchronous programming is crucial for handling concurrent operations efficiently.
  - Node.js's event loop enables it to handle multiple asynchronous tasks concurrently without blocking, making it suitable for real-time applications and web servers.

Where Asynchronous Programming is used

- Netflix:
  - Asynchronous programming in Node.js helps Netflix efficiently fetch and serve streaming content to users. It allows for non-blocking I/O operations when streaming video data and handling concurrent requests.
- Uber:
  - Uber relies on Node.js to handle real-time updates of drivers' locations and user ride requests. Asynchronous operations are crucial for tracking and responding to location data in real time.
- LinkedIn:
  - LinkedIn utilizes asynchronous programming to provide real-time notifications to users when they receive connection requests, messages, or updates from their network connections.

Where Asynchronous Programming is used

- PayPal:
  - PayPal uses asynchronous processing in Node.js to handle payment transactions and implement real-time fraud detection and prevention mechanisms, ensuring secure and responsive financial services.
- eBay:
  - eBay employs Node.js for real-time bidding updates during auctions and to handle concurrent user interactions on product listings and purchases.
- Walmart:
  - Walmart's e-commerce platform benefits from asynchronous operations in Node.js to manage real-time inventory updates, optimize product recommendations, and ensure responsive user experiences during online shopping.

Where Asynchronous Programming is used

- Medium:
  - Medium uses asynchronous programming for real-time comment posting and updating, allowing users to see comments from other readers as they are posted, enhancing the sense of community and engagement.
- Trello:
  - Trello's collaborative project management tool leverages Node.js for real-time board updates and synchronization of changes made by multiple team members, ensuring that everyone has the latest information.
- Slack:
  - Slack relies on Node.js for its messaging infrastructure to enable real-time chat and collaboration among team members. Asynchronous operations are essential for delivering messages and notifications instantly.

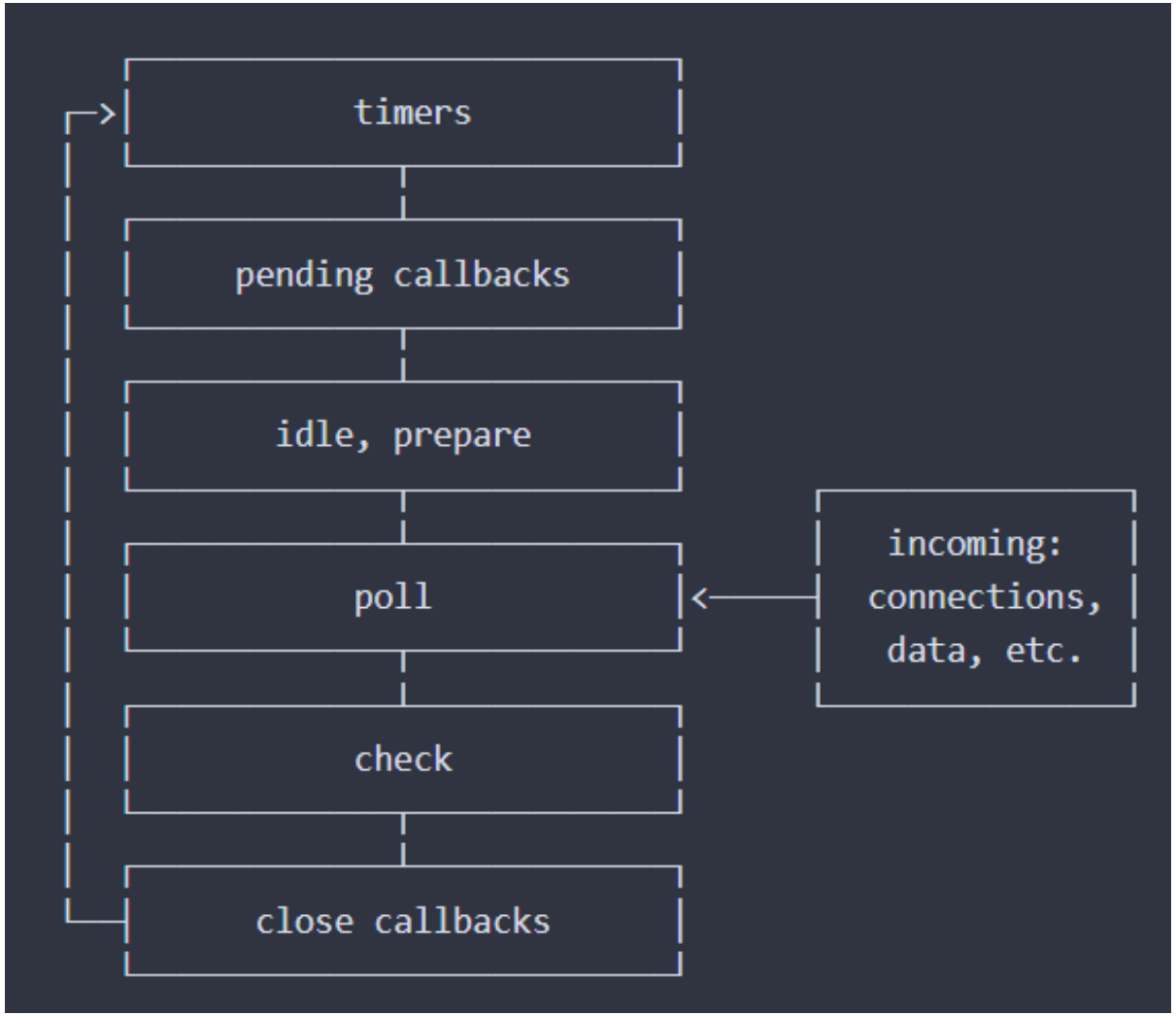
Event Loop

Discussed briefly. Refer Documentation (link) for more details.

What is Event Loop

- The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.
- Since most modern kernels are multi-threaded, they can handle multiple operations executing in the background. When one of these operations completes, the kernel tells Node.js so that the appropriate callback may be added to the poll queue to eventually be executed.

Event Loop



Each box will be referred to as a "phase" of the event loop.

Event Loop phases

- The event loop consists of multiple phases, and during each iteration of the loop, it processes events and tasks in a specific order:
  - **Timers**: Handles timer-related tasks, executing callbacks scheduled using setTimeout or setInterval.
  - **Pending Callbacks**: executes I/O callbacks deferred to the next loop iteration.
  - **Idle, Prepare**: Internal phases that are less commonly used in user code.
  - **Poll**: Retrieves new I/O events and executes their callbacks. This is where most of the work happens in the event loop.
  - **Check**: Executes `**setImmediate**` callbacks that are scheduled for the immediate execution phase.
  - **Close Callbacks**: Handles close event callbacks.

Event Loop

- Each phase has a FIFO queue of callbacks to execute.
- The event loop processes events in a round-robin fashion, continuously iterating through its phases. It checks for pending events or tasks in each phase and executes them if they are ready.
- While each phase is special in its own way, generally, when the event loop enters a given phase, it will perform any operations specific to that phase, then execute callbacks in that phase's queue until the queue has been exhausted or the maximum number of callbacks has executed. When the queue has been exhausted or the callback limit is reached, the event loop will move to the next phase, and so on.

Modules

What Are Modules?

- In Node.js, modules are a fundamental concept that allows you to organize and structure your code into reusable, self-contained units.
- Modules make it easier to manage code dependencies, improve code maintainability, and facilitate code sharing between different parts of your application.
- Modules are individual units of code that encapsulate functionality.
- Each module typically represents a single file containing variables, functions, or classes related to a specific task or feature.

Why Use Modules?

- Code Organization:
  - Modules help organize your code into manageable pieces. Instead of writing all your code in a single file, you can split it into modules based on functionality, making it easier to understand and maintain.
- Reusability:
  - Modules promote code reuse. You can use the same module in multiple parts of your application, reducing duplication and ensuring consistency.
- Encapsulation:
  - Modules encapsulate code, which means the internal details of a module are hidden from the outside. Only the functions or variables explicitly exported by the module are accessible from other parts of your code.

Creating Modules

- To create a module in Node.js, you typically create a JavaScript file (e.g., myModule.js) and define your variables, functions, or classes inside it.
- You can export specific functions, variables, or objects from a module using the module.exports or exports object.
- For example:

```
// myModule.js
const greeting = 'Hello, ';

function sayHello(name) {
  return greeting + name;
}

module.exports = {
  sayHello,
  greeting,
};
```

Using Modules

- To use a module in your Node.js application, you use the require function to import it.
- For example:

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

console.log(myModule.sayHello('Alice')); // Outputs: Hello, Alice
console.log(myModule.greeting); // Outputs: Hello,
```

- When you require a module, Node.js loads the module and returns the object or value that you exported from it.

Core Modules vs. Custom Modules

- Node.js provides core modules (e.g., fs for file system operations, http for creating web servers) that you can use out of the box.
- You can also create your custom modules to encapsulate application-specific functionality.

Modularization Best Practices

- Keep modules small and focused on a single responsibility.
- Use descriptive module names and organize them into directories.
- Avoid circular dependencies (where Module A depends on Module B, and Module B depends on Module A).

Module Resolution

- Module resolution is the process by which Node.js locates and loads modules or files when you use the require() function to import them in your code.
- It's a critical part of how Node.js manages dependencies and allows you to organize and use code from different files and packages effectively.
- Node.js follows a specific module resolution algorithm to locate and load modules. It searches for modules in the local directory, then in the node_modules directory, and so on.

Module Resolution

- Core Modules:
  - Node.js has a set of built-in core modules (e.g., fs, http, path) that can be required without specifying a path. For example, you can require the fs module like this:

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

- Relative Paths:
  - When you require a module using a relative path (e.g., ./myModule or ../utils/helper), Node.js resolves the module relative to the location of the current module (the module that is requiring the other module).
  - It starts by looking for the module in the same directory as the current module.

Module Resolution

- Node Modules Directory (node_modules):
  - If Node.js does not find the module in the current directory, it checks the node_modules directory in the current module's directory.
  - If the module is not found there, Node.js continues to search for it in parent directories, climbing up the directory tree until the root directory is reached.
- Global Modules Directory:
  - Node.js also checks a global directory for globally installed modules.
  - However, this is less common and not recommended for most use cases.

Module Resolution

- File Extensions:
  - When resolving a module, Node.js tries to match the module name to a file with the .js, .json, or .node extension. If you require a module without specifying the file extension (e.g., require('./myModule')), Node.js will try these extensions in that order.
- package.json and main Field:
  - If you require a directory instead of a specific file (e.g., require('./myDirectory')), Node.js looks for a package.json file within that directory. If it finds one, it checks the main field in the package.json file to determine which file to load as the entry point for the module.

Module Resolution

- Index.js as Default:
  - If no specific file is specified when requiring a directory, Node.js will attempt to load an index.js file by default. For example, require('./myDirectory') would try to load ./myDirectory/index.js.
- Node.js Core Modules and Globals:
  - If the module name is neither a relative path nor a module installed in node_modules, Node.js checks if it matches one of the built-in core modules or global variables. For example, require('http') would load the built-in http module.

Module Resolution

- Module Caching:
  - Once a module is loaded, Node.js caches it. If the same module is required again elsewhere in the application, Node.js returns the cached module instead of re-reading and re-executing the code. This caching mechanism helps improve performance and ensures that modules are singletons.

Understanding module resolution is essential for managing dependencies in Node.js projects.

REPL

Discussed briefly. Refer Documentation (link) for more details.

REPL

- REPL stands for Read-Evaluate-Print-Loop.
- REPL is a programming language environment (basically a console window) that takes single expression as user input and returns the result back to the console after execution.
- The REPL session provides a convenient way to quickly test simple JavaScript code.

REPL

Try in your terminal

- > node
- >

- The REPL is waiting for us to enter some JavaScript code, to be more precise.
- Start simple and enter

```
> console.log('test')
test
undefined
>
```

REPL

- Try

> 5 === '5'

false

>

- The _ special variable
  - If after some code you type _, that is going to print the result of the last operation.
- The Up arrow key
  - If you press the up arrow key, you will get access to the history of the previous lines of code executed in the current, and even previous REPL sessions.

Dot commands

- The REPL has some special commands, all starting with a dot .. They are
  - .help: shows the dot commands help
  - .editor: enables editor mode, to write multiline JavaScript code with ease. Once you are in this mode, enter ctrl-D to run the code you wrote.
  - .break: when inputting a multi-line expression, entering the .break command will abort further input. Same as pressing ctrl-C.
  - .clear: resets the REPL context to an empty object and clears any multi-line expression currently being input.
  - .load: loads a JavaScript file, relative to the current working directory
  - .save: saves all you entered in the REPL session to a file (specify the filename)
  - .exit: exits the repl (same as pressing ctrl-C two times)

Detailed Article

- For more refer : How to use the Node.js REPL
- https://nodejs.dev/en/learn/how-to-use-the-nodejs-repl/#:~:text=Note%3A%20REPL%20stands%20for%20Read,quickly%20test%20simple%20JavaScript%20code.

(Earlier examples are copied from this article)

REPL Module

- The node:repl module provides a Read-Eval-Print-Loop (REPL) implementation that is available both as a standalone program or includible in other applications. It can be accessed using:

- const repl = require('node:repl');

File System Module

Discussed briefly. Refer Documentation (link) for more details.

File System Module

- The File System module in Node.js, often referred to as **fs**, provides a set of methods and functionality for working with the file system.
- You can use it to read, write, delete, rename, and manipulate files and directories.

Reading and Writing a File with the fs Module

- In this example, we'll create a Node.js program that reads the contents of a text file, appends new data to it, and then reads the updated file contents.

1. Create a text file (example.txt):

- Before running the Node.js program, create a text file named example.txt in the same directory as your Node.js script. You can add some initial text to it.

2. Create a Node.js script (fs-example.js):

- Create a Node.js script (e.g., fs-example.js) and write the following code to demonstrate reading and writing a file using the fs module:

Example

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

// Define the file path
const filePath = 'example.txt';

// Step 1: Read the contents of the file asynchronously
fs.readFile(filePath, 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }

  console.log('File contents before appending:');
  console.log(data);
```

```
// Step 2: Append new data to the file asynchronously
 const newData = '\nThis is some appended text.';
 fs.appendFile(filePath, newData, (err) => {
  if (err) {
   console.error('Error appending to file:', err);
   return;
  }
  console.log('Data appended to the file.');

  // Step 3: Read the updated file contents
  fs.readFile(filePath, 'utf8', (err, updatedData) => {
   if (err) {
    console.error('Error reading updated file:', err);
    return;
   }
   console.log('File contents after appending:');
   console.log(updatedData);
  });
 });
});
```

Example

3. Run the Node.js script:

- Open your terminal or command prompt, navigate to the directory containing the fs-example.js script and the example.txt file, and then run the script using the node command:

```
node fs-example.js
```

4. Expected Output:

- The program should read the initial contents of the example.txt file, append new data to it, and then read and display the updated file contents. The output will look something like this:

```
File contents before appending:
This is the initial text in the file.
Data appended to the file.
File contents after appending:
This is the initial text in the file.
This is some appended text.
```

Buffers

Discussed briefly. Refer Documentation (link) for more details.

Buffers

- Buffers in Node.js are objects used to represent binary data in a more efficient and low-level way than JavaScript's built-in String and Array objects.
- Buffers are particularly useful when working with binary data, such as reading from or writing to files, interacting with network protocols, or dealing with binary data streams.
- Buffer objects are used to represent a fixed-length sequence of bytes. Many Node.js APIs support Buffers.

Buffers

- Binary Data Representation:
  - Buffers are designed to store binary data, which can be in the form of raw bytes, characters, or other binary formats. They provide a way to manipulate and work with binary data directly in memory.
- Fixed-Size:
  - Buffers have a fixed size when created, and this size cannot be changed after creation. This is in contrast to arrays or strings, which can dynamically grow or shrink.
- Efficiency:
  - Buffers are more memory-efficient and faster for certain operations involving binary data compared to using JavaScript strings or arrays. They are optimized for tasks like reading and writing files or working with network protocols.

Creation of Buffers

You can create a Buffer in Node.js using the Buffer.from(), Buffer.alloc(), or new Buffer() constructor. For example:

```
// Creating a Buffer from a string
const bufferFromString = Buffer.from('Hello, world!', 'utf8');

// Creating an empty Buffer of a specific size
const emptyBuffer = Buffer.alloc(16);
```

Accessing Buffer Data

Buffers store data as a sequence of bytes. You can access individual bytes within a Buffer using index notation. For example:

```
// Accessing the first byte
const firstByte = bufferFromString[0];
```

Modifying Buffers

- Buffers can be modified by assigning new values to their individual bytes.
- However, it's important to note that Buffer operations are performed at the byte level, and modifications must adhere to valid binary values.

```
// Modifying a byte in a Buffer
bufferFromString[0] = 72; // ASCII value for 'H'
```

Conversion to Strings

- You can convert a Buffer to a string using the toString() method, specifying the character encoding. For example:

```
const buffer = Buffer.from('Hello, world!', 'utf8');
const str = buffer.toString('utf8');
```

Buffers

- Buffer Pools:
  - Node.js uses a buffer pool for managing and recycling memory used for Buffers. This can help reduce the overhead of memory allocation and deallocation when working with Buffers.
- Common Use Cases:
  - Buffers are commonly used for tasks like reading and writing files, processing network data, working with streams, and interacting with binary protocols (e.g., TCP, UDP).
- Buffer Size and Memory Management:
  - Buffers should be used with care, especially when dealing with large binary data. Allocating large Buffers can consume a significant amount of memory, so it's important to manage memory effectively.

```
// Example : Creating a Buffer from a string
const bufferFromString = Buffer.from('Hello, Node.js!', 'utf8');

// Accessing and modifying individual bytes
bufferFromString[0] = 72; // ASCII value for 'H'
bufferFromString[1] = 105; // ASCII value for 'i'

// Printing the modified Buffer as hexadecimal values
console.log('Modified Buffer as Hex:', bufferFromString.toString('hex'));

// Creating an empty Buffer and filling it
const emptyBuffer = Buffer.alloc(10); // Creating an empty buffer of 10 bytes
emptyBuffer.fill(65); // Filling the buffer with ASCII value 'A'

// Printing the filled Buffer as a string
console.log('Filled Buffer as String:', emptyBuffer.toString());

// Combining Buffers
const combinedBuffer = Buffer.concat([bufferFromString, emptyBuffer]);

// Printing the combined Buffer as a string
console.log('Combined Buffer as String:', combinedBuffer.toString('utf8'));
```

Streams

Discussed briefly. Refer Documentation (link) for more details.

Streams

- Streams in Node.js are a powerful and efficient way to read from or write to data sources or destinations, especially when dealing with large volumes of data.
- They provide an abstraction that allows you to work with data in chunks, rather than loading it all into memory at once, making them memory-efficient and well-suited for various tasks like file I/O, network communication, data processing, and more.
- Node.js provides a built-in 'stream' module with several classes and methods for working with streams.

Stream types in Node.js

- Readable Streams:
  - Readable streams represent a source from which data can be read, such as a file, an HTTP request, or a network socket. They allow you to read data in chunks as it becomes available, rather than reading the entire dataset at once.
- Writable Streams:
  - Writable streams represent a destination to which data can be written, such as a file, an HTTP response, or a network socket. They allow you to write data in chunks, which can be especially useful for sending large files or streaming responses to clients.

Stream types in Node.js

- Duplex Streams:
  - Duplex streams combine both readable and writable capabilities, allowing you to both read from and write to a data source or destination. An example of a duplex stream is a network socket, where data can be both sent and received.
- Transform Streams:
  - Transform streams are a type of duplex stream that allows you to modify or transform data as it passes through the stream. They are often used for data manipulation, such as compressing or encrypting data during streaming.

How to use streams in Node.js

1. Creating Streams:

- You can create streams using the stream module's classes.
- For example, you can create a readable stream from a file using fs.createReadStream() or a writable stream to a file using fs.createWriteStream().

2. Piping Streams:

- One of the most powerful features of streams is the ability to pipe data from one stream to another using the pipe() method.
- This allows you to easily transfer data from a readable stream to a writable stream, often without buffering the entire dataset in memory.

How to use streams in Node.js

3. Handling Events:

- Streams emit events that you can listen to, such as 'data' (when data is available to read), 'end' (when reading is complete), and 'finish' (when writing is complete).
- You can attach event handlers to respond to these events.

4. Flow Control:

- Streams have built-in flow control mechanisms to handle the rate at which data is read from or written to the stream.
- This helps prevent memory overload when dealing with slow consumers or fast producers.

5. Custom Streams:

- You can create custom readable, writable, or transform streams by extending the built-in stream classes and implementing the necessary methods.

```
// Stream Example : Readable and Writable streams
const fs = require('fs');

// Create a readable stream from the source file
const sourceStream = fs.createReadStream('source.txt', 'utf8');

// Create a writable stream to the destination file
const destinationStream = fs.createWriteStream('destination.txt', 'utf8');

// Pipe the data from the source stream to the destination stream
sourceStream.pipe(destinationStream);

// Listen for the 'finish' event to know when the copying is complete
destinationStream.on('finish', () => {
  console.log('File copied successfully.');
});

// Handle errors
sourceStream.on('error', (err) => {
  console.error('Error reading source file:', err);
});

destinationStream.on('error', (err) => {
  console.error('Error writing to destination file:', err);
});
```

```
// Piping and Transform Example
// To read a file ‘input.txt’ and put it in UPPERCASE on the console
Const { Transform } = require('stream');
// Define a custom Transform stream
class UppercaseTransform extends Transform {
  constructor() {
    super();
  }

  _transform(chunk, encoding, callback) {
    // Convert the chunk (text) to uppercase and push it to the output
stream
    this.push(chunk.toString().toUpperCase());
    callback();
  }
}
// Create an instance of the custom Transform stream
const uppercaseStream = new UppercaseTransform();
```

```
// Create a readable stream with some text
const input = require('fs').createReadStream('input.txt', 'utf8');

// Handle errors on the input stream
input.on('error', (err) => {
  console.error('Error reading input:', err);
});

// Handle errors on the custom Transform stream
uppercaseStream.on('error', (err) => {
  console.error('Error in custom stream:', err);
});

// Pipe the input stream through the custom Transform stream and
print to the console
input.pipe(uppercaseStream).pipe(process.stdout);
```

```
// Custom Stream to add numbers passed to it

const { Writable } = require("stream");

class SumWritableStream extends Writable {
  constructor(options) {
    super({ objectMode: true }); // Enable object mode to handle numbers
    this.total = 0; // Initialize the sum
  }

  // Implement the _write method to handle incoming data
  _write(number, encoding, callback) {
    if (typeof number === "number" && !isNaN(number)) {
      this.total += number; // Add the number to the sum
    }
    callback(); // Callback to indicate that the data has been processed
  }
}
```

Custom Streams

```
const sumStream = new SumWritableStream();

// Set up an event listener for the 'finish' event
sumStream.on("finish", () => {
  console.log("Sum:", sumStream.total);
});

// Write some numbers to the stream
sumStream.write(5);
sumStream.write(10);
sumStream.write(15);

// Signal the end of writing
sumStream.end();
```

Networking module

Discussed briefly. Refer Documentation (link) for more details.

Networking (related) modules

- In Node.js, the networking module primarily refers to the built-in ***net*** and ***http*** modules, which provide the foundation for creating network-related applications, including both low-level TCP/HTTP server and client operations.
- These modules enable you to establish network connections, handle data transfer over networks, and build various network-based applications.

Networking (related) modules

- Key networking modules in Node.js are
  - net Module
    - For TCP/IPC Server and Client
  - http Module
    - For HTTP Server and Client)
  - https Module
    - For HTTPS Server and Client
  - dgram Module
    - For UDP Server and Client

net Module

- The net module allows you to create TCP (Transmission Control Protocol) servers and clients. You can also use it for inter-process communication (IPC) on the same machine.
- Key classes in the net module include net.Server for creating TCP servers and net.Socket for managing TCP client connections.
- You can use the net module for building custom network protocols, chat applications, real-time multiplayer games, and more.

http Module

- The http module is used for creating HTTP (Hypertext Transfer Protocol) servers and clients, which are essential for building web applications and RESTful APIs.
- Key classes include http.Server for creating HTTP servers and http.ClientRequest and http.ClientResponse for making HTTP requests and handling responses.
- The http module simplifies common HTTP-related tasks, such as serving web pages, handling HTTP requests and responses, and creating web services.

https Module

- The https module is an extension of the http module that provides support for secure HTTP (HTTPS) connections using SSL/TLS encryption.
- It allows you to create HTTPS servers and clients, making it suitable for building secure web applications, e-commerce sites, and any application requiring secure data transmission.

dgram Module

- The dgram module allows you to work with UDP (User Datagram Protocol) sockets for low-level, message-oriented communication.
- It is useful for scenarios requiring real-time data exchange, such as online gaming, VoIP applications, and network monitoring.

Where to use?

- These networking modules can be used to :
  - Create servers that listen for incoming network connections.
  - Establish network connections to remote servers or services.
  - Exchange data over the network, handle data transmission, and respond to incoming data.
  - Implement custom network protocols or use established protocols like HTTP and HTTPS.
  - Secure network communication with SSL/TLS encryption (HTTPS).
  - Build networked applications that serve content, provide services, or facilitate real-time communication.

Web module

Web module

- Node.js does not have a specific "web module" in its core modules.
- Instead, building web applications in Node.js typically involves using the http and https modules, as well as various third-party libraries and frameworks.
- These modules can be used to create web servers and handle HTTP requests and responses.

Example

- To create a basic web server (using http module) that listens for incoming HTTP requests and sends responses.

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

End