UNIT-II

Node.js:

Working with JSON in Node.js

JSON (JavaScript Object Notation) is a lightweight data-interchange format that is easy to read and write for humans and machines. It's widely used in APIs, configuration files, and data transmission over the web. Node.js provides native support for working with JSON data, making it easy to handle data in a JavaScript-friendly format.

In Node.js, you will frequently deal with JSON in tasks like reading and writing files, processing API responses, or handling configurations.

1. What is JSON?

JSON is a **text format** that represents structured data in a key-value pair format. It is language-independent but is used extensively with JavaScript.

Here's a basic example of JSON data:

```
{
  "name": "John Doe",
  "age": 30,
  "email": "johndoe@example.com"
}
```

This JSON object represents a person with a name, age, and email.

2. Parsing JSON (Converting JSON to JavaScript Objects)

When you receive JSON data (for example, from a file, API, or a network request), it is usually in a string format. In JavaScript, you need to **parse** the JSON string into a JavaScript object to work with the data.

Using JSON.parse()

JSON.parse() is used to convert a JSON string into a JavaScript object.

```
const jsonString = '{"name": "John", "age": 30}';
const obj = JSON.parse(jsonString);

console.log(obj);
console.log(obj.name);  // Accessing properties
```

Output:

```
{ name: 'John', age: 30 }
John
```

- JSON.parse() takes a JSON string and returns the corresponding JavaScript object. In the example above, jsonString is parsed into the object obj, allowing you to access its properties.

3. Stringifying JavaScript Objects (Converting Objects to JSON)

In many scenarios, you'll want to convert a JavaScript object back into a JSON string, especially when sending data over the network (e.g., in HTTP requests or saving it to a file).

Using JSON.stringify()

JSON.stringify() is used to convert a JavaScript object into a JSON string.

```
const obj = { name: "Alice", age: 25 };
const jsonString = JSON.stringify(obj);

console.log(jsonString);
```

Output:

```
{"name":"Alice","age":25}
```

- JSON.stringify() converts the object obj into a JSON-formatted string.

4. Working with JSON Files in Node.js

Node.js allows you to read from and write to JSON files, which is useful for configuration files, storing data, and more. You can use the fs (file system) module to handle file operations.

Reading JSON Data from a File

Here’s an example of how to read a JSON file asynchronously using fs.readFile() and parse it into a JavaScript object:

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

// Reading a JSON file
fs.readFile('data.json', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }

  // Parsing the JSON data
  const jsonData = JSON.parse(data);
  console.log(jsonData);
});
```

Explanation:

- **fs.readFile()**: Reads the file asynchronously.
- **JSON.parse()**: Converts the file content (string) into a JavaScript object.

Reading JSON File Synchronously

You can also read files synchronously using fs.readFileSync():

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

// Synchronously reading a JSON file
const data = fs.readFileSync('data.json', 'utf8');
const jsonData = JSON.parse(data);
console.log(jsonData);
```

Writing JSON Data to a File

If you need to write data back to a file, you can use fs.writeFile() or fs.writeFileSync().

```
const fs = require('fs');
const person = {
  name: 'Sarah',
  age: 28,
  email: 'sarah@example.com'
};

// Convert JavaScript object to JSON string
const jsonString = JSON.stringify(person);

// Writing JSON data to a file
fs.writeFile('person.json', jsonString, (err) => {
  if (err) {
    console.error('Error writing to file:', err);
  } else {
    console.log('File successfully written!');
  }
});
```

Explanation:

- **JSON.stringify()**: Converts the JavaScript object into a JSON-formatted string.
- **fs.writeFile()**: Writes the string to a file asynchronously.

Writing JSON Data Synchronously

If you need to write the JSON data synchronously, you can use fs.writeFileSync():

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

const person = {
  name: 'John',
  age: 35,
  email: 'john@example.com'
};

const jsonString = JSON.stringify(person);

// Writing to file synchronously
fs.writeFileSync('person.json', jsonString);
console.log('File written successfully!');
```

5. Using JSON with HTTP Requests

JSON is often used for exchanging data between a client and a server. When working with HTTP in Node.js, you can parse and send JSON data as part of the request/response.

Example: Sending JSON in HTTP Requests (Using http Module)

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

const options = {
  hostname: 'jsonplaceholder.typicode.com',
  port: 80,
  path: '/posts',
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  }
};

const data = JSON.stringify({
  title: 'foo',
  body: 'bar',
  userId: 1
});

const req = http.request(options, (res) => {
  let data = '';

  res.on('data', (chunk) => {
    data += chunk;
  });

  res.on('end', () => {
    console.log('Response:', data);
  });
});

req.on('error', (e) => {
  console.error('Problem with request:', e.message);
});
```

```
// Write the JSON data to the request body
req.write(data);
req.end();
```

Explanation:

- This example demonstrates making a **POST** request to a remote server with JSON data.
- **http.request()** is used to make the HTTP request, and the JSON data is sent in the body of the request.
- The **Content-Type: application/json** header specifies that the request body is JSON.

6. Handling JSON in API Responses

When working with APIs in Node.js, JSON is often the format used for both sending and receiving data. In this case, you'll receive the JSON data, parse it, and process it.

Example: Receiving JSON Data from an API

javascript
Copy

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

https.get('https://jsonplaceholder.typicode.com/posts/1', (res) => {
  let data = '';

  // Collect data chunks
  res.on('data', (chunk) => {
    data += chunk;
  });

  // Once data is fully received, parse it as JSON
  res.on('end', () => {
    const jsonData = JSON.parse(data);
    console.log('Received Data:', jsonData);
  });
});
```

Explanation:

- **https.get()** sends a GET request to fetch data from a URL.
- The response data is received in chunks, and once the entire response is received, it is parsed using **JSON.parse()**.

7. JSON Schema Validation

In some scenarios, you might need to ensure the received JSON data conforms to a specific schema. Although Node.js doesn't have built-in JSON schema validation, you can use libraries like **Ajv** or **Joi** for this purpose.

Example: Using Joi for JSON Validation

```
const Joi = require('joi');

// Define a schema for validation
const schema = Joi.object({
  name: Joi.string().required(),
  age: Joi.number().required(),
  email: Joi.string().email().required()
});

// Example JSON object
const user = {
  name: 'Alice',
  age: 25,
  email: 'alice@example.com'
};

// Validate the data
const { error } = schema.validate(user);
if (error) {
  console.log('Validation error:', error.details);
} else {
  console.log('Data is valid');
}
```

Explanation:

- **Joi** is a popular validation library that helps you ensure that incoming JSON data matches a predefined structure.
- You define a schema and then use **validate()** to check if the data is valid.

The **Buffer module** in Node.js is a powerful tool for working with raw binary data, such as when dealing with files, streams, or binary network protocols. A **Buffer** is a raw memory allocation outside the V8 JavaScript engine's heap, and it's used to represent binary data directly.

Buffers are especially useful for handling large amounts of data (e.g., image files, video files, binary data sent over a network) efficiently without converting them to strings.

Let's break down how to **use the Buffer module** in Node.js:

1. Creating Buffers

Node.js provides a global Buffer class that allows you to create Buffers in various ways.

a. From an Array or Array-like Object

You can create a buffer from an existing array or array-like object:

```
// Create a Buffer from an array of bytes
const buffer = Buffer.from([1, 2, 3, 4]);
console.log(buffer);  // Output: <Buffer 01 02 03 04>
```

b. From a String

You can also create a buffer from a string. You can specify the encoding (like 'utf8', 'ascii', 'base64', etc.):

```
// Create a Buffer from a string (UTF-8 encoding is default)
const bufferFromString = Buffer.from('Hello, Node.js!');
console.log(bufferFromString);  // Output: <Buffer 48 65 6c 6c 6f 2c 20 4e 6f 64 65 2e 6a 73 21>
```

c. Allocating a New Buffer

You can allocate a new Buffer with a specified size. Note that the content of the buffer is **initialized with zeroes** when using Buffer.alloc().

```
// Create an allocated buffer of 10 bytes (all zeroes)
const allocatedBuffer = Buffer.alloc(10);
console.log(allocatedBuffer);  // Output: <Buffer 00 00 00 00 00 00 00 00 00 00>
```

d. Unsafe Allocation (Not Recommended for Production)

You can use Buffer.allocUnsafe() for creating an uninitialized buffer. This method may provide better performance but is unsafe because it may contain old data. Always be cautious with this method.

```
// Create an uninitialized buffer (not recommended for security-critical tasks)
const unsafeBuffer = Buffer.allocUnsafe(10);
console.log(unsafeBuffer);
```

2. Reading and Writing Data to Buffers

Once you have a buffer, you can access and modify its content just like an array.

a. Reading Data from a Buffer

You can read the content of a buffer by indexing it (using zero-based indexing):

```
const buffer = Buffer.from('Hello, Node.js!');
console.log(buffer[0]);  // Output: 72 (ASCII code for 'H')

// You can convert it back to a string
console.log(buffer.toString('utf8'));  // Output: Hello, Node.js!
```

b. Writing Data to a Buffer

You can write data to a buffer using the write() method:

```
const buffer = Buffer.alloc(10);
buffer.write('Hello');
console.log(buffer);  // Output: <Buffer 48 65 6c 6c 6f 00 00 00 00 00>
```

Here, buffer.write('Hello') writes the string 'Hello' into the buffer at the start.

3. Manipulating Buffer Data

You can slice buffers, concatenate them, and perform other operations.

a. Slicing Buffers

You can use the slice() method to create a sub-buffer:

```
const buffer = Buffer.from('Hello, Node.js!');
const slicedBuffer = buffer.slice(0, 5);  // Get the first 5 bytes
console.log(slicedBuffer.toString());  // Output: Hello
```

b. Concatenating Buffers

You can concatenate multiple buffers into a single one using Buffer.concat():

```
const buffer1 = Buffer.from('Hello');
const buffer2 = Buffer.from(', ');
const buffer3 = Buffer.from('Node.js!');

const concatenatedBuffer = Buffer.concat([buffer1, buffer2, buffer3]);
console.log(concatenatedBuffer.toString());  // Output: Hello, Node.js!
```

4. Working with Buffer Length and Properties

You can get the length of a buffer using the length property:

```
const buffer = Buffer.from('Hello, Node.js!');
console.log(buffer.length);  // Output: 16 (number of bytes)
```

- **Note**: The length of the buffer is the number of bytes, not the number of characters (since some characters may take multiple bytes in certain encodings like UTF-8).

5. Buffer Encoding and Decoding

You can convert the content of a buffer back into a string using different encodings (UTF-8, ASCII, Base64, etc.):

```
const buffer = Buffer.from('Hello, Node.js!');

// Convert to string with UTF-8 encoding (default)
console.log(buffer.toString('utf8'));  // Output: Hello, Node.js!

// Convert to Base64 encoding
console.log(buffer.toString('base64'));  // Output: SGVsbG8sIE5vZGUuanMh

// Convert to ASCII encoding
console.log(buffer.toString('ascii'));  // Output: Hello, Node.js!
```

6. Example: Using Buffers for File I/O Operations

One of the common uses of buffers is when working with files. For instance, reading and writing binary files.

a. Reading a Binary File into a Buffer

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

// Read a file into a buffer (e.g., an image file)
fs.readFile('image.jpg', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }
  console.log('File read into buffer:', data);
  console.log('Buffer length:', data.length);  // Length of binary data
});
```

b. Writing Data from a Buffer to a File

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

// Create a buffer containing some binary data
const data = Buffer.from('This is some binary data!');

// Write the buffer to a file
fs.writeFile('output.bin', data, (err) => {
  if (err) {
    console.error('Error writing file:', err);
    return;
  }
  console.log('Data written to output.bin');
});
```

7. Performance Considerations

- Buffers are generally used to work with **binary data** (files, streams, network packets, etc.).
- Buffers are **more efficient** than strings when working with large amounts of binary data, as they avoid encoding and decoding steps (such as converting binary data into a UTF-8 string).
- Buffers do not have the overhead of strings, and because they are stored in memory outside of the V8 heap, they don't put pressure on garbage collection.

8. Buffer Pool (Node.js v10+)

Node.js maintains a pool of memory (also called a "buffer pool") for Buffer.allocUnsafe() allocations, which reduces memory fragmentation in long-running processes. However, care should be taken when using Buffer.allocUnsafe() since the memory may contain old data that could leak information.

The **Stream** module in Node.js is used for handling large amounts of data efficiently by reading or writing data in chunks. Instead of loading an entire file or data into memory, streams allow you to process the data piece by piece, which makes them very efficient for handling large files, network requests, or real-time data like video, audio, etc.

Streams in Node.js can be classified into **four main types**:

1. **Readable Streams**: For reading data.
2. **Writable Streams**: For writing data.
3. **Duplex Streams**: For both reading and writing data.
4. **Transform Streams**: A type of Duplex stream that can modify data as it is written and read.

Let's go through the basics of working with streams in Node.js:

1. Readable Streams

A **Readable stream** allows you to read data from a source (e.g., a file or a network response). Data is read in small chunks, and you can process it as it arrives.

a. Example: Reading a File with a Readable Stream

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

// Create a readable stream
const readableStream = fs.createReadStream('example.txt', 'utf8');

// When data is available, it will be read in chunks
readableStream.on('data', (chunk) => {
  console.log('Received chunk:', chunk);
});

// When the stream ends, this will be called
readableStream.on('end', () => {
  console.log('File reading finished.');
});

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

- **fs.createReadStream()** is a method that creates a readable stream to read data from a file.
- **'data' event** is fired when a chunk of data is available.
- **'end' event** is fired when the entire stream has been read.
- **'error' event** is fired if an error occurs during the read process.

b. Reading a Stream Using the pipe() Method

You can also use the **pipe()** method to automatically pipe the data from a readable stream into a writable stream. This is a common approach when working with files or network responses.

```
const fs = require('fs');
const readableStream = fs.createReadStream('example.txt');
const writableStream = fs.createWriteStream('copy.txt');

// Pipe the readable stream to the writable stream
readableStream.pipe(writableStream);

writableStream.on('finish', () => {
  console.log('Data successfully copied to copy.txt');
});
```

In this example:

- Data from **example.txt** is read through the **readable stream** and directly piped into the **writable stream** that writes to **copy.txt**.

2. Writable Streams

A **Writable stream** is used to write data to a destination (e.g., a file or a network socket). Data is written in chunks.

a. Example: Writing to a File with a Writable Stream

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

// Create a writable stream
const writableStream = fs.createWriteStream('output.txt');

// Write some data to the stream
writableStream.write('Hello, Node.js!\n');
writableStream.write('This is a writable stream.\n');

// Finalize the stream (indicating no more data will be written)
writableStream.end(() => {
  console.log('Data has been written to output.txt');
});
```

- **writableStream.write()** is used to write chunks of data to the stream.
- **writableStream.end()** is used to mark the end of the writable stream, signifying that no more data will be written.

b. Piping Data into a Writable Stream

You can pipe data from a readable stream into a writable stream, which is a great way to handle large files or data transfers without loading everything into memory.

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

const readableStream = fs.createReadStream('input.txt');
const writableStream = fs.createWriteStream('output.txt');
```

```
// Pipe data from readable stream to writable stream
readableStream.pipe(writableStream);

writableStream.on('finish', () => {
  console.log('Data successfully written to output.txt');
});
```

3. Duplex Streams

A **Duplex stream** allows both reading and writing. It can be used when you need to both receive and send data. An example of a duplex stream could be a network socket where you both read from and write to a server.

a. Example: Using a Duplex Stream

In this example, we'll simulate a duplex stream by creating a simple custom duplex stream:

```
const { Duplex } = require('stream');

class MyDuplexStream extends Duplex {
  constructor(options) {
    super(options);
    this.data = [];
  }

  // Implement the _read method (reading data)
  _read(size) {
    if (this.data.length > 0) {
      const chunk = this.data.shift();
      this.push(chunk);
    } else {
      this.push(null);  // No more data
    }
  }

  // Implement the _write method (writing data)
  _write(chunk, encoding, callback) {
    console.log('Received data:', chunk.toString());
    this.data.push(chunk);  // Store the chunk of data
    callback();  // Notify that the chunk has been processed
  }
}

// Create an instance of MyDuplexStream
const duplexStream = new MyDuplexStream();

// Write some data to the duplex stream
duplexStream.write('Hello, Duplex Stream!\n');
duplexStream.write('This is a custom duplex stream example.\n');
```

```
// Read data from the duplex stream
duplexStream.on('data', (chunk) => {
  console.log('Read chunk:', chunk.toString());
});

duplexStream.end();  // End the stream
```

4. Transform Streams

A **Transform stream** is a special kind of duplex stream where you can modify the data as it is being written and read. This is particularly useful for tasks like compression, encryption, or transforming the format of data.

a. Example: Creating a Transform Stream

```
const { Transform } = require('stream');

class ToUpperCaseTransform extends Transform {
  // Transform method will be used to modify the data
  _transform(chunk, encoding, callback) {
    this.push(chunk.toString().toUpperCase());
    callback();
  }
}

const toUpperCaseStream = new ToUpperCaseTransform();

// Write some data to the transform stream
toUpperCaseStream.write('Hello, ');
toUpperCaseStream.write('Transform Stream!\n');

// Read transformed data
toUpperCaseStream.on('data', (chunk) => {
  console.log(chunk.toString());  // Output: HELLO, TRANSFORM STREAM!
});
```

- **_transform()** method is where the actual transformation happens. It modifies the incoming data and pushes the transformed data to the readable side of the stream.

5. Handling Stream Events

Streams emit several events that you can listen to for managing and monitoring the stream lifecycle.

Common stream events:

- **'data'**: Fired when a chunk of data is available in the stream.
- **'end'**: Fired when there’s no more data to read in a readable stream.
- **'finish'**: Fired when all data has been written in a writable stream.
- **'error'**: Fired when an error occurs in the stream.

Example: Handling Stream Events

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

const readableStream = fs.createReadStream('input.txt');

// Listening to the 'data' event
readableStream.on('data', (chunk) => {
  console.log('Received chunk:', chunk.toString());
});

// Listening to the 'end' event
readableStream.on('end', () => {
  console.log('Stream has finished reading all data');
});

// Listening to the 'error' event
readableStream.on('error', (err) => {
  console.error('Error occurred:', err);
});
```

6. Performance Considerations with Streams

- **Memory Efficiency**: Streams work well with large files because they don't load the entire file into memory. They process the file chunk by chunk, making them ideal for handling data like video, images, or logs.
- **Backpressure Handling**: Streams automatically handle backpressure (when the writable stream can't keep up with the readable stream) by pausing and resuming the flow of data.

Accessing and working with the **file system** is a common task in Node.js, and it is done using the built-in **fs module** (File System module). This module allows you to interact with the file system in a synchronous or asynchronous way, providing methods for reading, writing, updating, and deleting files and directories.

Let's break down how to use the **fs module** to perform common file system tasks like opening, closing, reading, writing files, and performing other file-related tasks.

1. Importing the fs Module

Before accessing the file system, we need to require the fs module:

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

2. Reading Files

There are multiple ways to read files in Node.js using the fs module.

a. Asynchronous Read

Asynchronous file reading is non-blocking, meaning Node.js continues execution while reading the file in the background.

```
// Asynchronous read
```

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

- **readFile()** reads the content of a file asynchronously.
- The second argument specifies the encoding (e.g., 'utf8').
- The callback function receives two arguments: an error (if any) and the file content.

b. Synchronous Read

Synchronous reading blocks the program execution until the file is completely read.

```
// Synchronous read
try {
  const data = fs.readFileSync('example.txt', 'utf8');
  console.log('File content:', data);
} catch (err) {
  console.error('Error reading file:', err);
}
```

- **readFileSync()** is the synchronous version. It returns the content of the file directly or throws an error if something goes wrong.

3. Writing Files

a. Asynchronous Write

To write data asynchronously, you can use **fs.writeFile()**. This method will overwrite the content of the file if it already exists, or create a new file if it doesn't exist.

```
// Asynchronous write
fs.writeFile('output.txt', 'Hello, Node.js!', 'utf8', (err) => {
  if (err) {
    console.error('Error writing file:', err);
    return;
  }
  console.log('File has been written');
});
```

- The first argument is the file name.
- The second argument is the content to be written.
- The third argument is the encoding ('utf8' is common).
- The callback function is invoked once the write operation is complete.

b. Synchronous Write

```
// Synchronous write
try {
  fs.writeFileSync('output.txt', 'Hello, Node.js!', 'utf8');
  console.log('File has been written');
} catch (err) {
  console.error('Error writing file:', err);
}
```

- **writeFileSync()** is the synchronous version that blocks the program until the file is written.

4. Appending to Files

If you want to **append** content to an existing file (instead of overwriting it), you can use **fs.appendFile()**.

a. Asynchronous Append

```
// Asynchronous append
fs.appendFile('output.txt', '\nAppended text.', 'utf8', (err) => {
  if (err) {
    console.error('Error appending file:', err);
    return;
  }
  console.log('Data has been appended to the file');
});
```

b. Synchronous Append

```
// Synchronous append
try {
  fs.appendFileSync('output.txt', '\nAppended text.', 'utf8');
  console.log('Data has been appended to the file');
} catch (err) {
  console.error('Error appending file:', err);
}
```

5. Opening and Closing Files

You can open files for reading or writing using **fs.open()**, and close them using **fs.close()**.

a. Opening a File

```
// Opening a file (asynchronous)
fs.open('example.txt', 'r', (err, fd) => {  // 'r' means open for reading
  if (err) {
    console.error('Error opening file:', err);
    return;
  }
  console.log('File opened successfully with descriptor:', fd);

  // Perform reading or writing operations using the file descriptor
});
```

b. Closing a File

Once you’re done with the file, you should **close it**:

```
fs.close(fd, (err) => {
  if (err) {
    console.error('Error closing file:', err);
    return;
  }
  console.log('File closed successfully');
});
```

6. File Stats (Getting Information about a File)

To get details like size, modification time, permissions, etc., you can use **fs.stat()** or **fs.statSync()**.

a. Asynchronous File Stats

```
fs.stat('example.txt', (err, stats) => {
  if (err) {
    console.error('Error getting file stats:', err);
    return;
  }
  console.log('File stats:', stats);
  console.log('Is file?', stats.isFile());
  console.log('Size:', stats.size);
});
```

b. Synchronous File Stats

```
try {
  const stats = fs.statSync('example.txt');
  console.log('File stats:', stats);
  console.log('Is file?', stats.isFile());
  console.log('Size:', stats.size);
} catch (err) {
  console.error('Error getting file stats:', err);
}
```

7. Deleting Files

To delete a file, use **fs.unlink()** or **fs.unlinkSync()**.

a. Asynchronous File Deletion

```
fs.unlink('output.txt', (err) => {
  if (err) {
    console.error('Error deleting file:', err);
    return;
  }
  console.log('File deleted successfully');
});
```

b. Synchronous File Deletion

```
try {
  fs.unlinkSync('output.txt');
  console.log('File deleted successfully');
} catch (err) {
  console.error('Error deleting file:', err);
}
```

8. Creating Directories

You can create a new directory using **fs.mkdir()** or **fs.mkdirSync()**.

a. Asynchronous Directory Creation

```
fs.mkdir('new_directory', { recursive: true }, (err) => {
  if (err) {
    console.error('Error creating directory:', err);
    return;
  }
  console.log('Directory created successfully');
});
```

- The { recursive: true } option ensures that nested directories are created if necessary.

b. Synchronous Directory Creation

```
try {
  fs.mkdirSync('new_directory', { recursive: true });
  console.log('Directory created successfully');
} catch (err) {
  console.error('Error creating directory:', err);
}
```

9. Reading Directory Contents

To list the files and directories within a directory, you can use **fs.readdir()** or **fs.readdirSync()**.

a. Asynchronous Directory Read

```
fs.readdir('my_directory', (err, files) => {
  if (err) {
    console.error('Error reading directory:', err);
    return;
  }
  console.log('Files in directory:', files);
});
```

b. Synchronous Directory Read

```
try {
  const files = fs.readdirSync('my_directory');
  console.log('Files in directory:', files);
} catch (err) {
  console.error('Error reading directory:', err);
}
```

10. Renaming Files

To rename or move a file, use **fs.rename()** or **fs.renameSync()**.

a. Asynchronous File Rename

```
fs.rename('old_name.txt', 'new_name.txt', (err) => {
  if (err) {
    console.error('Error renaming file:', err);
    return;
  }
  console.log('File renamed successfully');
});
```

b. Synchronous File Rename

```
try {
  fs.renameSync('old_name.txt', 'new_name.txt');
  console.log('File renamed successfully');
} catch (err) {
  console.error('Error renaming file:', err);
}
```

Node.js is commonly used to implement **HTTP services** and create web servers due to its non-blocking, event-driven architecture. In this context, we typically use the **http** module to set up an HTTP server, handle incoming requests, process **URLs**, **query strings**, and **form parameters**, and send appropriate **responses**.

Let's go through the different aspects of implementing HTTP services in Node.js, including processing URLs, query strings, and form parameters.

1. Setting Up a Basic HTTP Server

To create an HTTP server in Node.js, you use the http module. Here's a simple example of setting up a basic HTTP server:

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

// Create an HTTP server
const server = http.createServer((req, res) => {
  // Set the HTTP response header
  res.writeHead(200, { 'Content-Type': 'text/plain' });

  // Send a response body
  res.end('Hello, World!');
});

// The server listens on port 3000
server.listen(3000, () => {
  console.log('Server running at http://localhost:3000/');
});
```

- **http.createServer()**: This method creates an HTTP server that listens for requests and sends responses.
- **req (Request)**: The request object represents the incoming request (URL, headers, body, etc.).
- **res (Response)**: The response object is used to send back data to the client.

The server is listening on port 3000. When you visit http://localhost:3000/, the server responds with **"Hello, World!"**.

2. Processing URLs in Node.js

When an HTTP request is made, it contains a URL. You can extract the URL from the request object to handle routing and responses based on different paths.

Example: Handling Different URLs

```
const http = require('http');
const url = require('url'); // Required to parse URLs

const server = http.createServer((req, res) => {
  // Parse the request URL
  const parsedUrl = url.parse(req.url, true); // `true` to parse query strings

  // Handle different paths
  if (parsedUrl.pathname === '/') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('Home Page');
  } else if (parsedUrl.pathname === '/about') {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end('About Page');
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Page Not Found');
  }
});

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

- **url.parse()**: The url module's parse() method is used to extract parts of the URL, such as the pathname and query string.
- In this example, depending on the path (/ or /about), different responses are sent.

3. Processing Query Strings

When making a request to a URL with query parameters, the query string appears after the ? symbol in the URL (e.g., http://localhost:3000/search?term=nodejs).

You can process query strings by parsing the URL, which will give you an object representing the query parameters.

Example: Extracting Query Strings

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

const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url, true); // `true` to parse query string into an object
  const queryParams = parsedUrl.query;

  // If there's a `term` query parameter, return its value
  if (queryParams.term) {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end(`Search term: ${queryParams.term}`);
  } else {
    res.writeHead(400, { 'Content-Type': 'text/plain' });
    res.end('Missing search term');
  }
});

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

- **url.parse(req.url, true)**: The second argument true makes sure the query string is parsed into a JavaScript object (e.g., { term: 'nodejs' }).
- The server responds with the value of the term query parameter, if provided (e.g., http://localhost:3000/search?term=nodejs).

4. Handling Form Parameters

When submitting a form via HTTP (POST method), form data is sent in the body of the request. Node.js doesn't automatically parse this data, so you need to manually handle it.

For simplicity, let's use the **querystring module** to parse URL-encoded form data.

Example: Handling a POST Form Submission

Let's create an HTML form that submits data to a Node.js server:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Form Example</title>
</head>
<body>
  <form action="http://localhost:3000/submit" method="POST">
    <label for="name">Name:</label>
    <input type="text" id="name" name="name" required><br><br>
    <input type="submit" value="Submit">
  </form>
</body>
</html>
```

Now, create a server to handle the form submission:

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

const server = http.createServer((req, res) => {
  // Handle POST request at /submit
  if (req.method === 'POST' && req.url === '/submit') {
    let body = '';

    // Collect data as it comes in chunks
    req.on('data', (chunk) => {
      body += chunk;
    });

    // Once all data is received, parse the form data
    req.on('end', () => {
      const formData = querystring.parse(body); // Parse the form data
      console.log(formData); // { name: 'John' }

      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end(`Form submitted with name: ${formData.name}`);
    });
  } else {
    // For other routes, send 404
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Page not found');
  }
});

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

Explanation:

- When the form is submitted, it sends a **POST** request to http://localhost:3000/submit.
- The server listens for data chunks using the **req.on('data')** event and assembles the complete body in the body variable.
- Once the data is fully received (**req.on('end')**), we parse it using **querystring.parse()** to extract form parameters (like name).
- Finally, we send the form data back in the response.

5. Understanding Request, Response, and Server Objects

- **req (Request)**: Represents the HTTP request from the client. It contains properties like:
  - req.url: The full URL of the request (including query strings).
  - req.method: The HTTP method (GET, POST, etc.).
  - req.headers: An object containing request headers.
  - req.body: Contains data sent in the request body (available in POST requests).
- **res (Response)**: Represents the HTTP response sent to the client. It contains methods like:
  - res.writeHead(statusCode, headers): Sends the response status code and headers.
  - res.end(): Ends the response and sends data to the client.
  - res.write(): Writes data to the response body.
- **server**: The server object created using http.createServer(). It listens for incoming requests and handles them based on the URL and HTTP method.

In Node.js, you can easily implement both **HTTP clients** and **servers**, as well as **HTTPS clients** and **servers**, thanks to the built-in http and https modules.

Let’s break down the two main areas you asked about:

1. **Implementing HTTP Servers and Clients in Node.js**
2. **Implementing HTTPS Servers and Clients in Node.js**

1. Implementing HTTP Servers and Clients in Node.js

HTTP Server in Node.js

An HTTP server listens for incoming HTTP requests and sends back HTTP responses. This can be done using the http module.

Example: Basic HTTP Server

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

// Create an HTTP server
const server = http.createServer((req, res) => {
  // Set HTTP headers
  res.writeHead(200, { 'Content-Type': 'text/plain' });

  // Handle different URL paths
  if (req.url === '/') {
    res.end('Welcome to the Home Page!');
  } else if (req.url === '/about') {
    res.end('Welcome to the About Page!');
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Page Not Found');
  }
});

// Server listens on port 3000
server.listen(3000, () => {
  console.log('Server is running at http://localhost:3000/');
});
```

- **http.createServer()**: Creates the HTTP server and accepts a callback function that handles incoming requests (req) and sends back responses (res).
- **req.url**: Used to determine the requested URL path and handle requests differently based on the URL.
- **res.writeHead()**: Sets the response status code (200 for success, 404 for not found) and the content type.
- **res.end()**: Ends the response and sends data back to the client.

When you visit http://localhost:3000/, the server will respond with "Welcome to the Home Page!" and on http://localhost:3000/about, it will return "Welcome to the About Page!".

HTTP Client in Node.js

Node.js also provides the http module to make HTTP requests to other servers. For example, you can create an HTTP client to request data from an external server.

Example: HTTP Client (GET Request)

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

// Send a GET request to an external URL
http.get('http://example.com', (res) => {
  let data = '';

  // Collect data chunks
  res.on('data', (chunk) => {
    data += chunk;
  });

  // When the response is fully received, log the result
  res.on('end', () => {
    console.log('Response from server:');
    console.log(data);
  });
}).on('error', (err) => {
  console.error('Error:', err.message);
});
```

- **http.get()**: This method sends an HTTP GET request to the specified URL. The response is handled in the callback function.
- **res.on('data')**: Handles the data received in chunks from the server.
- **res.on('end')**: Triggered when all the response data has been received.

When you run this client, it will request data from http://example.com and print the response to the console.

2. Implementing HTTPS Servers and Clients in Node.js

HTTPS Server in Node.js

Node.js can also handle **HTTPS** requests, which is a secure version of HTTP. To implement an HTTPS server, you need an SSL/TLS certificate. You can either use a self-signed certificate for testing or a certificate from a trusted certificate authority (CA).

Example: Basic HTTPS Server

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

// Load SSL certificates (self-signed for testing)
const options = {
  key: fs.readFileSync('server-key.pem'),
  cert: fs.readFileSync('server-cert.pem'),
};

const server = https.createServer(options, (req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Secure HTTPS Server!');
});

// Server listens on port 3000
server.listen(3000, () => {
  console.log('HTTPS Server running at https://localhost:3000/');
});
```

- **https.createServer()**: The method for creating an HTTPS server, similar to http.createServer(), but requires additional options for SSL/TLS encryption.
- **options.key**: The private key for your SSL certificate (usually .pem or .key file).
- **options.cert**: The public certificate for your SSL certificate (usually .pem or .crt file).

Generating Self-Signed SSL Certificates for Testing

If you're just testing locally, you can generate a self-signed SSL certificate using OpenSSL:

```
openssl genpkey -algorithm RSA -out server-key.pem
openssl req -new -key server-key.pem -out server.csr
openssl x509 -req -in server.csr -signkey server-key.pem -out server-cert.pem
```

This will generate server-key.pem and server-cert.pem, which you can use in your HTTPS server.

HTTPS Client in Node.js

To make secure HTTPS requests (similar to HTTP client), you can use the https module in Node.js.

Example: HTTPS Client (GET Request)

javascript
Copy
```javascript
const https = require('https');

// Send a GET request to an external HTTPS URL
https.get('https://jsonplaceholder.typicode.com/posts/1', (res) => {
  let data = '';

  // Collect data chunks
  res.on('data', (chunk) => {
    data += chunk;
  });

  // When the response is fully received, log the result
  res.on('end', () => {
    console.log('Response from secure server:');
    console.log(data);
  });
}).on('error', (err) => {
  console.error('Error:', err.message);
});
```

- **https.get()**: Sends a GET request to a secure (HTTPS) server, like the previous HTTP client example.
- The server will respond with JSON data from jsonplaceholder.typicode.com, and the client will log it to the console.

Summary of Key Differences

- **HTTP vs HTTPS**:
  - **HTTP**: Standard protocol for unencrypted data transmission.
  - **HTTPS**: Secure version of HTTP that encrypts data using SSL/TLS certificates.
  - For an **HTTPS server**, you need SSL certificates to establish a secure connection.
- **Server Setup**:
  - **HTTP Server**: Use the http.createServer() method.
  - **HTTPS Server**: Use the https.createServer() method, which requires an SSL certificate and private key.
- **Client Setup**:
  - **HTTP Client**: Use the http.get() or http.request() methods to make HTTP requests.
  - **HTTPS Client**: Use the https.get() or https.request() methods for secure HTTPS requests.

Important Notes:

- You can use the **https module** for making secure requests to other HTTPS servers, as shown in the client examples.
- For **development** and **testing**, you can generate self-signed SSL certificates, but for production environments, it’s important to use a trusted certificate authority (CA).

Node.js provides a rich set of built-in modules that you can use to handle various system-level operations, such as interacting with the operating system, working with utilities, resolving domain names, and performing cryptographic operations. Below, we'll look at the **os**, **util**, **dns**, and **crypto** modules in detail.

1. Using the os Module

The os (Operating System) module provides methods for interacting with the system's operating system-related functionalities, such as fetching information about the CPU, memory, and network interfaces.

Common os Module Methods:

- **os.arch()**: Returns the architecture of the CPU (e.g., 'x64' or 'arm').
- **os.cpus()**: Returns an array of objects containing information about the system's CPUs (core count, speed, etc.).
- **os.freemem()**: Returns the amount of free system memory in bytes.
- **os.totalmem()**: Returns the total system memory in bytes.
- **os.homedir()**: Returns the home directory path of the current user.
- **os.networkInterfaces()**: Returns network interfaces and their details.
- **os.platform()**: Returns the platform of the operating system (e.g., 'linux', 'win32', 'darwin' for macOS).

Example: Using the os Module

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

// Get the architecture of the operating system
console.log('Architecture:', os.arch());

// Get CPU information
```

```
console.log('CPU Info:', os.cpus());

// Get free memory
console.log('Free Memory:', os.freemem());

// Get total memory
console.log('Total Memory:', os.totalmem());

// Get the home directory of the current user
console.log('Home Directory:', os.homedir());

// Get network interfaces
console.log('Network Interfaces:', os.networkInterfaces());
```

2. Using the util Module

The util module provides utility functions that help with common programming tasks such as inspecting objects, formatting strings, and debugging.

Common util Module Methods:

- **util.format()**: Works like printf in C; formats a string using placeholders.
- **util.inspect()**: Returns a string representation of an object for debugging purposes.
- **util.promisify()**: Converts callback-based functions into promise-based functions.
- **util.deprecate()**: Marks a function as deprecated and issues a warning when it’s used.

Example: Using the util Module

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

// Format a string with values
const formattedString = util.format('Hello, %s!', 'World');
console.log(formattedString);  // Output: Hello, World!

// Inspect an object
const obj = { name: 'Alice', age: 25 };
console.log(util.inspect(obj));  // Output: { name: 'Alice', age: 25 }

// Promisify a callback-based function
const fs = require('fs');
const readFile = util.promisify(fs.readFile);

// Use the promisified function with async/await
async function readFileAsync() {
  try {
    const data = await readFile('example.txt', 'utf8');
    console.log(data);
  } catch (err) {
    console.error('Error reading file:', err);
  }
}

readFileAsync();
```

3. Using the dns Module

The dns (Domain Name System) module provides functions to interact with the DNS system. You can resolve domain names into IP addresses or get the reverse DNS lookup.

Common dns Module Methods:

- **dns.lookup()**: Resolves a domain name into an IP address.
- **dns.resolve()**: Resolves a domain name to an array of records (e.g., A, AAAA, MX, etc.).
- **dns.reverse()**: Performs a reverse DNS lookup, returning the domain name associated with an IP address.
- **dns.promises**: The dns.promises API provides promise-based versions of DNS methods.

Example: Using the dns Module

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

// Resolve a domain name to an IP address
dns.lookup('example.com', (err, address, family) => {
  if (err) {
    console.error('DNS lookup error:', err);
  } else {
    console.log('IP Address:', address);  // Example: 93.184.216.34
    console.log('IP Family:', family);    // Example: 4 (IPv4)
  }
});

// Resolve a domain to its MX (Mail Exchange) records
dns.resolve('example.com', 'MX', (err, records) => {
  if (err) {
    console.error('DNS resolve error:', err);
  } else {
    console.log('MX Records:', records);
  }
});

// Reverse DNS lookup for an IP address
dns.reverse('8.8.8.8', (err, hostnames) => {
  if (err) {
    console.error('Reverse DNS lookup error:', err);
  } else {
    console.log('Hostnames:', hostnames);
  }
});
```

4. Using the crypto Module

The crypto module provides cryptographic functionality, such as creating hashes, encryption/decryption, and secure random number generation.

Common crypto Module Methods:

- **crypto.createHash()**: Creates a hash object that can be used to hash data (e.g., sha256 or md5).
- **crypto.createHmac()**: Creates an HMAC (Hash-based Message Authentication Code) object.
- **crypto.randomBytes()**: Generates cryptographically strong pseudo-random data.
- **crypto.pbkdf2()**: Implements the PBKDF2 (Password-Based Key Derivation Function 2) algorithm.
- **crypto.createCipheriv()**: Encrypts data using a cipher algorithm.
- **crypto.createDecipheriv()**: Decrypts data using a cipher algorithm.

Example: Using the crypto Module

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

// Create a hash using SHA-256
const hash = crypto.createHash('sha256');
hash.update('Hello, World!');
const result = hash.digest('hex');
console.log('SHA-256 Hash:', result);

// Generate random bytes (e.g., 16 bytes)
crypto.randomBytes(16, (err, buffer) => {
  if (err) throw err;
  console.log('Random Bytes:', buffer.toString('hex'));
});

// Create an HMAC using a secret key and SHA-256
const hmac = crypto.createHmac('sha256', 'secret-key');
hmac.update('Some message');
console.log('HMAC:', hmac.digest('hex'));

// PBKDF2 key derivation (for password hashing)
crypto.pbkdf2('password', 'salt', 100000, 64, 'sha512', (err, derivedKey) => {
  if (err) throw err;
  console.log('PBKDF2 Derived Key:', derivedKey.toString('hex'));
});
```