Day 2: Node.js Core Modules

- Introduction to modules in Node.js
- Working with the File System (fs) module
- Understanding and using the Path module
- Creating an HTTP server

1. Introduction to Modules in Node.js

What are Modules?

- **Modules** in Node.js are reusable blocks of code that encapsulate related functions, making it easier to organize and manage code.
- Node.js uses a **module system** where each file is considered a module.
- There are two types of modules:
  - **Core Modules**: These are built into Node.js, like fs, path, http.
  - **User-Defined Modules**: Custom modules created by developers.

How to Use a Module?

- Modules are imported using the require() function.

moduleExport

```
function express() {
  return "Hema coding school";
}

hema = {
  name: "Mahesh",
  age: 23,
  dev: true,
};

// module.exports = hema;
// module.exports = express;

module.exports = {
  hema,
  express,
};

console.log(exports, require, module, __filename, __dirname)
```

moduleImport

```
const call = require('./moduleExport.js')
```

```
console.log(call.express())
console.log(call.hema)
```

2. Working with the File System (fs) Module

Overview of fs Module:

- The fs (File System) module provides functions to interact with the file system, like reading and writing files, creating directories, and more.

```
// ------------------------------------------------fs

// const fs = require('fs')
// console.log(fs)

// fs.writeFile('newFile.txt','This is a new document!',()=>{
//     console.log("File has created succesfully")
// })
// console.log(fs)

// fs.readFile('newFile.txt',(err,data)=>{
//     console.log("File content:", data)
// })
// output: File content: <Buffer 54 68 69 73 20 69 73 20 61 20 6e 65 77 20 64 6f 63 75 6d 65
6e 74 21>

// fs.readFile('newFile.txt','utf-8',(err,data)=>{
//     console.log("File content:", data)
// })
// output: File content: This is a new document!
// fs.readFile('newFile.txt','utf-8',(err,data)=>{
//    if(err){
//     console.log("Please check ")

//    }else{

//        console.log("File content:", data)
//    }
// })

// fs.unlink('newFile.txt',()=>{
//     console.log("File has deleted")
// })
```

```

```

3. Understanding and Using the path Module

Overview of path Module:

- The **path** module helps with handling and transforming file paths. It provides utilities for working with file and directory paths in a cross-platform manner.

```
// const path = require('path');
// console.log(path)
// const filePath = path.join('folder', 'subfolder', 'file.txt')

// console.log(filePath)
// console.log(path.extname(filePath))


// const filePath = '/folder/subfolder/file.txt';

// const parsedPath = path.parse(filePath);

// console.log(parsedPath);
```

Cross-Platform Path Handling: If you need to handle file paths on both **Windows** and **Linux**, where slashes are different (\ vs /), the path module abstracts this for you:

```

// const logFilePath = path.join(__dirname,"logs","app.log")
// console.log(logFilePath)
```

Working with File Extensions: If you need to check file types, such as validating that an uploaded file is a .jpg image:

```

// const filePath = 'uploads/photo.jpg';

// if(path.extname(filePath) ===.jpg')
// {
//     console.log("This is JPG file")
```

```
// }
```

Parsing and Rebuilding File Paths: When working with paths in different parts of your application, you can easily split and rebuild paths:

```
// const filePath = '/folder/subfolder/file.txt';
// const parsedPath = path.parse(filePath);

// console.log(parsedPath.base);  // Output: file.txt

// // Modify and rebuild path
// const newFilePath = path.format({
//   dir: parsedPath.dir,
//   name: 'newFile',
//   ext: parsedPath.ext
// });

// console.log(newFilePath);  // Output: /folder/subfolder/newFile.txt
```

Readline :

```
// const readline = require('readline');

// // Create an interface to read data from the command line
// const rl = readline.createInterface({
//   input: process.stdin,
//   output: process.stdout
// });

// // Ask the user for input
// rl.question('What is your name? ', (answer) => {
//   console.log(`Hello, ${answer}!`);

//   // Close the input stream once the input is received
//   rl.close();
// });
// rl.on('line', (input) => {
//     console.log(`Received: ${input}`);
//     rl.close()
//   });
```

4. Creating an HTTP Server

Overview of HTTP Server:

- The **http** module is used to create an HTTP server, which can handle incoming requests and send responses.

```
const http = require('http');
const server = http.createServer((req,res)=>{
   res.writeHead(200,{"Content-Type": "text/plain"})
   res.end("Welocme hema coding school")
})
// PORT, HOSTNAME, BACKLOG, CALLBACK
const PORT = 3000;
const HOSTNAME = "127.0.0.1";
const BACKLOG = 551;
const CALLBACK = ()=>{
   console.log(`Server has started http://${HOSTNAME}:${PORT}`)
}
server.listen(PORT, HOSTNAME, BACKLOG, CALLBACK);
```

Introduction to Modules in Node.js

1. **What are modules in Node.js, and why are they important?**
    - **Answer:** Modules in Node.js are reusable blocks of code that help organize related functionalities into separate files or packages. They are important because they promote code reuse, maintainability, and separation of concerns. Node.js has two main types of modules:
        - **Core modules:** Provided by Node.js (e.g., fs, path, http).
        - **User-defined modules:** Custom modules created by developers.
2. **How do you import and export modules in Node.js?**
    - **Answer:**
        - To import a module, you use the require() functio

```
const fs = require('fs');  // Core module
const myModule = require('./myModule');  // User-defined module
```

To export functionality from a module, you use module.exports:

```
// Inside myModule.js
function greet() {
  console.log('Hello!');
}
module.exports = greet;
```

What is the difference between require() and import in Node.js?

- **Answer:** require() is used to import modules in Node.js (CommonJS module system), whereas import is part of the ES6 module system. Node.js primarily uses require(), but with newer versions, you can enable ES6 modules using "type": "module" in your package.json file.

Working with the File System (fs) Module

4. **What is the fs module in Node.js, and what is it used for?**
    - **Answer:** The fs (File System) module in Node.js provides methods to interact with the file system, allowing you to read, write, update, delete, and manage files and directories. It is a core module, so no external installation is required.
5. **How do you read a file in Node.js using the fs module?**
    - **Answer:** You can use the fs.readFile() method to read a file asynchronously

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

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

What is the difference between fs.readFile() and fs.readFileSync()?

- **Answer:**
  - **fs.readFile()** is asynchronous, meaning it doesn't block the execution of the rest of the code while the file is being read.
  - **fs.readFileSync()** is synchronous and blocks the event loop until the file has been completely read.

```
// Asynchronous
fs.readFile('example.txt', 'utf8', (err, data) => { /* ... */ });

// Synchronous
const data = fs.readFileSync('example.txt', 'utf8');
```

How do you write data to a file using the fs module?

- **Answer:** You can use the fs.writeFile() method to write data to a file. It will create the file if it doesn't exist or overwrite it if it does:

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

fs.writeFile('output.txt', 'Hello, Node.js!', (err) => {
  if (err) {
    console.error('Error writing to file:', err);
  } else {
    console.log('File written successfully!');
  }
});
```

How do you delete a file using the fs module?

- **Answer:** You can use the fs.unlink() method to delete a file

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

fs.unlink('output.txt', (err) => {
  if (err) {
    console.error('Error deleting file:', err);
```

```
  } else {
    console.log('File deleted successfully!');
  }
});
```

Understanding and Using the path Module

9. **What is the path module in Node.js, and why is it used?**
    - **Answer:** The path module in Node.js provides utilities for working with file and directory paths. It helps handle file paths across different operating systems (e.g., / for Unix-like systems, \ for Windows), ensuring cross-platform compatibility.
10. **How do you join multiple path segments using the path module?**
    - **Answer:** You can use the path.join() method to concatenate multiple path segments into a single path:

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

const fullPath = path.join('folder', 'subfolder', 'file.txt');
console.log(fullPath);  // Output: folder/subfolder/file.txt (Unix-like) or
folder\subfolder\file.txt (Windows)
```

How do you get the file extension of a file using the path module?

- **Answer:** The path.extname() method returns the file extension from a given file path

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

const filePath = '/folder/subfolder/file.txt';
console.log(path.extname(filePath));  // Output: .txt
```

How do you get the base file name from a file path using the path module?

- **Answer:** You can use the path.basename() method to get the file name (including extension) from the path:

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

const filePath = '/folder/subfolder/file.txt';
console.log(path.basename(filePath));  // Output: file.txt
```

How do you convert a relative path to an absolute path using the path module?

- **Answer:** The path.resolve() method converts a relative path into an absolute path based on the current working directory:

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

const absolutePath = path.resolve('folder', 'subfolder', 'file.txt');
console.log(absolutePath);  // Output: /absolute/path/to/folder/subfolder/file.txt
```

Creating an HTTP Server

14. **How do you create a simple HTTP server in Node.js?**
    - **Answer:** You can create an HTTP server using the http module and the http.createServer() method. The server listens on a specific port and handles incoming requests.

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

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!');
});

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

How do you handle different routes in an HTTP server?

- **Answer:** You can handle different routes by checking the req.url property in the server callback function.

```
const server = http.createServer((req, res) => {
   if (req.url === '/') {
     res.statusCode = 200;
     res.setHeader('Content-Type', 'text/html');
     res.end('<h1>Home Page</h1>');
   } else if (req.url === '/about') {
     res.statusCode = 200;
     res.setHeader('Content-Type', 'text/html');
     res.end('<h1>About Us</h1>');
   } else {
     res.statusCode = 404;
     res.end('<h1>Page Not Found</h1>');
   }
 });
```

How do you send JSON data in an HTTP response?

- **Answer:** You can set the content type to application/json and use JSON.stringify() to send a JSON object in the response:

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

const server = http.createServer((req, res) => {
  const data = { name: 'John', age: 30 };
  res.statusCode = 200;
  res.setHeader('Content-Type', 'application/json');
  res.end(JSON.stringify(data));
});

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

What is the difference between res.write() and res.end() in an HTTP server?

- **Answer:**
  - **res.write()**: Writes chunks of data to the response body but does not close the connection.
  - **res.end()**: Ends the response and optionally sends the last chunk of data. Once called, the response is considered complete.

```
res.write('First chunk of data');
res.write('Second chunk of data');
res.end('Final chunk of data');  // Ends the response
```