

Node.js Documentation

Module 1: Intro to Node.js

- **What is Node.js?**

Node.js is a runtime environment that allows you to run JavaScript on the server side. It uses the V8 JavaScript engine, the same engine that powers Google Chrome, to execute code.

- **PHP vs. Node.js**

PHP and Node.js are both popular choices for server-side development but have different strengths and use cases.

PHP

1. **Blocking I/O:** PHP uses a blocking I/O model where each I/O operation (like reading a file, or querying a database) blocks the execution of the script until the operation is completed.

2. **Synchronous:** PHP scripts are executed in a synchronous manner.

3. **Common Use Case:** Traditional web development with tools like WordPress, Laravel, etc.

Node.js

1. **Non-blocking I/O:** Node.js uses a non-blocking I/O model where operations can be executed without waiting for previous ones to complete.

2. **Asynchronous:** Node.js is designed for asynchronous execution.
3. **Common Use Case:** Real-time applications, API development, and applications requiring high concurrency.

- **RAM vs. I/O Latency**

RAM (Random Access Memory)

RAM is a type of computer memory that can be accessed randomly. It is used to store data that is being actively used or processed by the CPU. Accessing data from RAM is much faster compared to accessing data from a hard drive or SSD.

Key Points:

1. **Fast Access:** Data in RAM can be accessed in nanoseconds, making it extremely fast.
2. **Volatile:** RAM is volatile memory, meaning it loses all stored information when the power is turned off.
3. **Temporary Storage:** RAM is used for temporary storage of data that the CPU needs quick access to.
4. **Limited Size:** RAM is typically smaller in size compared to disk storage but is much faster.
5. **Used for Active Data:** RAM stores data that is actively being used or processed by applications.

6. **Expensive:** RAM is more expensive per byte compared to disk storage.
7. **Performance Impact:** More RAM can significantly improve the performance of applications by reducing the need to access slower storage.
8. **Random Access:** RAM allows for data to be read and written in any order, providing flexibility and speed.

I/O Latency (Input/Output Latency)

I/O latency refers to the time it takes to complete an I/O operation, such as reading from or writing to a disk or network. It is typically measured in milliseconds or microseconds.

Key Points:

- **Slower Access:** I/O operations, such as reading from or writing to a disk or network, have higher latency compared to RAM access.
- **Persistent Storage:** Unlike RAM, data stored in I/O devices (like hard drives and SSDs) is persistent and remains even when the power is off.
- **Physical Limitations:** The speed of I/O operations is limited by the physical properties of the storage medium or network.
- **Measured in Milliseconds:** I/O latency is typically measured in milliseconds or microseconds, which is much slower than nanoseconds for RAM.

- **Blocking Operations:** Synchronous I/O operations can block the execution of a program until the operation is completed.
 - **Asynchronous Operations:** Asynchronous I/O operations allow other tasks to continue executing while waiting for the I/O operation to complete, improving efficiency.
 - **Used for Long-term Storage:** I/O devices are used for long-term storage of data, including files, databases, and backups.
 - **Cost-effective:** I/O devices, especially traditional hard drives, are more cost-effective per byte compared to RAM.
-
- **Blocking vs Non-blocking**

Blocking Operations

Blocking operations cause the execution of subsequent code to wait until the current operation completes. This means that no other operations can be processed until the blocking operation finishes.

Characteristics of Blocking Operations:

1. **Synchronous Execution:** The code is executed line by line, and each operation must complete before the next one starts.
2. **Potential for Delays:** Long-running operations can cause significant delays, making the application unresponsive.
3. **Simple but Inefficient:** Easy to write and understand but can lead to performance bottlenecks.

Example of Blocking Code:

The screenshot shows a Visual Studio Code interface. The title bar says "Index3.js - Mobzway - Visual Studio Code". The status bar at the top right shows "Tue Jul 9 16:28" and "100%". The Explorer sidebar on the left shows files: index.js, index1.js, index2.js, index3.js (which is selected), and index4.js. The main editor area contains the following code:

```
1 const fs = require('fs');
2
3 // Synchronous file read (blocking)
4 const data = fs.readFileSync('/path/to/file', 'utf8');
5 console.log(data);
6 console.log('This message is logged after the file is read');
```

The terminal below shows the execution of the script:

```
node "/home/anuj/Desktop/Mobzway/index3.js"
anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/index3.js"
node:internal/fs/utils:356
    throw err;
^
```

The status bar at the bottom indicates "Ln 7, Col 1" and other settings like "Spaces: 4", "UTF-8", and "LF".

In this example:

- `fs.readFileSync` is a blocking operation that reads the file synchronously.
- The execution of `console.log (This message is logged after the file is read')` is blocked until the file read operation completes.

Non-Blocking Operations

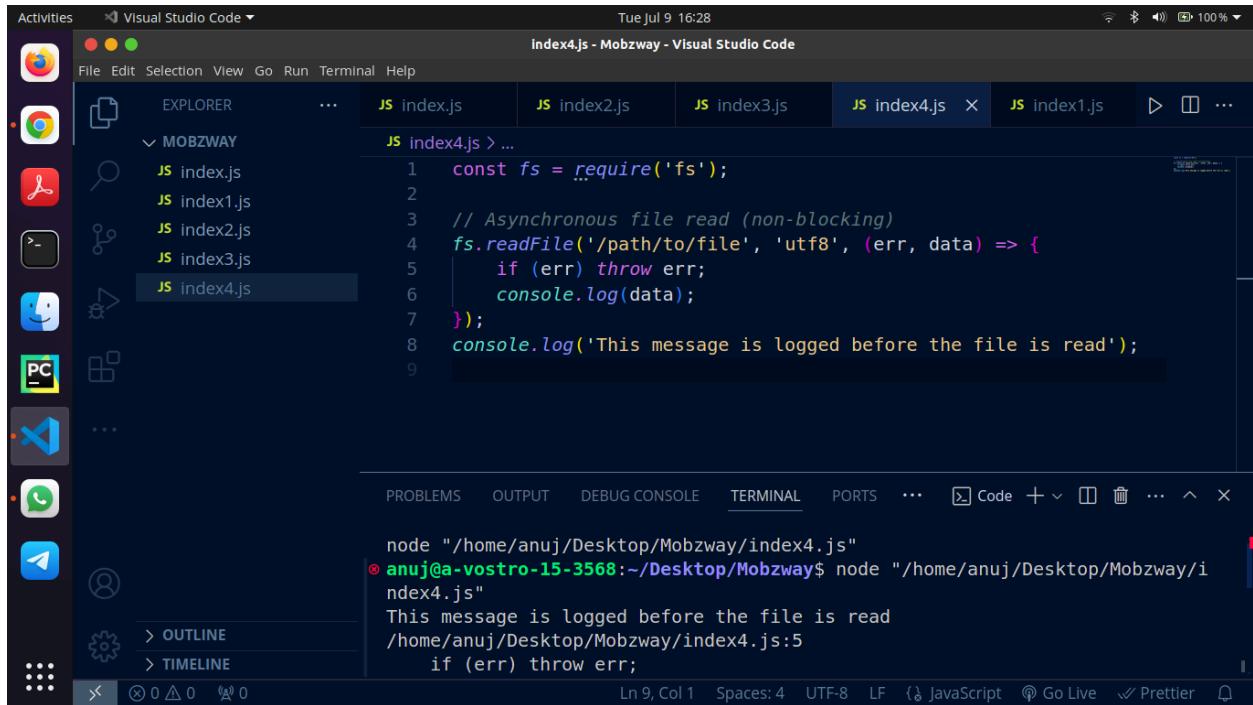
Non-blocking operations allow other tasks to be processed while the current operation is being executed. This approach enables higher concurrency and better performance, especially for I/O-bound tasks.

Characteristics of Non-Blocking Operations:

1. **Asynchronous Execution:** The code continues to execute without waiting for the current operation to complete.

2. **Concurrency:** Multiple operations can be handled concurrently, improving efficiency.
3. **More Complex:** Requires handling callbacks, promises, or async/await, making the code more complex but highly performant.

Example of Non-Blocking Code:



The screenshot shows a Visual Studio Code interface with the following details:

- Title Bar:** Activities, Visual Studio Code, Tue Jul 9 16:28, 100%.
- File Explorer (Left):** Shows files: index.js, index1.js, index2.js, index3.js, index4.js (selected), and index.js > ...
- Code Editor (Center):** Displays the contents of index4.js:

```

1  const fs = require('fs');
2
3  // Asynchronous file read (non-blocking)
4  fs.readFile('/path/to/file', 'utf8', (err, data) => {
5      if (err) throw err;
6      console.log(data);
7  });
8  console.log('This message is logged before the file is read');

```
- Terminal (Bottom):** Shows the command being run: node "/home/anuj/Desktop/Mobzway/index4.js". The output shows the message "This message is logged before the file is read" followed by the error message from line 5 of index4.js.
- Status Bar (Bottom):** Ln 9, Col 1, Spaces: 4, UTF-8, LF, JavaScript, Go Live, Prettier.

In this example:

- `fs.readFile` is a non-blocking operation that reads the file asynchronously.
- `console.log('This message is logged before the file is read')` is executed immediately without waiting for the file read operation to complete.

- **Event-Driven Programming**

Event-driven programming means a callback function will be executed when the corresponding event occurs.

For this, we make an object of the EventEmitter class and through this object, we can access any method in this class.

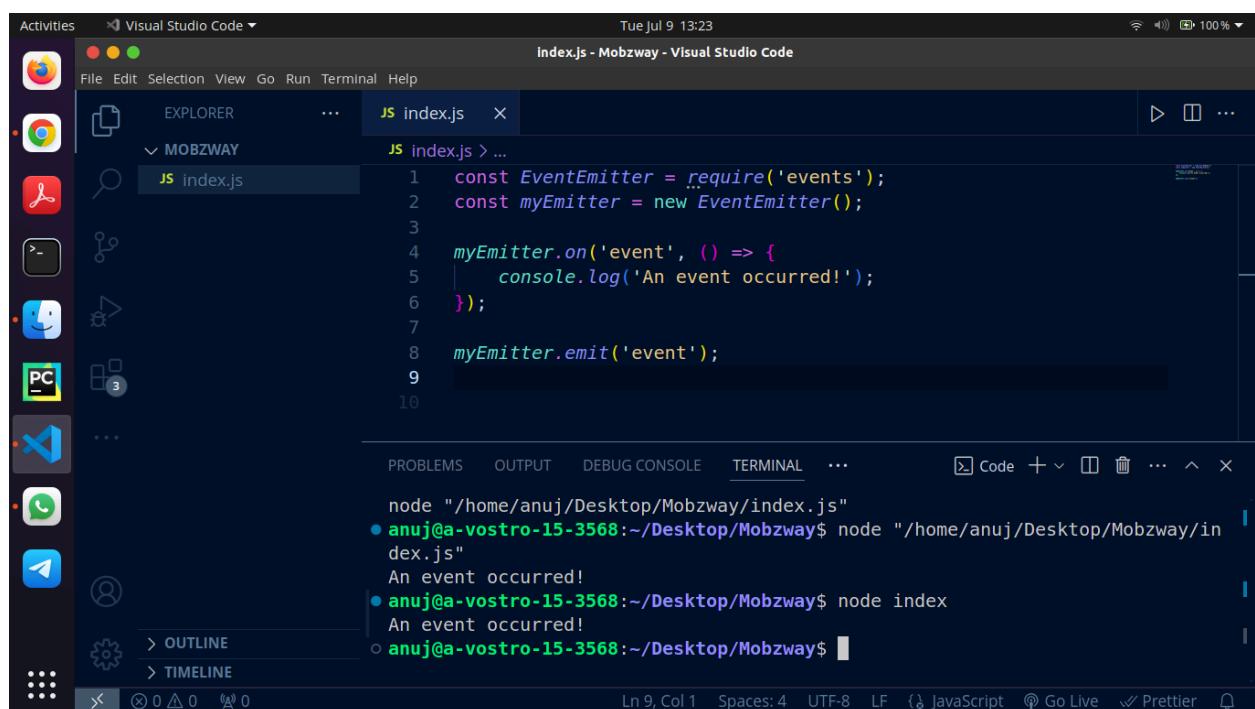
We use .emit method for the occurrence of the event.

.emit method takes the event name as an argument.

.on method has two parameters:

1. Event name
2. Listener: It is a callback function that will be executed when the corresponding event occurs.

Below code is an example of event-driven programming.



The screenshot shows a Visual Studio Code interface. The left sidebar displays various icons for different applications like a browser, file manager, and terminal. The main area shows an 'EXPLORER' view with a file tree under 'MOBZWAY' containing 'index.js'. The code editor window contains the following JavaScript code:

```
1 const EventEmitter = require('events');
2 const myEmitter = new EventEmitter();
3
4 myEmitter.on('event', () => {
5   console.log('An event occurred!');
6 });
7
8 myEmitter.emit('event');
9
```

Below the code editor, the terminal window shows the output of running the script:

```
node "/home/anuj/Desktop/Mobzway/index.js"
● anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/index.js"
An event occurred!
● anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node index
An event occurred!
○ anuj@a-vostro-15-3568:~/Desktop/Mobzway$
```

The status bar at the bottom indicates the code is in JavaScript mode, with 9 lines, 1 column, 4 spaces, and LF line endings.

In the above code, when an event occurs in line 8, the corresponding callback function, which is in line 4, will be executed.

● Event Loop

The event loop is a fundamental concept in Node.js that allows it to handle asynchronous operations and non-blocking I/O. It is the mechanism that Node.js

uses to manage the execution of multiple pieces of code without the need for multiple threads.

Event Loop Explained

1. **Single-Threaded:** Node.js operates on a single-threaded event loop, which is responsible for executing JavaScript code, handling events, and performing I/O operations asynchronously.
2. **Non-blocking I/O:** The event loop enables Node.js to perform non-blocking I/O operations despite being single-threaded. This means it can handle other tasks while waiting for I/O operations to complete.

3. Phases of the Event Loop:

- **Timers:** Executes callbacks scheduled by `setTimeout` and `setInterval`.
 - **Pending Callbacks:** Executes I/O callbacks deferred to the next iteration of the loop.
 - **Idle, Prepare:** Internal use only.
 - **Poll:** Retrieves new I/O events; executes I/O callbacks (almost all events except close, timers, and `setImmediate`).
 - **Check:** Executes callbacks scheduled by `setImmediate`.
 - **Close Callbacks:** Executes close event callbacks like `socket.on('close' ...)`.
-
4. **Microtasks:** Tasks such as `process.nextTick` and resolved Promises have a higher priority than other tasks and are executed immediately after the currently executing script before the event loop continues.

Example Explanation:

The screenshot shows a Visual Studio Code interface with the following details:

- Title Bar:** Visual Studio Code - index1.js - Mobzway - Visual Studio Code
- File Explorer:** Shows a folder named MOBZWAY containing two files: index.js and index1.js. index1.js is currently selected.
- Code Editor:** Displays the following JavaScript code:

```
1 console.log('Start');
2
3 setTimeout(() => {
4   console.log('Timeout 1');
5 }, 0);
6
7 Promise.resolve().then(() => {
8   console.log('Promise 1');
9 });
10
11 console.log('End');
```
- Terminal:** Shows the command being run and its output:

```
anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/index1.js"
Start
End
Promise 1
Timeout 1
anuj@a-vostro-15-3568:~/Desktop/Mobzway$
```
- Status Bar:** Ln 9, Col 4, Spaces: 4, UTF-8, LF, JavaScript, Go Live, Prettier

Let's break down the execution of this example step by step.

1. **Synchronous Code Execution:** The code starts executing from the top.
 - `console.log('Start');` is executed, and "Start" is printed to the console.
 2. **setTimeout:**
 - `setTimeout` is called with a delay of 0 milliseconds. This means the callback function is added to the Timer phase of the event loop but will only be executed after the current stack and microtasks are completed.
 3. **Promise:**
 - `Promise.resolve().then(...)` schedules the callback function to be executed as a microtask (part of the microtask queue).

4. Synchronous Code Execution Continues:

- `console.log('End');` is executed, and "End" is printed to the console.

5. Microtasks Execution:

- After the synchronous code is executed, the event loop checks the microtask queue. The callback from `Promise.resolve().then(...)` is executed, printing "Promise 1" to the console.

6. Timers Phase:

- Once all microtasks are completed, the event loop moves to the Timers phase and executes the `setTimeout` callback, printing "Timeout 1" to the console.

Summary of the Event Loop Execution

1. **Initial Execution:** `console.log('Start');` and `console.log('End');` are executed sequentially.
2. **Scheduling Tasks:** `setTimeout` schedules a callback in the Timers phase, and `Promise.resolve().then(...)` schedules a microtask.
3. **Microtasks:** Microtasks from the microtask queue (like resolved Promises) are executed after the current stack is completed but before moving to the next phase of the event loop.
4. **Timers Phase:** Once all synchronous code and microtasks are executed, the event loop proceeds to execute the `setTimeout` callback in the Timers phase.

- **Blocking the Event Loop**

Blocking the event loop in Node.js is a critical concept to understand because it directly affects the performance and responsiveness of your application. Node.js is designed to handle many operations concurrently, but blocking the event loop can prevent it from processing other tasks, leading to delays and decreased performance.

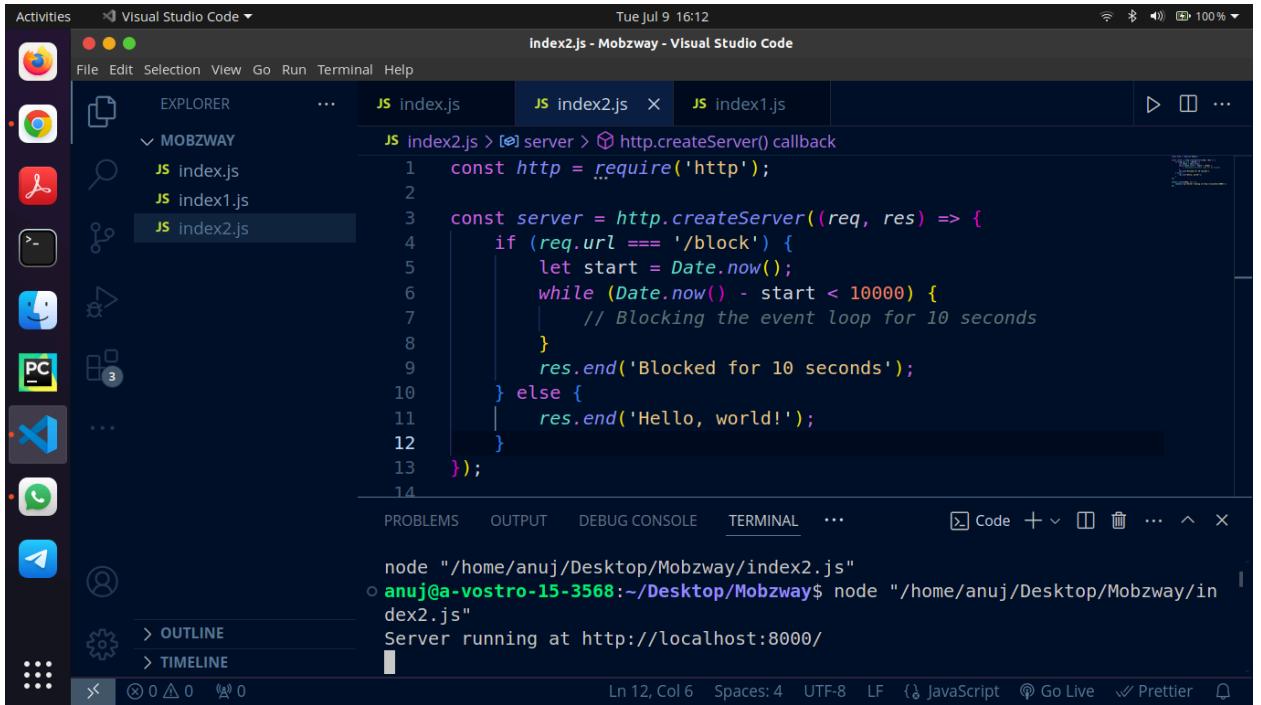
What is Blocking the Event Loop?

Blocking the event loop occurs when a task takes a long time to complete and prevents other tasks from executing. Since Node.js uses a single-threaded event loop to handle all operations, any function that takes a long time to execute will block the event loop and delay the execution of other pending tasks.

How Does Blocking Happen?

1. **Synchronous Code:** Any synchronous code that takes a long time to execute will block the event loop. Examples include complex calculations, large loops, or heavy file operations.
2. **I/O Operations:** While Node.js is designed to handle I/O operations asynchronously, if you use synchronous I/O methods, they will block the event loop.
3. **Long-Running Operations:** Functions that perform long-running operations without yielding control back to the event loop will cause blocking.

Example of Blocking Code



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

const server = http.createServer((req, res) => {
  if (req.url === '/block') {
    let start = Date.now();
    while (Date.now() - start < 10000) {
      // Blocking the event loop for 10 seconds
    }
    res.end('Blocked for 10 seconds');
  } else {
    res.end('Hello, world!');
  }
});
```

The screenshot shows a Visual Studio Code interface with a dark theme. The left sidebar has icons for various applications like a browser, terminal, and file explorer. The main area shows an 'EXPLORER' view with files 'index.js', 'index2.js', and 'index1.js' under a folder 'MOBZWAY'. The 'index2.js' file is open and contains the provided code. Below the editor is a terminal window showing the command 'node "/home/anuj/Desktop/Mobzway/index2.js"' being run from a terminal session on a 'vostro-15-3568' laptop. The output shows the server is running at 'http://localhost:8000'. The bottom status bar indicates the code is in JavaScript mode.

Breakdown of the Example

1. **Create HTTP Server:** An HTTP server is created.
2. **Blocking Route:** If the request URL is `/block`, the server will enter a while loop that runs for 10 seconds.
3. **Non-blocking Route:** For any other request URL, the server responds with "Hello, world!".

When a request is made to `/block`, the server will not be able to handle any other requests until the 10-second loop completes. This blocks the event loop and makes the server unresponsive during that time.

How to Avoid Blocking the Event Loop

1. **Use Asynchronous Operations:** Always prefer asynchronous methods over synchronous ones. For example, use `fs.readFile` instead of `fs.readFileSync`.
2. **Break Long-Running Tasks:** Split long-running tasks into smaller chunks and use `setImmediate` or `process.nextTick` to yield control back to the event loop.
3. **Offload Intensive Tasks:** Offload CPU-intensive tasks to worker threads or child processes.

Module 2: Node.js Platform Setup

- **Node REPL**

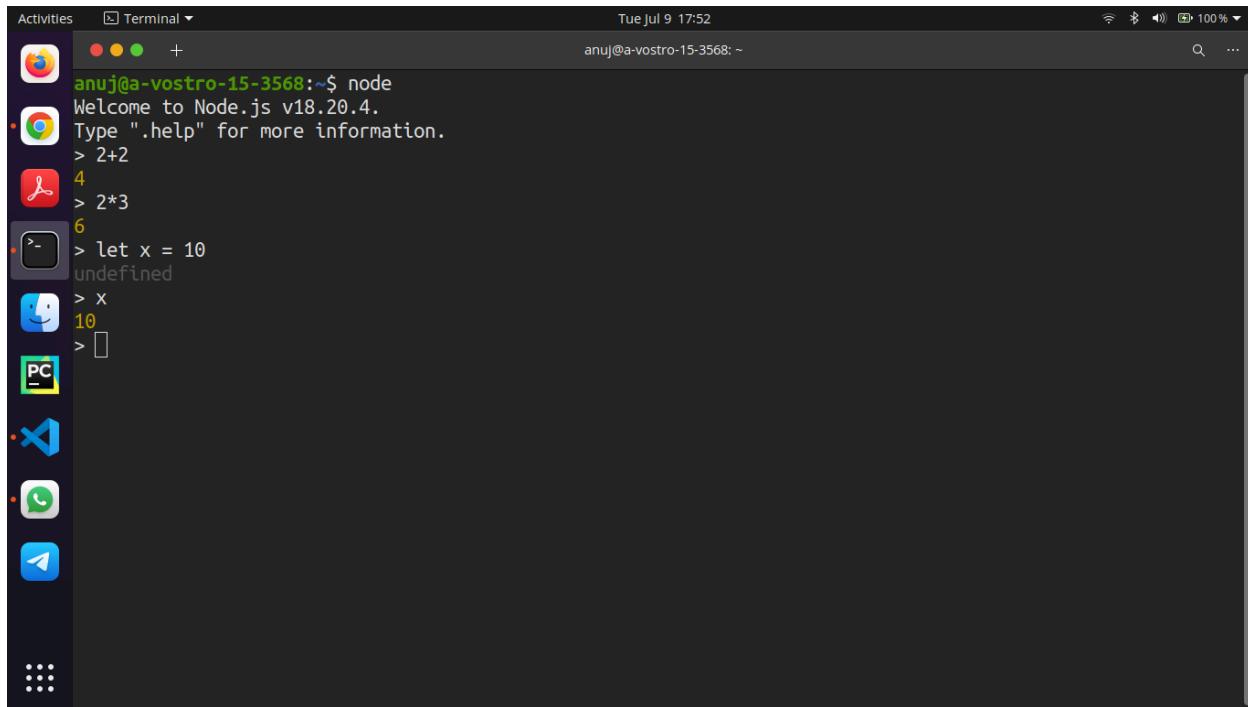
The Node.js REPL (Read-Eval-Print Loop) is an interactive shell that allows you to enter JavaScript code and immediately see the results. It's a useful tool for testing and experimenting with Node.js code snippets in real-time.

Starting the Node.js REPL

To start the Node.js REPL, simply open your terminal or command prompt and type: `node`

This will start the REPL, and you'll see a prompt (`>`) where you can start typing JavaScript code.

Basic Operations in the REPL



The screenshot shows a Linux desktop environment with a dark theme. On the left is a vertical dock containing icons for various applications: a browser, a terminal, a code editor, a file manager, a messaging app, and a file transfer app. The main window is a terminal titled 'Terminal' with the command 'node' entered. The output shows the Node.js version (v18.20.4), a welcome message, and several basic arithmetic operations and variable assignments.

```
anuj@a-vostro-15-3568:~$ node
Welcome to Node.js v18.20.4.
Type ".help" for more information.
> 2+2
4
> 2*3
6
> let x = 10
undefined
> x
10
> []
```

- **Download and Install Node.js**

To download and install Node.js, follow these steps based on your operating system:

For Windows

1. **Download Node.js Installer:**

- Go to the Node.js download page.
- Click the Windows Installer button to download the `.msi` file.

2. **Run the Installer:**

- Locate the downloaded `.msi` file and double-click to run it.
- Follow the prompts in the Node.js Setup Wizard. Accept the license agreement and choose the installation options.
- Make sure to check the box that says "Automatically install the necessary tools" (if available).

3. Verify Installation:

- Open Command Prompt (`cmd`) or PowerShell.
- Type the following commands to verify the installation: `node -v`
- `npm -v`

For macOS

1. Download Node.js Installer:

- Go to the Node.js download page.
- Click the macOS Installer button to download the `.pkg` file.

2. Run the Installer:

- Locate the downloaded `.pkg` file and double-click to run it.
- Follow the prompts in the Node.js Installer. Accept the license agreement and proceed with the installation.

3. Verify Installation:

- Open Terminal.
- Type the following commands to verify the installation: `node -v`
- `npm -v`

• Import Required Modules

In Node.js, you can import required modules using the `require` function for CommonJS modules, which is the module system used by Node.js by default. Here's how you can import some commonly used modules:

Core Modules

```
const fs = require('fs'); // File system module
const http = require('http'); // HTTP module for creating web servers
const path = require('path'); // Path module for working with file and directory paths
```

```
const os = require('os'); // OS module for getting operating system-related utility methods and properties
const util = require('util'); // Utilities module
```

External Modules

External modules are installed via npm (Node Package Manager). For example, to use the `express` module, you first need to install it using npm:

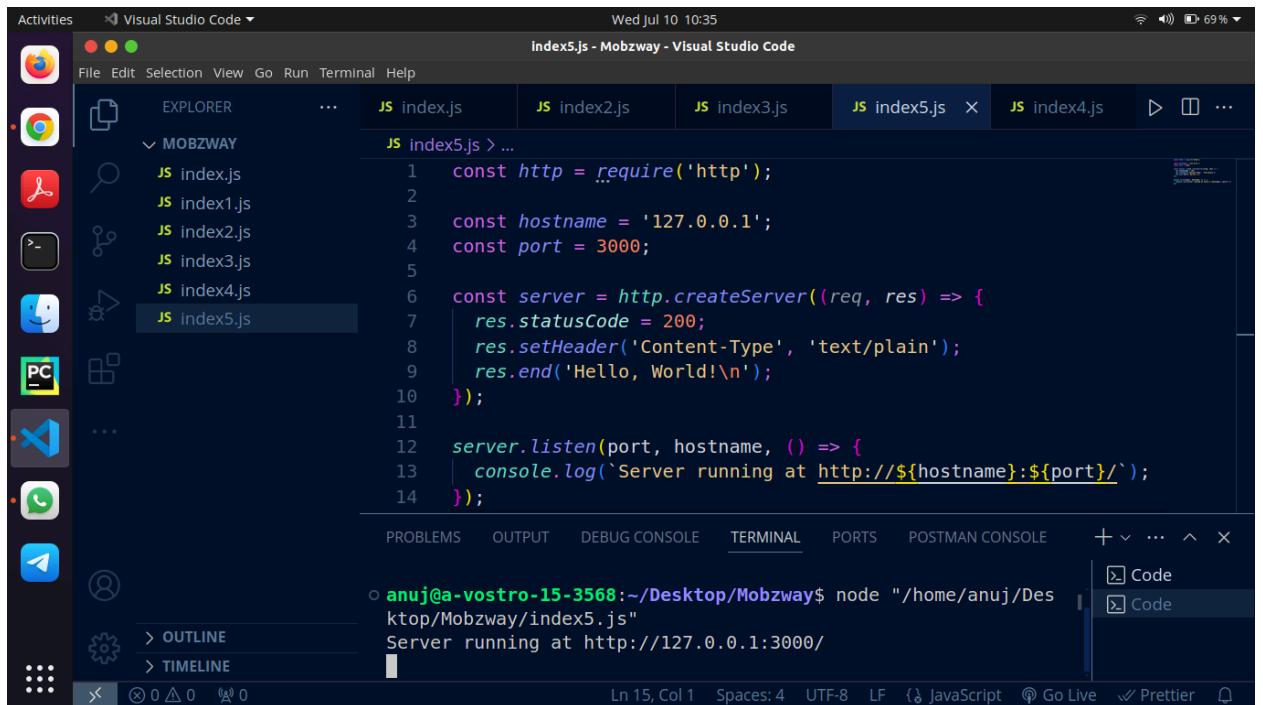
```
npm install express
```

Then, you can import it in your Node.js application.

• **Creating Web Server**

Creating a web server in Node.js is straightforward, I'll show you how to create a web server: using the built-in `http` module.

Using the `http` Module



The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows a folder named "MOBZWAY" containing files: index.js, index1.js, index2.js, index3.js, index4.js, and index5.js. "index5.js" is currently selected.
- Code Editor:** Displays the following code for "index5.js":

```
1 const http = require('http');
2
3 const hostname = '127.0.0.1';
4 const port = 3000;
5
6 const server = http.createServer((req, res) => {
7   res.statusCode = 200;
8   res.setHeader('Content-Type', 'text/plain');
9   res.end('Hello, World!\n');
10 });
11
12 server.listen(port, hostname, () => {
13   console.log(`Server running at ${hostname}:${port}/`);
14 });
```

- Terminal:** Shows the command "node /home/anuj/Desktop/Mobzway/index5.js" being run, followed by the output "Server running at http://127.0.0.1:3000/".
- Status Bar:** Shows the file is "index5.js - Mobzway - Visual Studio Code", the date and time "Wed Jul 10 10:35", and the battery level "69%".

let's break down this code step by step:

Importing the `http` Module

This will import the built-in `http` module, which provides the functionality to create an HTTP server.

Setting the Hostname and Port

Here, `hostname` is set to '`127.0.0.1`', which is the loopback IP address (localhost). The `port` is set to `3000`. These are the address and port number where the server will listen for incoming requests.

Creating the Server

1. `http.createServer((req, res) => { ... })`: This function creates an HTTP server. It takes a callback function as an argument. This callback function is called every time a request is received by the server.
2. `(req, res)`: The callback function takes two arguments:
 - `req`: An object representing the incoming request.
 - `res`: An object representing the response that will be sent to the client.
3. `res.statusCode = 200;`: This line sets the status code of the response to `200`, which means "OK".
4. `res.setHeader('Content-Type', 'text/plain');`: This line sets the `Content-Type` header of the response to `text/plain`. This tells the client that the response body is plain text.
5. `res.end('Hello, World!\n');`: This line ends the response and sends the string '`Hello, World!\n`' to the client.

Starting the Server

1. `server.listen(port, hostname, () => { ... })`: This method starts the server and makes it listen on the specified `port` and `hostname`.

2. `() => { ... }`: This is a callback function that is executed once the server starts successfully.

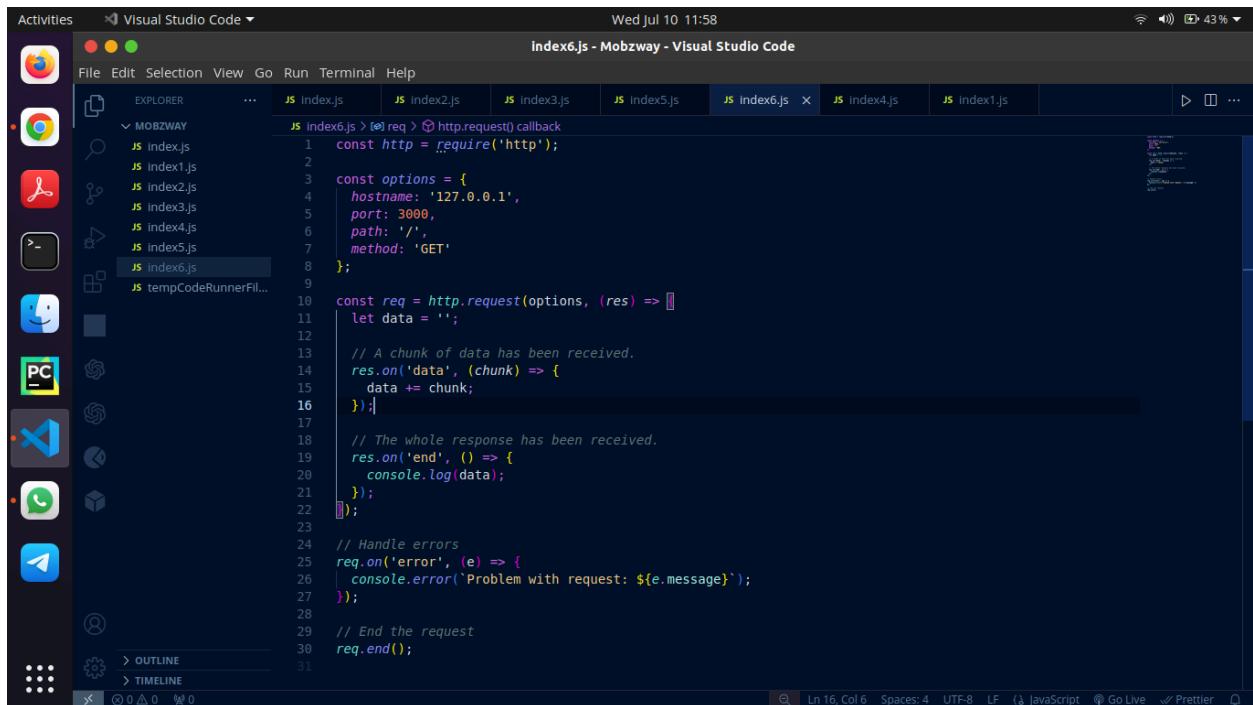
3. `console.log(Server running at http://${hostname}:${port}/);`: This line logs a message to the console, indicating that the server is running and providing the URL where it can be accessed.

- **Sending Request**

Here is the example of sending HTTP requests using http module in Node.js:

Using the Built-in http Module

Here's an example of how to send a GET request using the `http` module:



```

Activities  Visual Studio Code ▾
File Edit Selection View Go Run Terminal Help
EXPLORE... JS index.js JS index2.js JS index3.js JS index5.js JS index6.js × JS index4.js JS index1.js ...
MOBZWAY
JS index.js
JS index1.js
JS index2.js
JS index3.js
JS index4.js
JS index5.js
JS index6.js
JS tempCodeRunnerFil...
1 const http = require('http');
2
3 const options = {
4   hostname: '127.0.0.1',
5   port: 3008,
6   path: '/',
7   method: 'GET'
8 };
9
10 const req = http.request(options, (res) => [
11   let data = '';
12
13   // A chunk of data has been received.
14   res.on('data', (chunk) => {
15     data += chunk;
16   });
17
18   // The whole response has been received.
19   res.on('end', () => {
20     console.log(data);
21   });
22 ]);
23
24 // Handle errors
25 req.on('error', (e) => {
26   console.error(`Problem with request: ${e.message}`);
27 });
28
29 // End the request
30 req.end();
31

```

Setting Up Request Options

Here, an `options` object is created to specify the details of the HTTP request:

- `hostname`: The server address. In this case, it's the loopback IP address (`localhost`).
- `port`: The port number to connect to on the server.
- `path`: The URL path to request from the server.
- `method`: The HTTP method to use for the request (GET, in this case).

Making the Request

`http.request(options, callback)`: This function creates a new HTTP request using the provided `options` and a callback function that handles the response.

1. `Callback function (res) => { ... }:` This function is called when the server responds. It receives a `res` (response) object.
2. `res.on('data', (chunk) => { ... })`: This event listener handles the incoming data chunks. The data received is appended to the `data` variable.
3. `res.on('end', () => { ... })`: This event listener is called when the entire response has been received. It logs the complete response data to the console.

Handling Errors

`req.on('error', (callback))`: This event listener is called if there is an error with the request. It logs the error message to the console.

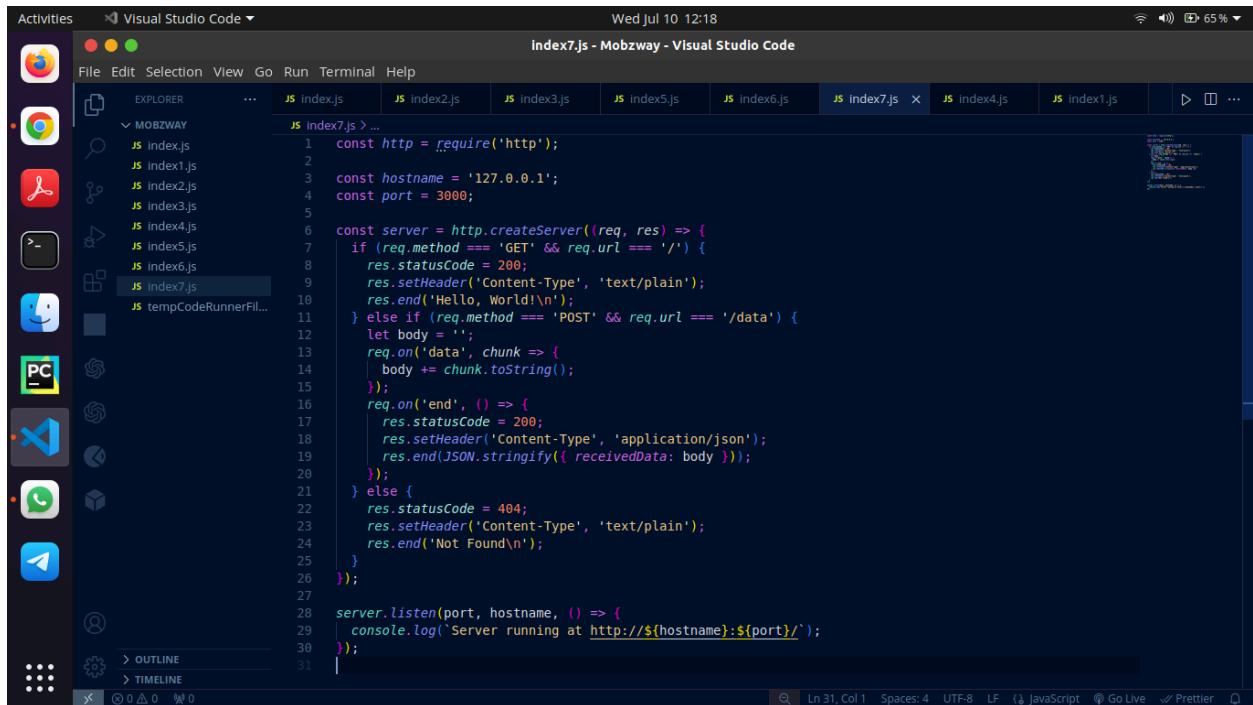
Ending the Request

`req.end()`: This function signals that the request has been completed and no more data will be sent. It is necessary to call `req.end()` to actually send the request.

- **Handling HTTP requests**

Handling HTTP requests in Node.js typically involves creating a server that listens for incoming requests and responds appropriately. This can be done using the built-in `http` module.

Using the Built-in `http` Module



The screenshot shows the Visual Studio Code interface with the following details:

- Title Bar:** Activities, Visual Studio Code, index7.js - Mobzway - Visual Studio Code, Wed Jul 10 12:18
- File Explorer:** Shows files in the MOBZWAY folder: index.js, index1.js, index2.js, index3.js, index4.js, index5.js, index6.js, index7.js (selected), and tempCodeRunnerFile.js.
- Code Editor:** Displays the following JavaScript code for a simple HTTP server:

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  if (req.method === 'GET' && req.url === '/') {
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hello, World!\n');
  } else if (req.method === 'POST' && req.url === '/data') {
    let body = '';
    req.on('data', chunk => {
      body += chunk.toString();
    });
    req.on('end', () => {
      res.statusCode = 200;
      res.setHeader('Content-Type', 'application/json');
      res.end(JSON.stringify({ receivedData: body }));
    });
  } else {
    res.statusCode = 404;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Not Found\n');
  }
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```
- Status Bar:** Ln 31, Col 1, Spaces: 4, UTF-8, LF, JavaScript, Go Live, Prettier

Handling GET Requests

1. **Checking the Request Method and URL:** This `if` statement checks if the request method is `GET` and the request URL is `/`.
2. **Setting the Status Code:** `res.statusCode = 200;` sets the HTTP status code of the response to `200`, which means "OK".
3. **Setting the Header:** `res.setHeader('Content-Type', 'text/plain');` sets the `Content-Type` header of the response to `text/plain`, indicating that the response body is plain text.
4. **Ending the Response:** `res.end('Hello, World!\n');` ends the response and sends the string `'Hello, World!\n'` to the client.

Handling POST Requests

Checking the Request Method and URL: This `else if` statement checks if the request method is `POST` and the request URL is `/data`.

1. Handling Incoming Data:

- `let body = '';` initializes an empty string to accumulate the incoming data chunks.
- `req.on('data', chunk => { body += chunk.toString(); })`; sets up an event listener for the `data` event, which is emitted when a chunk of data is received. The received chunk is converted to a string and appended to the `body` variable.

2. Handling End of Data:

- `req.on('end', () => { ... })`; sets up an event listener for the `end` event, which is emitted when all data has been received.
- Inside the `end` event listener:
 - `res.statusCode = 200`; sets the HTTP status code of the response to `200`, indicating success.
 - `res.setHeader('Content-Type', 'application/json')`; sets the `Content-Type` header to `application/json`, indicating that the response body is JSON.
 - `res.end(JSON.stringify({ receivedData: body }))`; ends the response and sends a JSON string containing the received data back to the client.

- Write your First Hello-World Program

The screenshot shows the Visual Studio Code interface with a dark theme. The left sidebar has a 'MOBZWAY' folder containing files: index.js, index1.js, index2.js, index3.js, index4.js, index5.js (which is the active file), index6.js, index7.js, and tempCodeRunnerFile.js. The main editor area displays the following code:

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 3000;

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

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

The terminal tab at the bottom shows the command `node "/home/anuj/Desktop/Mobzway/index5.js"` being run, followed by the output: `anuj@anuj-Vostro-15-3568:~/Desktop/Mobzway\$ node "/home/anuj/Desktop/Mobzway/index5.js" Server running at http://127.0.0.1:3000/`.

Explanation

1. **Import the `http` Module:** The first line imports the built-in `http` module, which is required to create an HTTP server.
 2. **Define the Hostname and Port:** The `hostname` is set to '`127.0.0.1`' (localhost), and the `port` is set to `3000`.
 3. **Create the HTTP Server:** The `http.createServer` function creates a new HTTP server. The server takes a callback function that is executed each time a request is received.
 4. **Start the Server:** The `server.listen` function starts the server and makes it listen on the specified port and hostname. When the server starts, a message is logged to the console.
- Deploy your node app on any platform like Heroku, Render and firebase

Deploying on Heroku

1. **Install Heroku CLI:** If you haven't already, download and install the Heroku CLI.
2. **Login to Heroku:** Open your terminal and run: heroku login
3. **Prepare Your Application:** Ensure your `helloWorld.js` and `package.json` are in the same directory. Your `package.json` should look something like this:

```
{  
  "name": "hello-world",  
  "version": "1.0.0",  
  "main": "helloWorld.js",  
  "scripts": {  
    "start": "node helloWorld.js"  
  }  
}
```

4. **Initialize a Git Repository:**

```
git init  
git add .  
git commit -m "Initial commit"
```

5. **Create a Heroku App:**

```
heroku create
```

6. Deploy Your Code:

```
git push heroku master
```

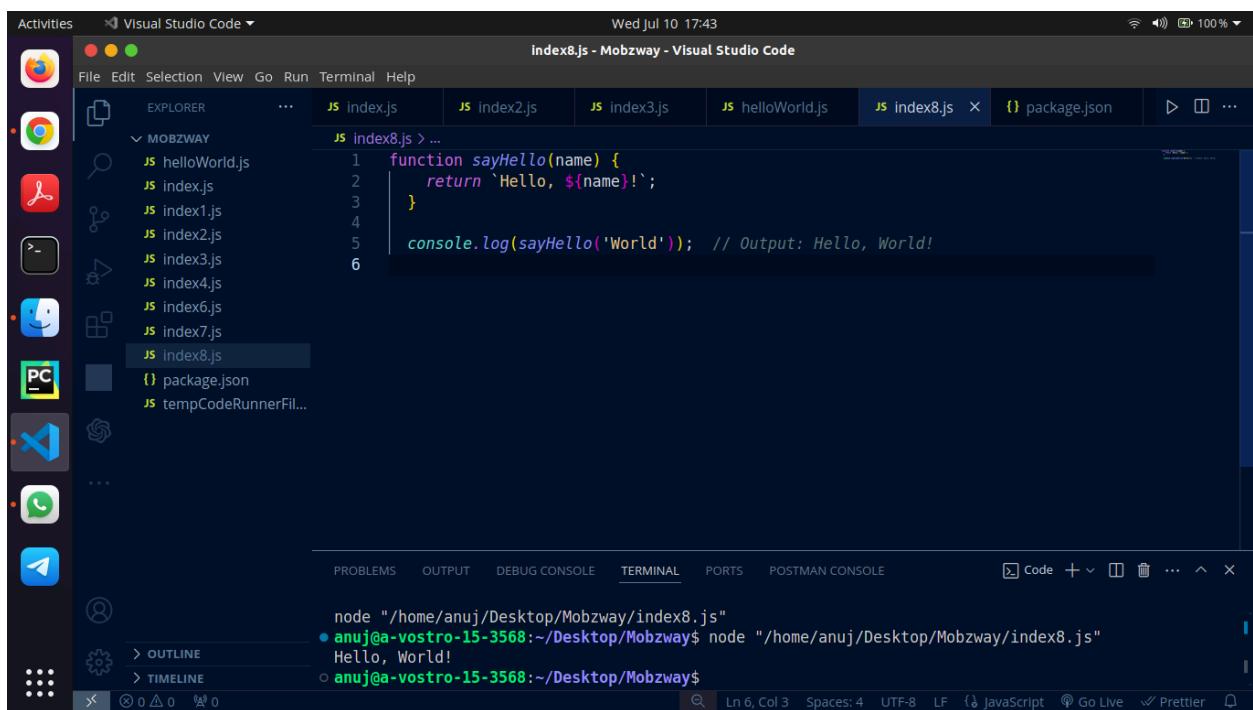
7. Open Your App:

```
heroku open
```

Module 3: Node JS Modules

● Functions

Named Functions



The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows files in the "MOBZWAY" folder: helloWorld.js, index.js, index1.js, index2.js, index3.js, index4.js, index6.js, index7.js, and index8.js (which is currently selected).
- Code Editor:** Displays the content of index8.js:

```
function sayHello(name) {
  return `Hello, ${name}!`;
}

console.log(sayHello('World')); // Output: Hello, World!
```
- Terminal:** Shows the command "node index8.js" being run, with the output "Hello, World!".
- Bottom Status Bar:** Shows file information like "Ln 6, Col 3" and "Spaces: 4".

Anonymous Functions

The screenshot shows a Visual Studio Code interface with a dark theme. The left sidebar has a 'MOBZWAY' folder containing files like index.js, index2.js, index3.js, helloWorld.js, index8.js, package.json, and tempCodeRunnerFile... The main editor window displays the following code:

```
1 const sayHello = function(name) {
2   return `Hello, ${name}!`;
3 }
4
5 console.log(sayHello('World')); // Output: Hello, World!
```

The terminal at the bottom shows the output of running the script:

```
Hello, World!
● anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/index8.js"
Hello, World!
○ anuj@a-vostro-15-3568:~/Desktop/Mobzway$
```

Arrow Functions

The screenshot shows a Visual Studio Code interface with a dark theme. The left sidebar has a 'MOBZWAY' folder containing files like index.js, index2.js, index3.js, helloWorld.js, index8.js, package.json, and tempCodeRunnerFile... The main editor window displays the following code:

```
1 const sayHello = (name) => {
2   return `Hello, ${name}!`;
3 }
4
5 console.log(sayHello('World')); // Output: Hello, World!
```

The terminal at the bottom shows the output of running the script:

```
Hello, World!
● anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/index8.js"
Hello, World!
○ anuj@a-vostro-15-3568:~/Desktop/Mobzway$
```

Asynchronous Functions

1. Callbacks

The screenshot shows the Visual Studio Code interface with the title bar "index8.js - Mobzway - Visual Studio Code" and the date "Wed Jul 10 17:47". The Explorer sidebar on the left shows a folder named "MOBZWAY" containing several JavaScript files: "helloWorld.js", "index.js", "index1.js", "index2.js", "index3.js", "index4.js", "index6.js", "index7.js", and "index8.js". The "index8.js" file is currently selected and open in the editor. The code in the editor is:

```
1 const fs = require('fs');
2
3 fs.readFile('example.txt', 'utf8', (err, data) => {
4   if (err) {
5     console.error(err);
6     return;
7   }
8   console.log(data);
9 });
10 
```

The terminal at the bottom shows the output of the code execution:

```
syscall: 'open',
path: 'example.txt'

```

The status bar at the bottom right indicates the terminal is running on "anuj@a-vostro-15-3568:~/Desktop/Mobzway\$".

2. Promises

The screenshot shows the Visual Studio Code interface with the title bar "index8.js - Mobzway - Visual Studio Code" and the date "Wed Jul 10 17:48". The Explorer sidebar on the left shows a folder named "MOBZWAY" containing several JavaScript files: "helloWorld.js", "index.js", "index1.js", "index2.js", "index3.js", "index4.js", "index6.js", "index7.js", and "index8.js". The "index8.js" file is currently selected and open in the editor. The code in the editor is:

```
1 const fs = require('fs').promises;
2
3 fs.readFile('example.txt', 'utf8')
4   .then(data => {
5     console.log(data);
6   })
7   .catch(err => {
8     console.error(err);
9   });
10 
```

The terminal at the bottom shows the output of the code execution:

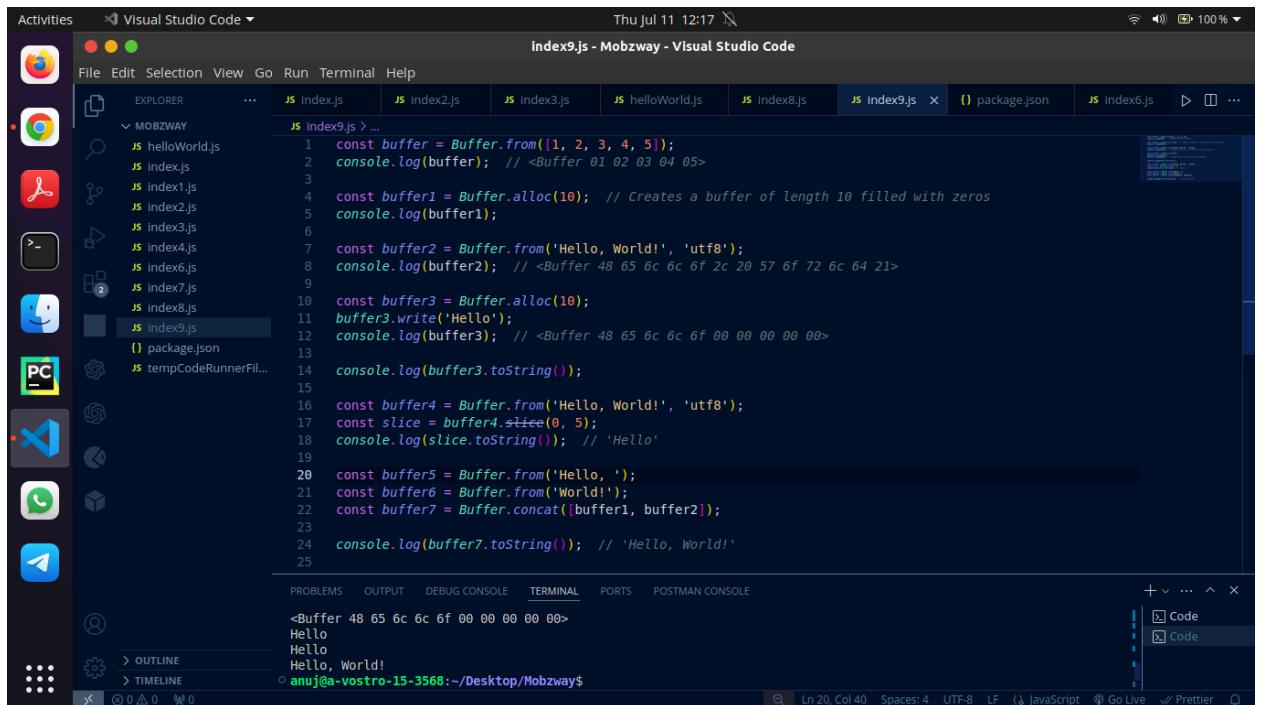
```
syscall: 'open',
path: 'example.txt'

```

The status bar at the bottom right indicates the terminal is running on "anuj@a-vostro-15-3568:~/Desktop/Mobzway\$".

- **Buffer**

Buffers in Node.js are used to handle binary data. They are particularly useful when dealing with file streams, TCP streams, and other types of data where the encoding is not always UTF-8. Buffers allow for the manipulation and storage of raw binary data.



```
const buffer = Buffer.from([1, 2, 3, 4, 5]);
console.log(buffer); // <Buffer 01 02 03 04 05>

const buffer1 = Buffer.alloc(10); // Creates a buffer of length 10 filled with zeros
console.log(buffer1);

const buffer2 = Buffer.from('Hello, World!', 'utf8');
console.log(buffer2); // <Buffer 48 65 6c 6c 6f 2c 20 57 6f 72 6c 64 21>

const buffer3 = Buffer.alloc(10);
buffer3.write('Hello');
console.log(buffer3); // <Buffer 48 65 6c 6c 6f 00 00 00 00 00>

console.log(buffer3.toString());

const buffer4 = Buffer.from('Hello, World!', 'utf8');
const slice = buffer4.slice(0, 5);
console.log(slice.toString()); // 'Hello'

const buffer5 = Buffer.from('Hello, ');
const buffer6 = Buffer.from('World!');
const buffer7 = Buffer.concat([buffer1, buffer2]);

console.log(buffer7.toString()); // 'Hello, World!'
```

Allocating a Buffer

You can create a buffer of a specified size using `Buffer.alloc`.

From an Array

You can create a buffer from an array of bytes.

From a String

You can create a buffer from a string.

Writing to a Buffer

Reading from a Buffer

Slicing Buffers

You can create a new buffer that references the same memory as the original buffer but only represents a subset of it.

Concatenating Buffers

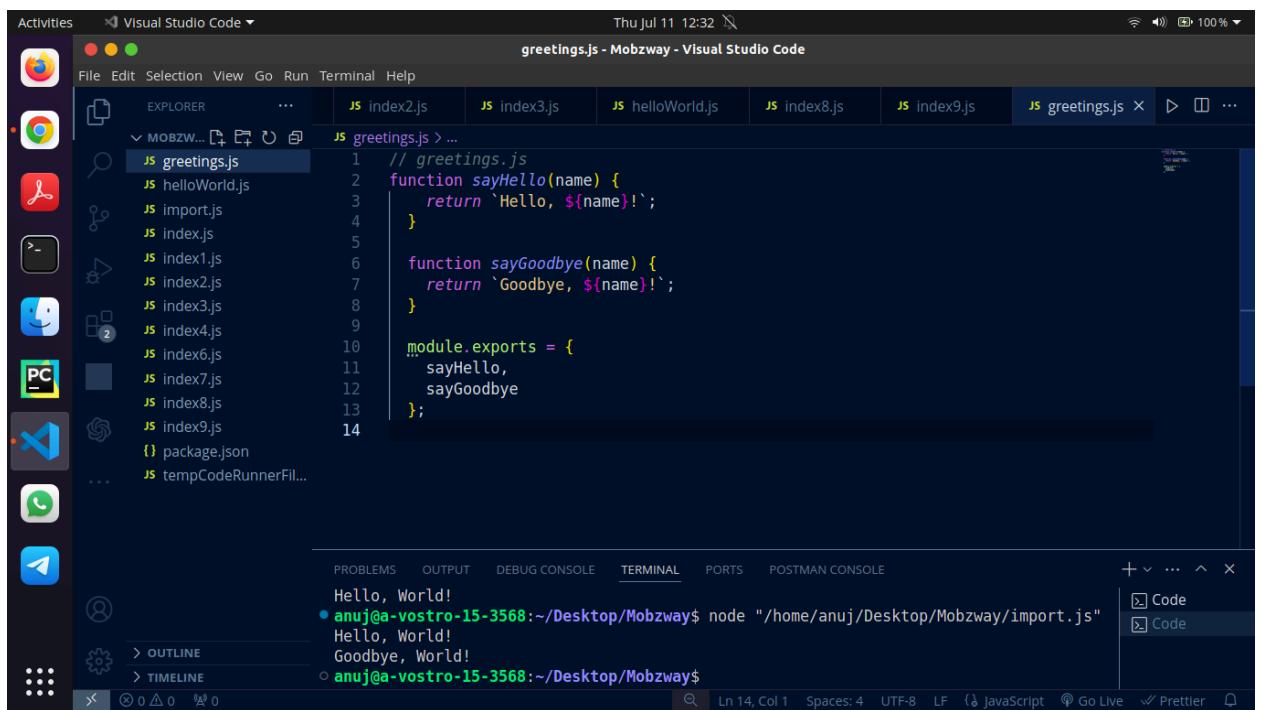
You can concatenate multiple buffers into a single buffer.

● **Module**

Modules in Node.js are a way to organize and encapsulate code into reusable components. Each module in Node.js is treated as a separate file, and you can export and import functions, objects, and values between modules. This modular approach helps in maintaining clean and manageable code.

Creating and Exporting a Module

You can create a module by defining functions, objects, or values and exporting them using `module.exports` or `exports`.



The screenshot shows a Visual Studio Code interface with the following details:

- File Explorer:** Shows files like `index2.js`, `index3.js`, `helloWorld.js`, `index8.js`, `index9.js`, and `greetings.js`.
- Code Editor:** The `greetings.js` file is open, containing the following code:

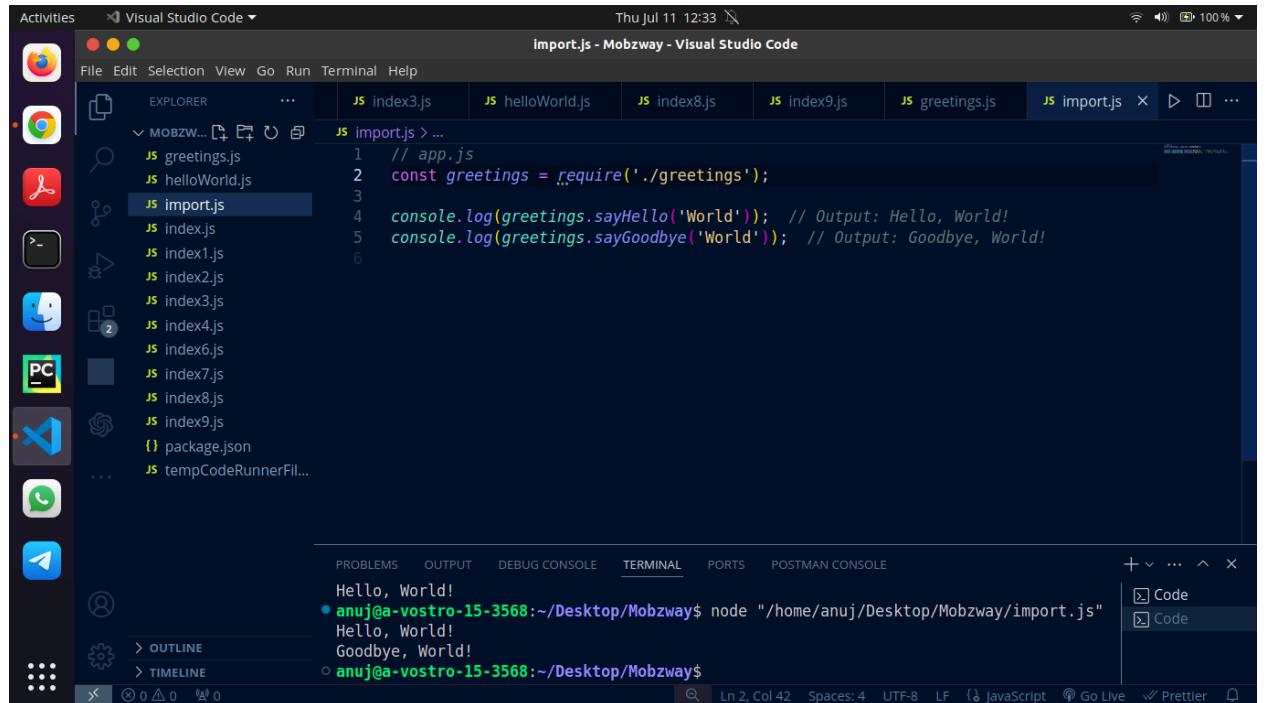
```
// greetings.js
function sayHello(name) {
  return `Hello, ${name}!`;
}

function sayGoodbye(name) {
  return `Goodbye, ${name}!`;
}

module.exports = {
  sayHello,
  sayGoodbye
};
```
- Terminal:** Shows the command `node "/home/anuj/Desktop/Mobzway/import.js"` being run, with the output "Hello, World!" and "Goodbye, World!".
- Status Bar:** Shows "Ln 14, Col 1" and "Spaces: 4".

Importing a Module

To use the exported functions from a module in another file, you use the `require` function.



The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists several JavaScript files: greetings.js, helloWorld.js, import.js, index1.js, index2.js, index3.js, index4.js, index6.js, index7.js, index8.js, index9.js, package.json, and tempCodeRunnerFil... The main editor area displays the content of import.js:

```
// app.js
const greetings = require('./greetings');
console.log(greetings.sayHello('World')); // Output: Hello, World!
console.log(greetings.sayGoodbye('World')); // Output: Goodbye, World!
```

The Terminal tab at the bottom shows the command `node "/home/anuj/Desktop/Mobzway/import.js"` being run, with the output "Hello, World!" followed by "Hello, World!" and "Goodbye, World!".

● **Modules Types**

Built-in (Core) Modules

Node.js comes with a set of built-in modules that provide essential functionalities for building applications. These modules are part of the Node.js runtime and do not require installation.

Examples of Built-in Modules:

- `fs`: File system operations.
- `http`: HTTP server and client functionality.
- `path`: Utilities for working with file and directory paths.

User-defined Modules

These are modules that you create in your Node.js application. They are useful for organizing and encapsulating your code into reusable components.

Third-party Modules

These modules are created by the community and can be installed using the Node Package Manager (npm). They provide additional functionalities that are not part of the core Node.js library.

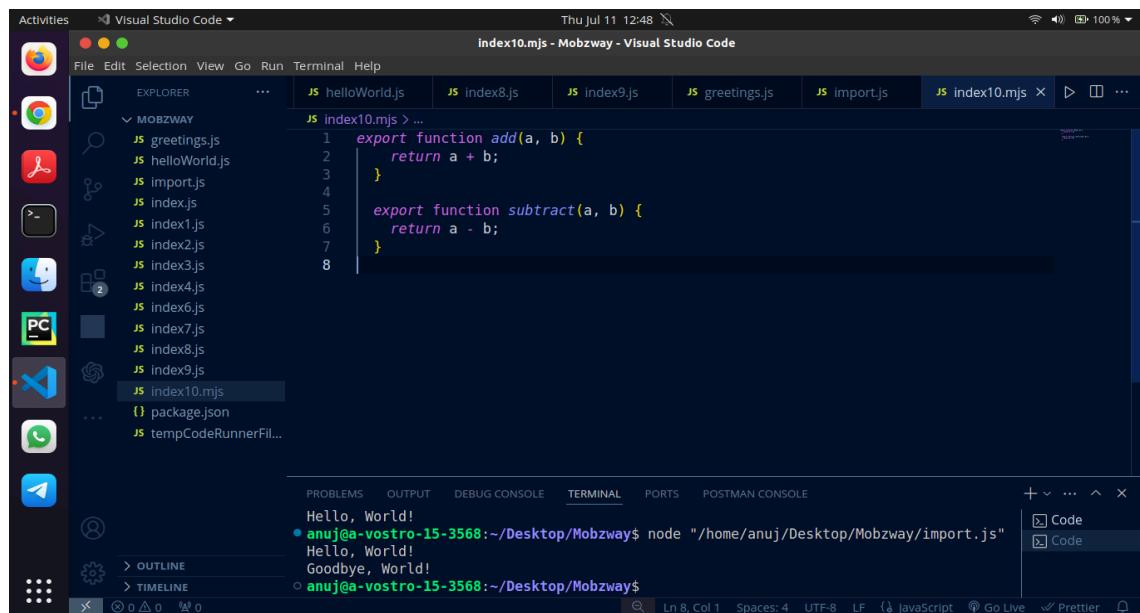
Examples of Popular Third-party Modules:

- **express**: Web application framework.
- **lodash**: Utility library for JavaScript.
- **mongoose**: MongoDB object modeling tool.

ES6 Modules (ECMAScript Modules)

Node.js supports ES6 modules, which use `import` and `export` syntax. This is different from the CommonJS module system that uses `require` and `module.exports`.

An MJS file is a source code file containing an ES Module (ECMAScript Module) for use with a Node.js application.



The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows a folder named "MOBZWAY" containing files: greetings.js, helloWorld.js, import.js, index.js, index1.js, index2.js, index3.js, index4.js, index6.js, index7.js, index8.js, index9.js, and index10.mjs (which is currently selected).
- Code Editor:** Displays the contents of index10.mjs:

```
1 export function add(a, b) {
2   return a + b;
3 }
4
5 export function subtract(a, b) {
6   return a - b;
7 }
```
- Terminal:** Shows the output of running the script: "Hello, World!" followed by "Hello, World!" and "Goodbye, World!".
- Status Bar:** Shows the file path as "/home/anuj/Desktop/Mobzway/import.js", the line number as "Ln 8, Col 1", and the character position as "Spaces: 4, UTF-8, LF".

```

Thu Jul 11 12:52:48
import1.mjs - Mobzway - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER ... JS index9.js JS greetings.js JS import.js JS index10.mjs JS import1.mjs X {} package.json D I ...
MOBZWAY
JS greetings.js
JS helloWorld.js
JS import.js
JS import1.mjs
JS index.js
JS index1.js
JS index2.js
JS index3.js
JS index4.js
JS index6.js
JS index7.js
JS index8.js
JS index9.js
JS index10.mjs
{} package.json
JS tempCodeRunnerFil...
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
Node.js v18.20.4
● anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/import1.mjs"
5
3
○ anuj@a-vostro-15-3568:~/Desktop/Mobzway$
```

To run ES6 modules, you need to use the `.mjs` extension or set "type": "`module`" in your `package.json`.

Local Modules

Local modules are user-defined modules that are stored locally in your project and used within the project. They can be reused across different files of the same project.

● Core Modules

Node.js comes with a set of built-in modules that provide essential functionalities for building applications. These modules are part of the Node.js runtime and do not require installation.

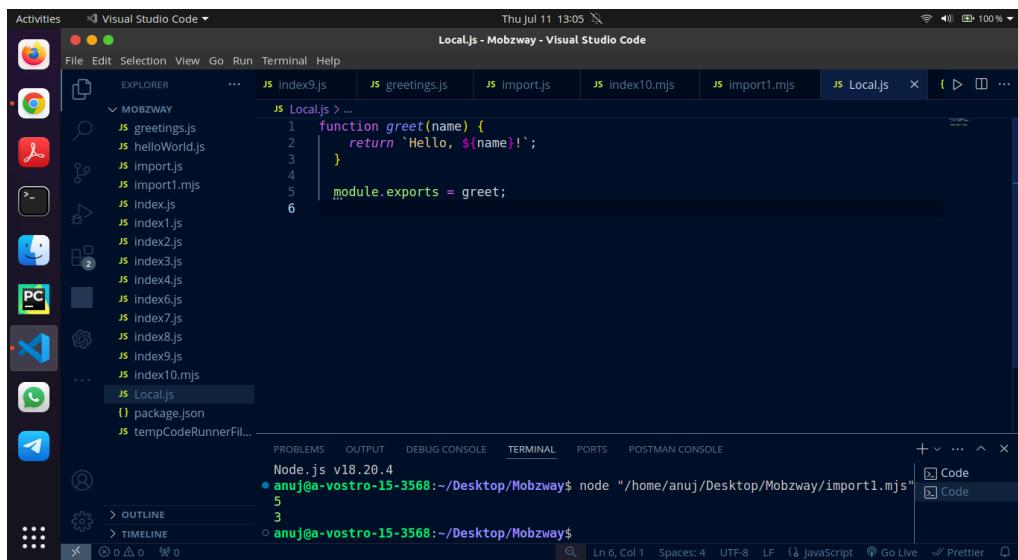
Examples of Built-in Modules:

- `fs`: File system operations.
- `http`: HTTP server and client functionality.
- `path`: Utilities for working with file and directory paths.
- `os`: Information about the operating system.

- **events**: Event-driven programming support.
- **util**: Utilities for various tasks like debugging, deprecation, and inheritance.

● Local Modules

Local modules are user-defined modules that are stored locally in your project and used within the project. They can be reused across different files of the same project.



The screenshot shows the Visual Studio Code interface with the title bar "Thu Jul 11 13:05 Local.js - Mobzway - Visual Studio Code". The Explorer sidebar on the left shows a folder named "MOBZWAY" containing several JavaScript files: greetings.js, helloWorld.js, import.js, import1.mjs, index.js, index1.js, index2.js, index3.js, index4.js, index6.js, index7.js, index8.js, index9.js, index10.mjs, Local.js, package.json, and tempCodeRunnerFile... The Local.js file is open in the editor, displaying the following code:

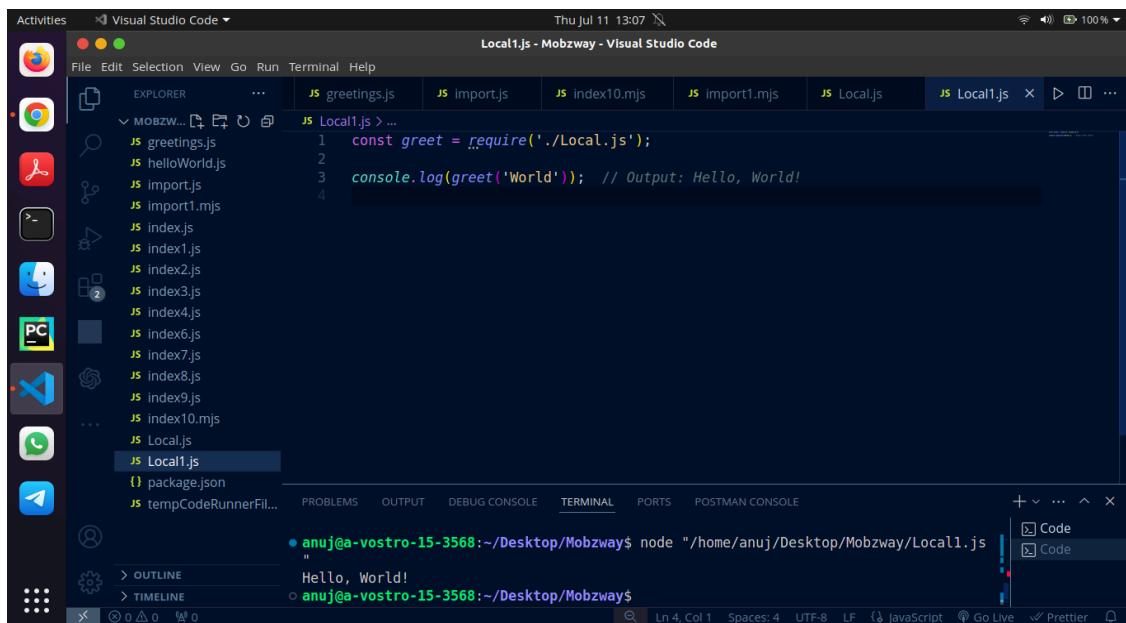
```

function greet(name) {
  return `Hello, ${name}!`;
}

module.exports = greet;

```

The Terminal tab at the bottom shows the command "node ./Local.js" being run, with the output "Hello, World!" displayed.



The screenshot shows the Visual Studio Code interface with the title bar "Thu Jul 11 13:07 Local1.js - Mobzway - Visual Studio Code". The Explorer sidebar on the left shows a folder named "MOBZWAY" containing several JavaScript files: greetings.js, helloWorld.js, import.js, import1.mjs, index.js, index1.js, index2.js, index3.js, index4.js, index6.js, index7.js, index8.js, index9.js, index10.mjs, Local.js, Local1.js, package.json, and tempCodeRunnerFile... The Local1.js file is open in the editor, displaying the following code:

```

const greet = require('./Local.js');

console.log(greet('World')) // Output: Hello, World!

```

The Terminal tab at the bottom shows the command "node Local1.js" being run, with the output "Hello, World!" displayed.

- **Modules Exports**

In Node.js, the `module.exports` object is used to define what a module exports and makes available for other modules to import. This allows you to organize your code into reusable components.

Here's a detailed explanation of how `module.exports` works and various ways to use it:

Basic Usage

Exporting a Single Function or Object

When you want to export a single function or object, you can assign it directly to `module.exports`.

Exporting Multiple Functions or Objects

If you need to export multiple functions or objects, you can add them as properties of the `module.exports` object.

Module 4: Node Package Manager

- **What is NPM**

npm (Node Package Manager) is a package manager for JavaScript, primarily used for managing dependencies of Node.js projects. It is the default package manager for the Node.js runtime environment and allows developers to share and reuse code. npm also provides a command-line interface (CLI) for interacting with the npm ecosystem.

- **Installing Packages Locally**

To install a package locally, use the `npm install` command followed by the package name. This installs the package into the `node_modules` directory and adds it to the `dependencies` section of your `package.json` file.

```
npm install <package-name>
```

- **Installing Package globally**

To install packages globally using npm, you use the `-g` flag with the `npm install` command. Global installation makes the package available system-wide, meaning it can be used from any directory on your system, which is particularly useful for command-line tools.

To install a package globally, use the `-g` flag:

```
npm install -g <package-name>
```

- **Adding dependencies in package.json**

When you install a package using `npm install`, it automatically adds the package to the `dependencies` or `devDependencies` section of your `package.json` file, depending on the flags you use.

Adding a Dependency:

```
npm install <package-name> --save
```

The `--save` flag is actually the default behavior and can be omitted. This command installs the package and adds it to the `dependencies` section.

- **Updating Packages**

Updating packages in a Node.js project can involve different approaches depending on the specific needs. Here's how you can update packages to the latest versions or to specific versions.

Using npm Update Command

The `npm update` command updates all the packages listed in the `package.json` file to their latest versions, respecting the version constraints specified.

```
npm update
```

Updating Specific Packages

To update a single package to the latest version:

```
npm install <package-name>@latest
```

Module 5: File System

- **Read File**

In Node.js, you can read files using the built-in `fs` (File System) module. There are multiple ways to read files, including synchronous and asynchronous methods. Here's how you can read files using both approaches:

Asynchronous File Reading

Using the asynchronous method is generally preferred for non-blocking code execution, which is crucial for performance in a Node.js environment.

A screenshot of the Visual Studio Code interface. The title bar shows "Activities" and "Visual Studio Code". The status bar indicates "Mon Jul 15 12:34". The main area displays an asynchronous JavaScript file named "Async.js" with the following code:

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

The Explorer sidebar on the left lists various files in the project, including "index.js", "index2.js", "index3.js", "helloWorld.js", and "Sync.js". The "Sync.js" file is currently selected.

Synchronous File Reading

Synchronous methods are simpler but can block the event loop, making them less suitable for performance-critical applications.

A screenshot of the Visual Studio Code interface. The title bar shows "Activities" and "Visual Studio Code". The status bar indicates "Mon Jul 15 12:36". The main area displays a synchronous JavaScript file named "Sync.js" with the following code:

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

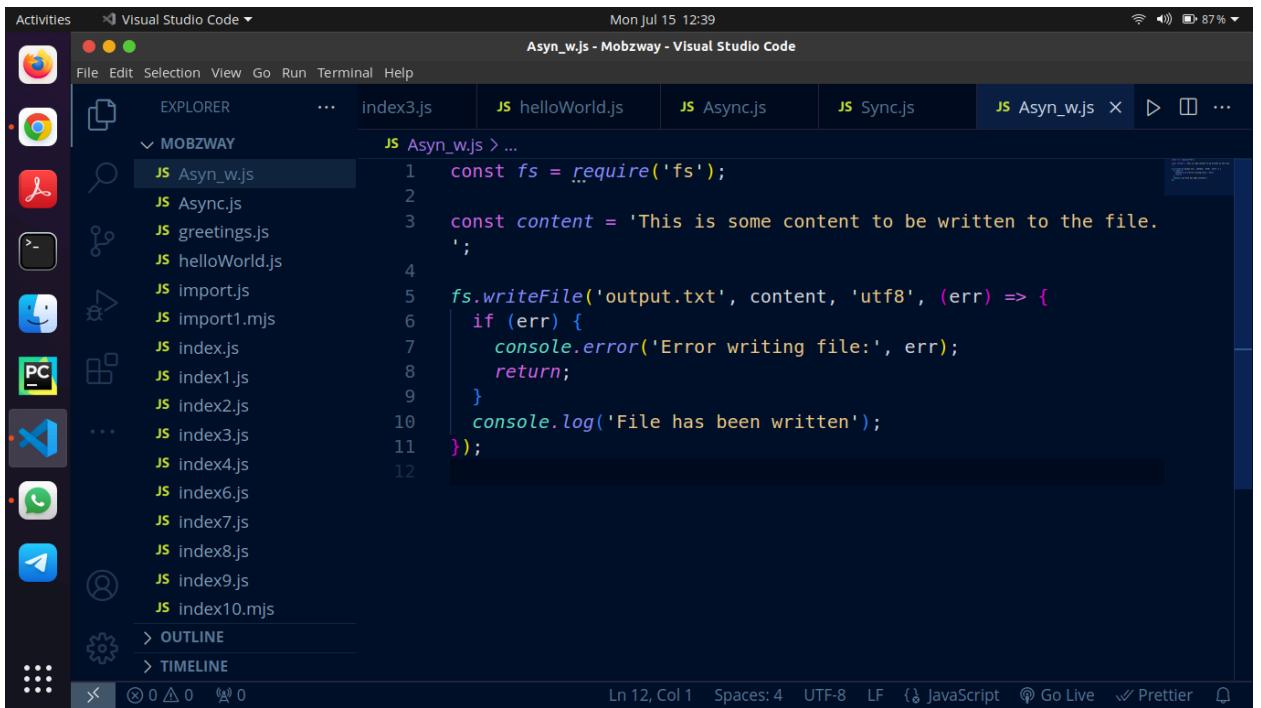
The Explorer sidebar on the left lists various files in the project, including "index2.js", "index3.js", "helloWorld.js", "Async.js", and "Sync.js". The "Sync.js" file is currently selected.

- **Writing a File**

Writing a file in Node.js can be done using the built-in `fs` (File System) module. Like reading files, you have multiple options to write files, including synchronous and asynchronous methods. Here's how you can write files using both approaches:

Asynchronous File Writing

Asynchronous file writing is preferred to avoid blocking the event loop.



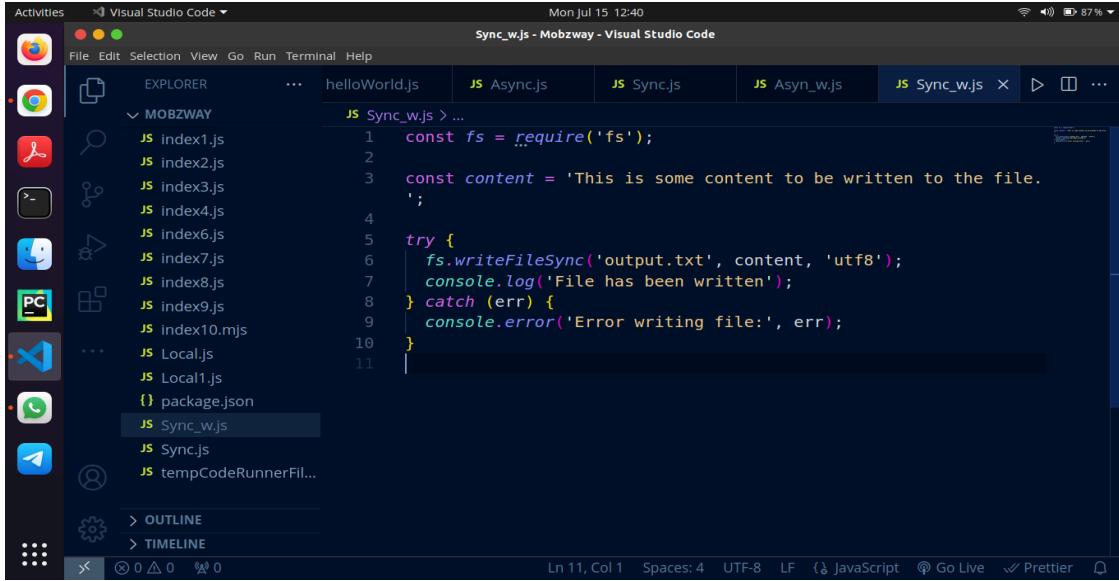
A screenshot of the Visual Studio Code interface. The title bar says "Mon Jul 15 12:39 Asyn_w.js - Mobzway - Visual Studio Code". The left sidebar shows a file tree with a folder "MOBZWAY" containing files like "index3.js", "JS Async.js", "JS greetings.js", "JS helloWorld.js", "JS import.js", "JS import1.mjs", "JS index.js", "JS index1.js", "JS index2.js", "JS index3.js", "JS index4.js", "JS index6.js", "JS index7.js", "JS index8.js", "JS index9.js", and "JS index10.mjs". The main editor tab is "JS Asyn_w.js", which contains the following code:

```
1  const fs = require('fs');
2
3  const content = 'This is some content to be written to the file.
4
5  fs.writeFile('output.txt', content, 'utf8', (err) => {
6    if (err) {
7      console.error('Error writing file:', err);
8      return;
9    }
10   console.log('File has been written');
11 });
12 
```

The status bar at the bottom shows "Ln 12, Col 1 Spaces: 4 UTF-8 LF ⚡ JavaScript ⚡ Go Live ⚡ Prettier".

Synchronous File Writing

Synchronous methods are simpler but can block the event loop.



A screenshot of the Visual Studio Code interface. The title bar shows "Sync_w.js - Mobzway - Visual Studio Code". The left sidebar has a "MOBZWAY" folder containing various JavaScript files like index1.js, index2.js, etc. The main editor tab is "Sync_w.js" which contains the following code:

```
const fs = require('fs');
const content = 'This is some content to be written to the file.

try {
  fs.writeFileSync('output.txt', content, 'utf8');
  console.log('File has been written');
} catch (err) {
  console.error('Error writing file:', err);
}
```

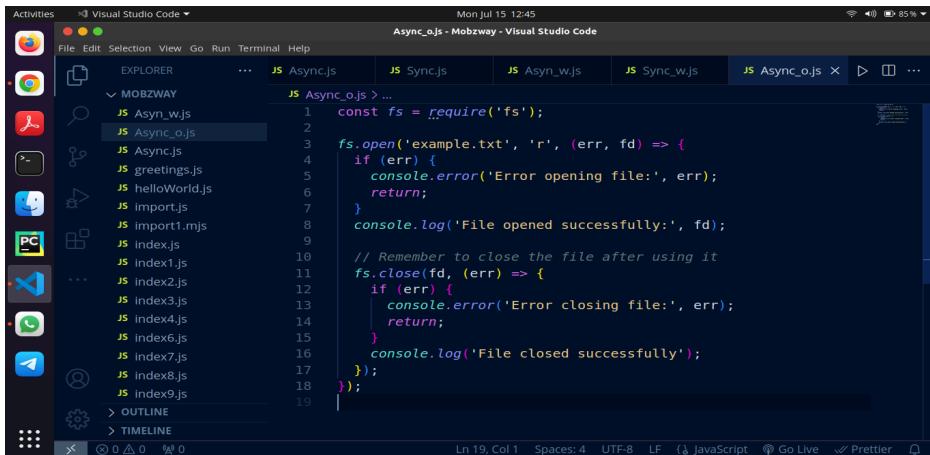
The status bar at the bottom shows "Ln 11, Col 1" and other settings.

● Opening a File

In Node.js, opening a file involves using the built-in `fs` (File System) module. You can open a file to read from or write to it using asynchronous or synchronous methods. Opening a file provides you with a file descriptor, which you can then use for various file operations.

Asynchronous File Opening

Using the asynchronous method is preferred for non-blocking code execution, which is crucial for performance in a Node.js environment.



A screenshot of the Visual Studio Code interface. The title bar shows "Async_o.js - Mobzway - Visual Studio Code". The left sidebar has a "MOBZWAY" folder containing various JavaScript files like Asyn_w.js, Async_o.js, etc. The main editor tab is "Async_o.js" which contains the following code:

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

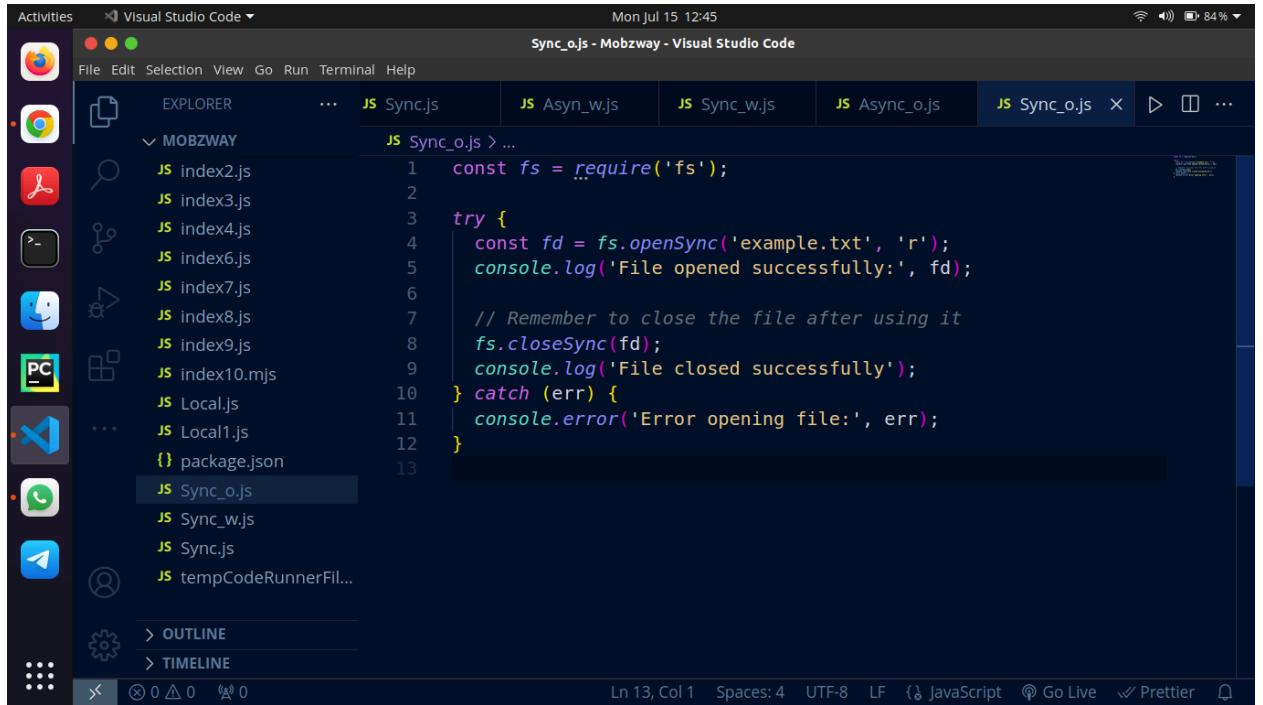
fs.open('example.txt', 'r', (err, fd) => {
  if (err) {
    console.error('Error opening file:', err);
    return;
  }
  console.log('File opened successfully:', fd);

  // Remember to close the file after using it
  fs.close(fd, (err) => {
    if (err) {
      console.error('Error closing file:', err);
      return;
    }
    console.log('File closed successfully');
  });
});
```

The status bar at the bottom shows "Ln 19, Col 1" and other settings.

Synchronous File Opening

Synchronous methods are simpler but can block the event loop, making them less suitable for performance-critical applications.



A screenshot of the Visual Studio Code interface. The title bar shows "Sync_o.js - Mobzway - Visual Studio Code" and the date "Mon Jul 15 12:45". The left sidebar is the Explorer view, showing a folder named "MOBZWAY" containing several JavaScript files: index2.js, index3.js, index4.js, index6.js, index7.js, index8.js, index9.js, index10.mjs, Local.js, Local1.js, package.json, Sync_o.js (which is selected), Sync_w.js, Sync.js, and tempCodeRunnerFil... Below the Explorer are sections for OUTLINE and TIMELINE. The main editor area contains the following code:

```
1  const fs = require('fs');
2
3  try {
4      const fd = fs.openSync('example.txt', 'r');
5      console.log('File opened successfully:', fd);
6
7      // Remember to close the file after using it
8      fs.closeSync(fd);
9      console.log('File closed successfully');
10 } catch (err) {
11     console.error('Error opening file:', err);
12 }
```

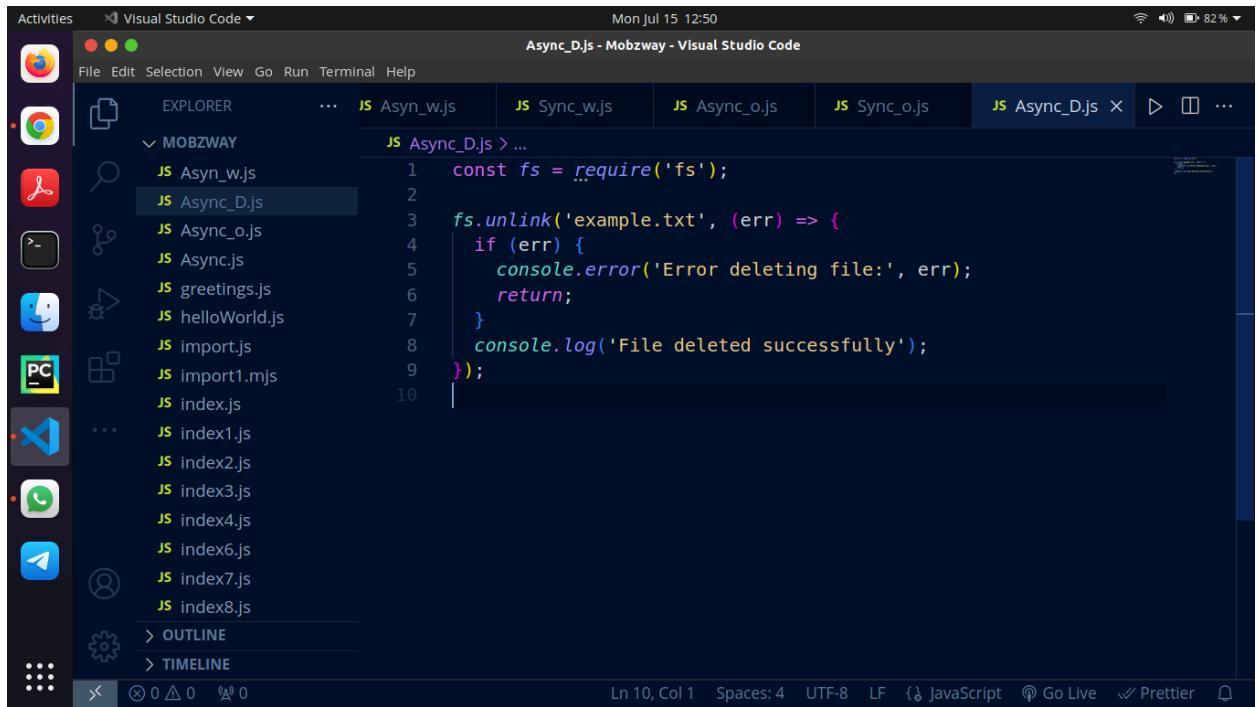
The status bar at the bottom shows "Ln 13, Col 1" and other settings like "Spaces: 4", "UTF-8", "LF", "JavaScript", "Go Live", "Prettier", and a refresh icon.

- **Deleting a File**

Deleting a file in Node.js can be done using the `fs` (File System) module. There are both asynchronous and synchronous methods available for this task.

Asynchronous File Deletion

The asynchronous method is preferred to avoid blocking the event loop.

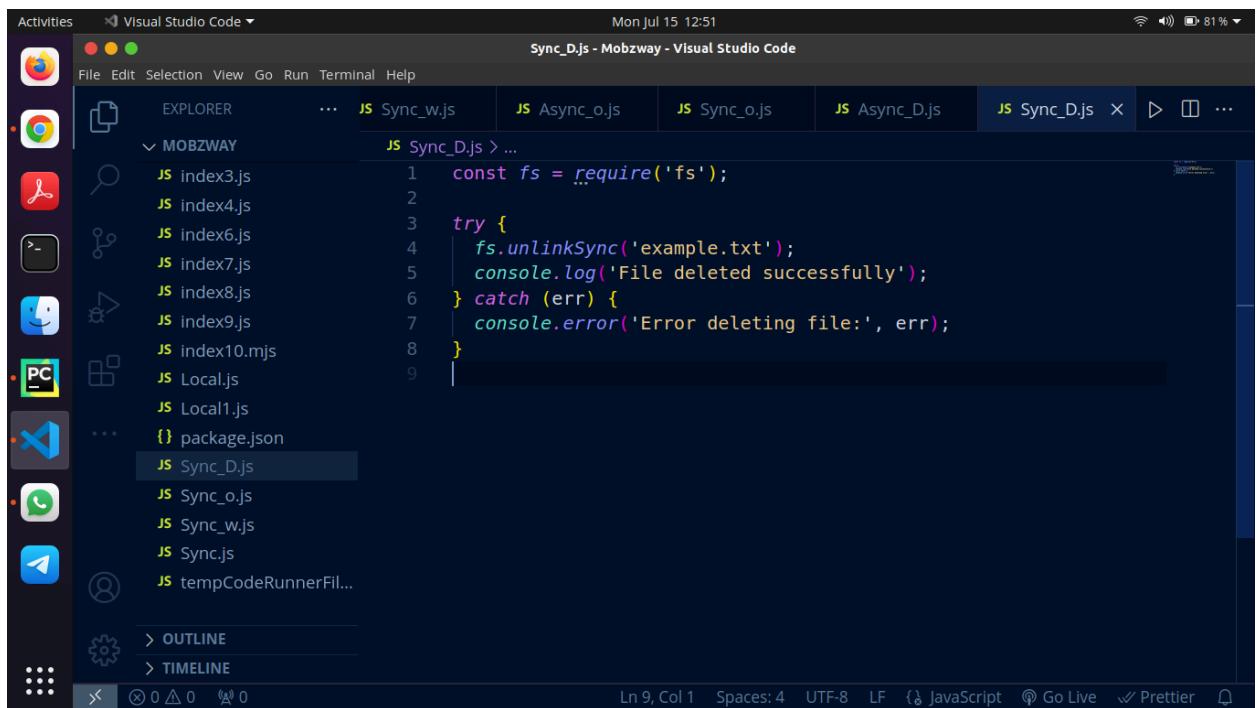


A screenshot of Visual Studio Code showing an asynchronous file deletion script. The file `Async_D.js` is open in the editor, displaying the following code:

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

Synchronous File Deletion

The synchronous method is simpler but can block the event loop.

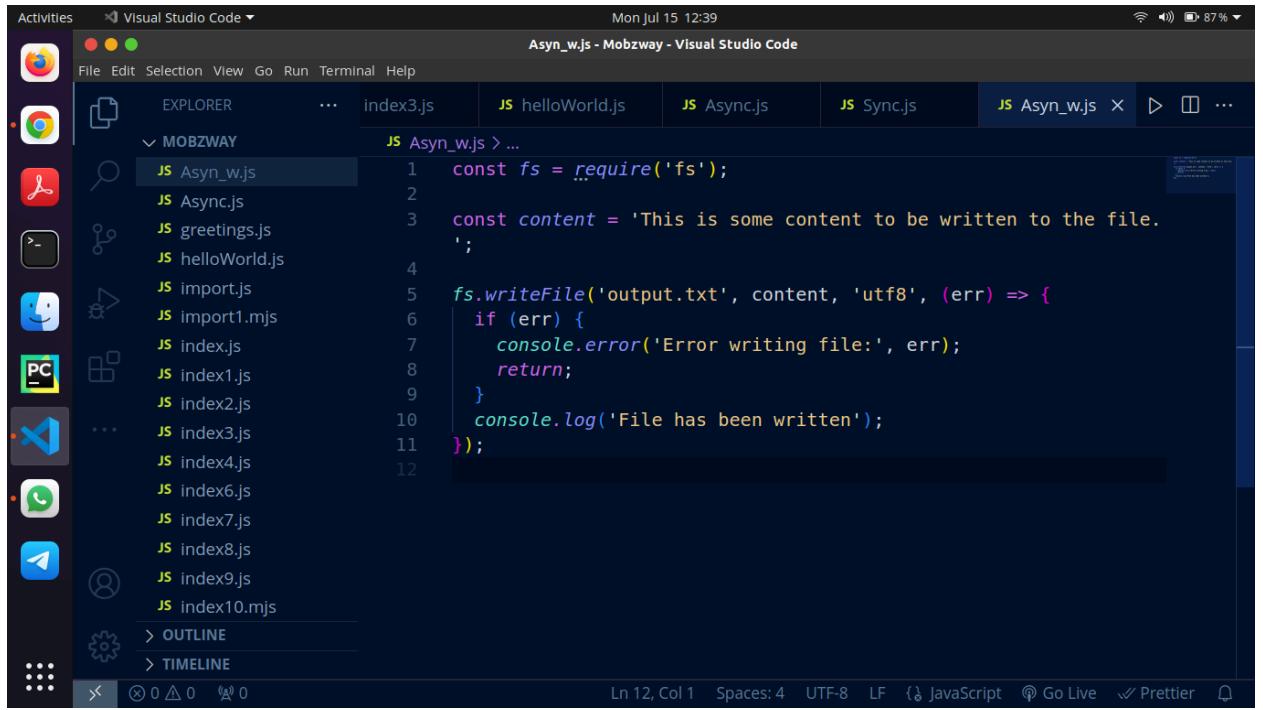


A screenshot of Visual Studio Code showing a synchronous file deletion script. The file `Sync_D.js` is open in the editor, displaying the following code:

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

- **Writing a file asynchronously**

Asynchronous file writing is preferred to avoid blocking the event loop.



The screenshot shows a Visual Studio Code interface with a dark theme. The left sidebar has icons for various applications like a browser, code editor, terminal, and file manager. The main area shows a file tree under 'MOBZWAY' with files like 'index3.js', 'JS Async.js', 'JS Sync.js', and 'JS Asyn_w.js'. The 'JS Asyn_w.js' file is open in the center, displaying the following code:

```
const fs = require('fs');
const content = 'This is some content to be written to the file.
';
fs.writeFile('output.txt', content, 'utf8', (err) => {
  if (err) {
    console.error('Error writing file:', err);
    return;
  }
  console.log('File has been written');
});
```

The status bar at the bottom shows 'Ln 12, Col 1' and other settings like 'Spaces: 4', 'UTF-8', and 'JavaScript'.

In this example:

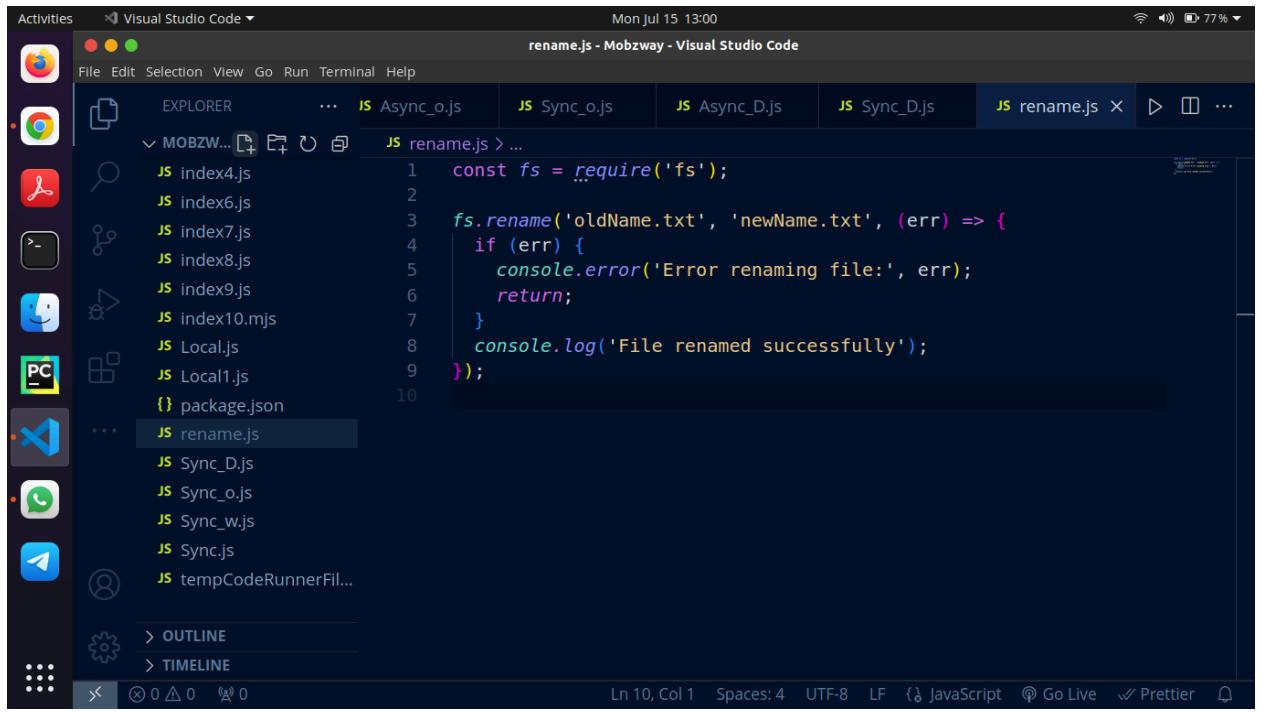
- `fs.writeFile` writes the file asynchronously.
- The first argument is the path to the file.
- The second argument is the content to be written.
- The third argument is the encoding (optional, but `utf8` is commonly used).
- The fourth argument is a callback function that handles any error that occurs.

- **Other I/O Operations**

In Node.js, the `fs` (File System) module provides various I/O operations for working with the file system. Here are some common I/O operations apart from reading, writing, and deleting files:

Renaming a File

You can rename or move a file using the `fs.rename` method.



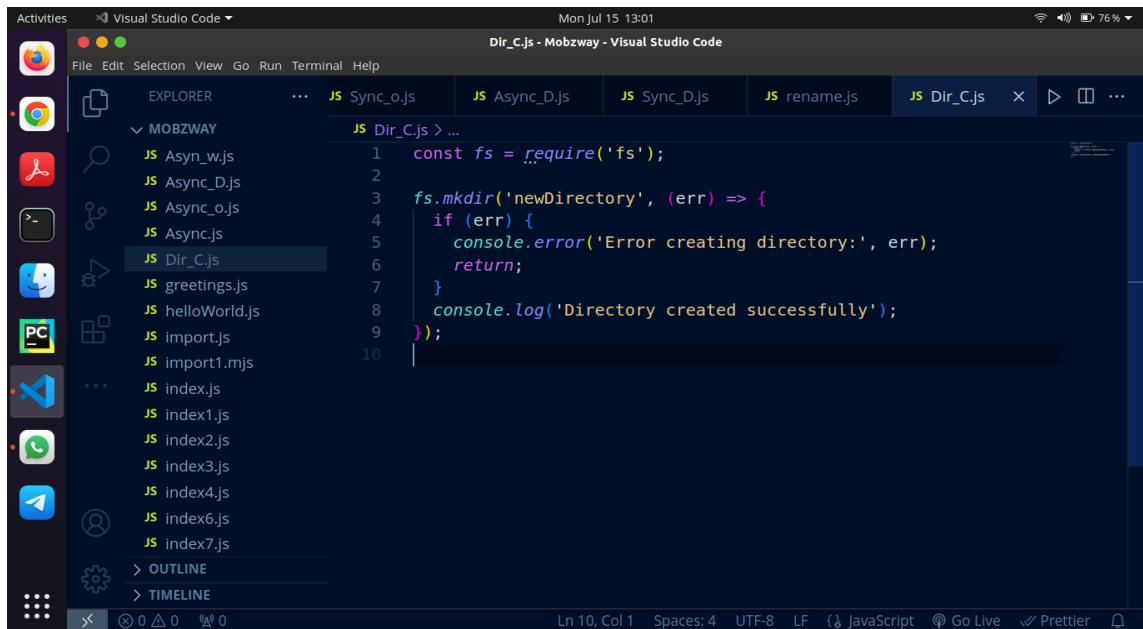
A screenshot of the Visual Studio Code interface. The title bar shows "rename.js - Mobzway - Visual Studio Code". The Explorer sidebar on the left lists files like index4.js, index6.js, index7.js, index8.js, index9.js, index10.mjs, Local.js, Local1.js, package.json, and JS rename.js. The JS rename.js file is selected. The main editor area contains the following code:

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

The status bar at the bottom shows "Ln 10, Col 1 Spaces: 4 UTF-8 LF ⚡ JavaScript ⚡ Go Live ⚡ Prettier".

Creating a Directory

You can create a new directory using the `fs.mkdir` method.



A screenshot of the Visual Studio Code interface. The title bar shows "Dir_C.js - Mobzway - Visual Studio Code". The Explorer sidebar on the left lists files like Asyn_w.js, Async_D.js, Async_o.js, Async.js, JS Dir_C.js, greetings.js, helloWorld.js, import.js, import1.mjs, index.js, index1.js, index2.js, index3.js, index4.js, index6.js, index7.js, and OUTLINE/TIMELINE. The JS Dir_C.js file is selected. The main editor area contains the following code:

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

The status bar at the bottom shows "Ln 10, Col 1 Spaces: 4 UTF-8 LF ⚡ JavaScript ⚡ Go Live ⚡ Prettier".

Module 6: Buffers

- **Why Buffers exist**

Buffers in Node.js are designed to handle binary data directly, allowing you to work with raw memory and perform operations on binary data. Here's a detailed explanation of why buffers exist and their importance:

Reasons for Buffers in Node.js

- 1. **Handling Binary Data:**

- JavaScript, the language in which Node.js is written, primarily handles strings and doesn't have a built-in way to handle binary data. Buffers provide a mechanism to work with binary data directly, enabling operations on raw bytes.

- 2. **Efficient Data Manipulation:**

- Buffers are essential for performance-critical applications. They allow efficient data manipulation without converting data between binary and string formats, which can be costly in terms of performance.

- 3. **Networking:**

- Networking protocols often require dealing with streams of binary data. For instance, when sending or receiving data over TCP or UDP sockets, buffers allow you to manage these streams efficiently.

- 4. **File I/O:**

- When reading from or writing to files, data is often handled in binary form. Buffers enable direct reading and writing of binary data to and from files, making file I/O operations more efficient.

- 5. **Compatibility with C/C++ Libraries:**

- Node.js can interface with C/C++ libraries using the Node-API (formerly N-API). Buffers provide a way to exchange binary data between JavaScript and native modules, ensuring seamless integration and performance.

- 6. **Stream Handling:**

- Buffers are used internally by Node.js streams. They help manage data flow in a stream, handling chunks of binary data, which is crucial for processing large data sets efficiently (e.g., handling large files, streaming media).

- **Creating Buffers**

The screenshot shows a Visual Studio Code interface. The left sidebar has a dark theme with various icons for file types like JSON, JS, and HTML. The main area shows a file named 'Buffer_C.js' with the following code:

```
const buf = Buffer.alloc(10); // Allocates a buffer of 10 bytes
console.log(buf); // <Buffer 00 00 00 00 00 00 00 00 00 00>
```

The terminal tab at the bottom shows the command 'node "/home/anuj/Desktop/Mobzway/Buffer_C.js"' being run, and it outputs the buffer content: '<Buffer 00 00 00 00 00 00 00 00 00 00>'.

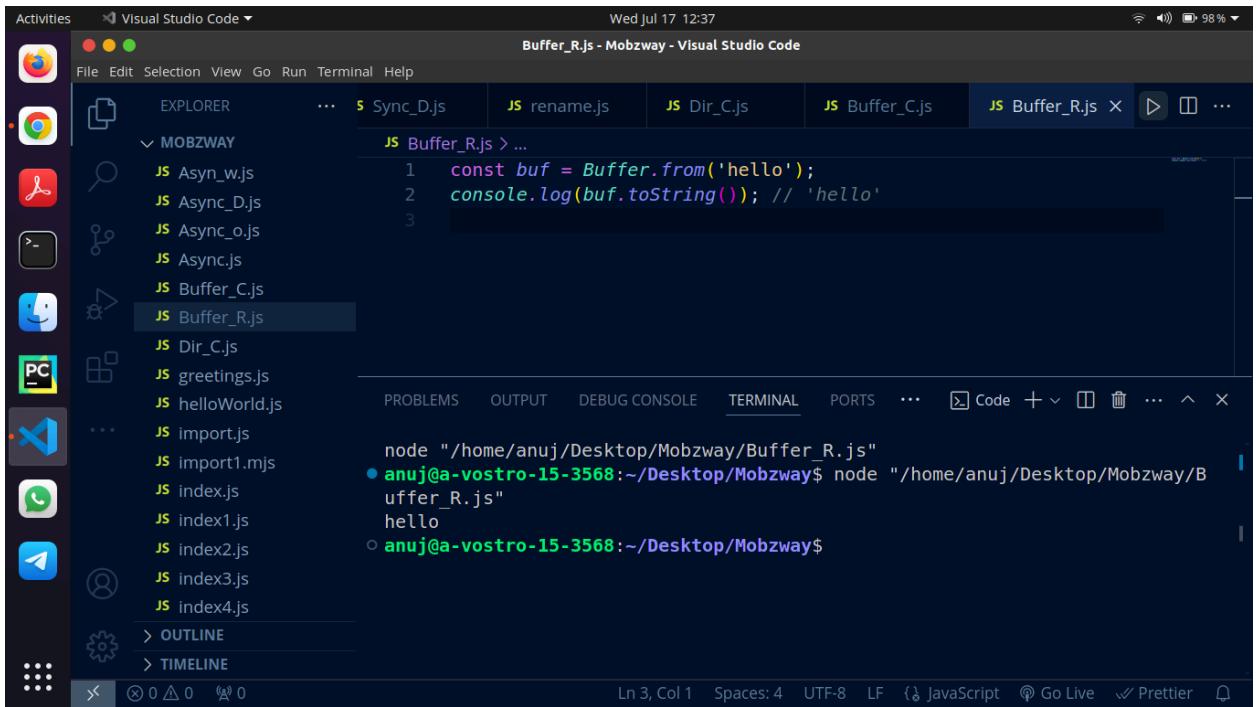
Buffer.alloc(size): This method allocates a new buffer of the specified size in bytes. In this case, `size` is 10, so a buffer of 10 bytes is allocated.

Initialization: The `Buffer.alloc` method initializes all bytes in the buffer to `0`. This means every byte in the buffer will have a value of `0` upon allocation.

- **Reading and Writing Buffers**

Reading from a Buffer

You can read data from a buffer using methods like `toString` or by accessing individual bytes directly.



The screenshot shows the Visual Studio Code interface with the title bar "Buffer_R.js - Mobzway - Visual Studio Code". The status bar at the top right indicates "Wed Jul 17 12:37" and "98%". The left sidebar has a dark theme with various icons for file types like PDF, JSON, and JS. The main workspace shows several files in the Explorer view, including "Sync_D.js", "rename.js", "Dir_C.js", "Buffer_C.js", and "Buffer_R.js". The "Buffer_R.js" file is currently selected. The code editor displays the following JavaScript code:

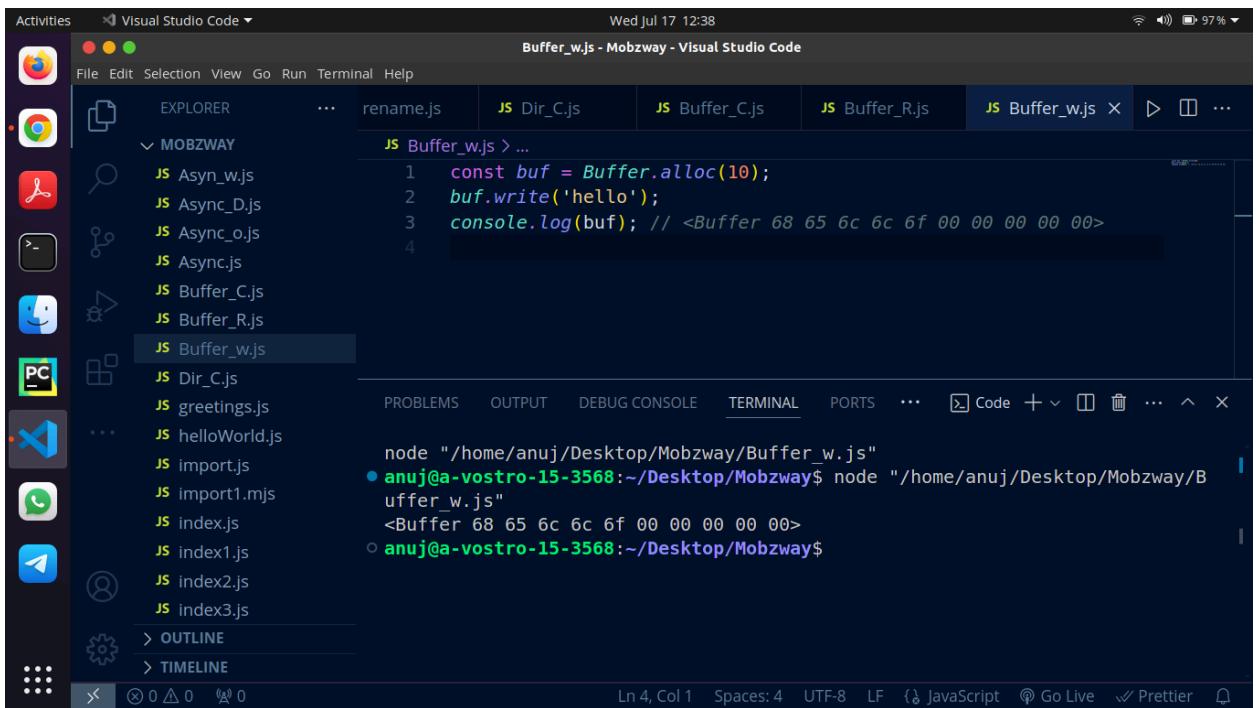
```
const buf = Buffer.from('hello');
console.log(buf.toString()); // 'hello'
```

The terminal below shows the output of running the script:

```
node "/home/anuj/Desktop/Mobzway/Buffer_R.js"
anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/Buffer_R.js"
hello
anuj@a-vostro-15-3568:~/Desktop/Mobzway$
```

Writing to a Buffer

You can write data to a buffer using the `write` method or by directly setting the buffer's values.



The screenshot shows the Visual Studio Code interface with the title bar "Buffer_w.js - Mobzway - Visual Studio Code". The status bar at the top right indicates "Wed Jul 17 12:38" and "97%". The left sidebar has a dark theme with various icons for file types like PDF, JSON, and JS. The main workspace shows several files in the Explorer view, including "rename.js", "Dir_C.js", "Buffer_C.js", "Buffer_R.js", and "Buffer_w.js". The "Buffer_w.js" file is currently selected. The code editor displays the following JavaScript code:

```
const buf = Buffer.alloc(10);
buf.write('hello');
console.log(buf); // <Buffer 68 65 6c 6c 6f 00 00 00 00 00>
```

The terminal below shows the output of running the script:

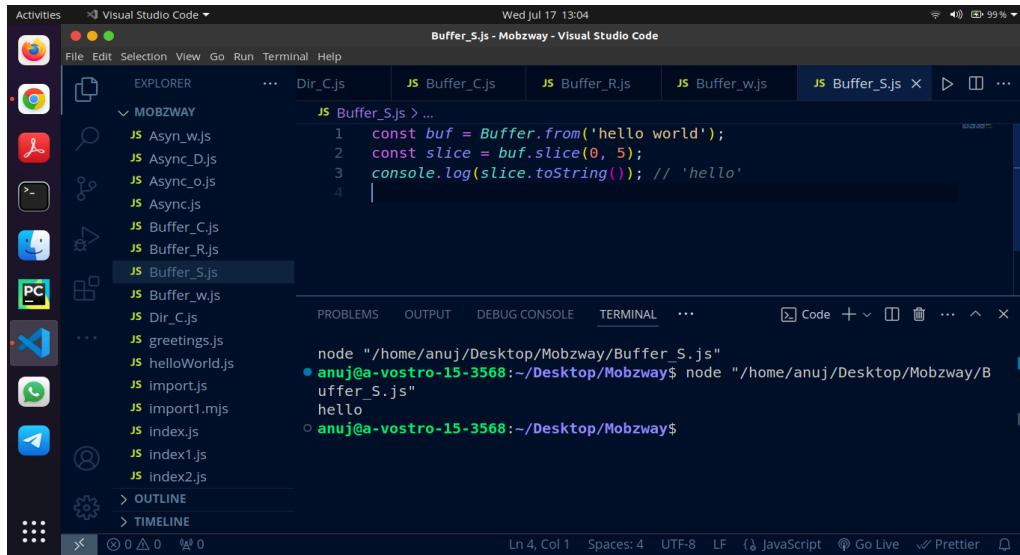
```
node "/home/anuj/Desktop/Mobzway/Buffer_w.js"
anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/Buffer_w.js"
<Buffer 68 65 6c 6c 6f 00 00 00 00 00>
anuj@a-vostro-15-3568:~/Desktop/Mobzway$
```

- **Manipulating Buffers**

Manipulating buffer data in Node.js involves performing operations such as reading, writing, slicing, copying, and filling buffers.

Slicing a Buffer

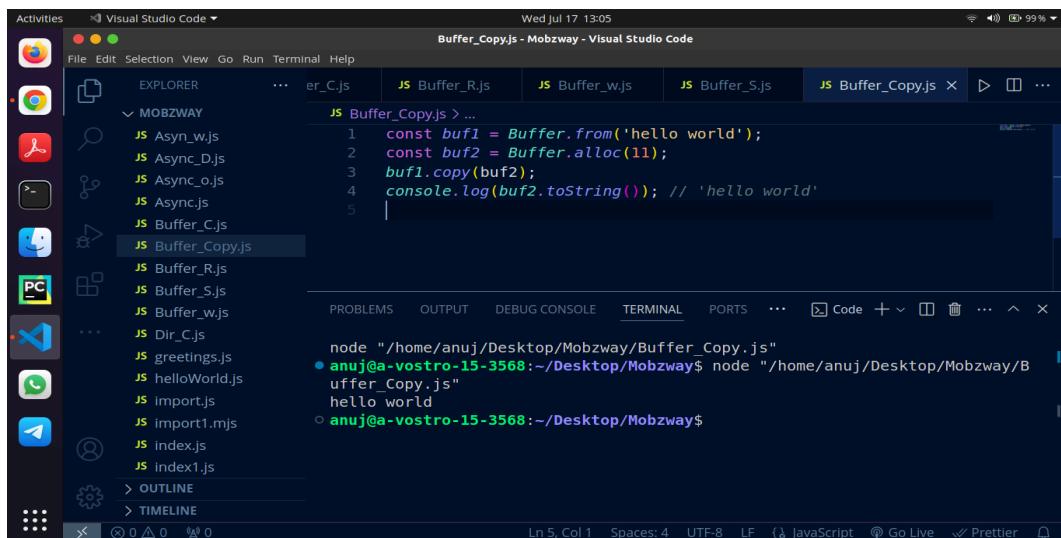
You can create a new buffer that references a subset of the original buffer's memory using the `slice` method:



```
const buf = Buffer.from('hello world');
const slice = buf.slice(0, 5);
console.log(slice.toString()); // 'hello'
```

Copying a Buffer

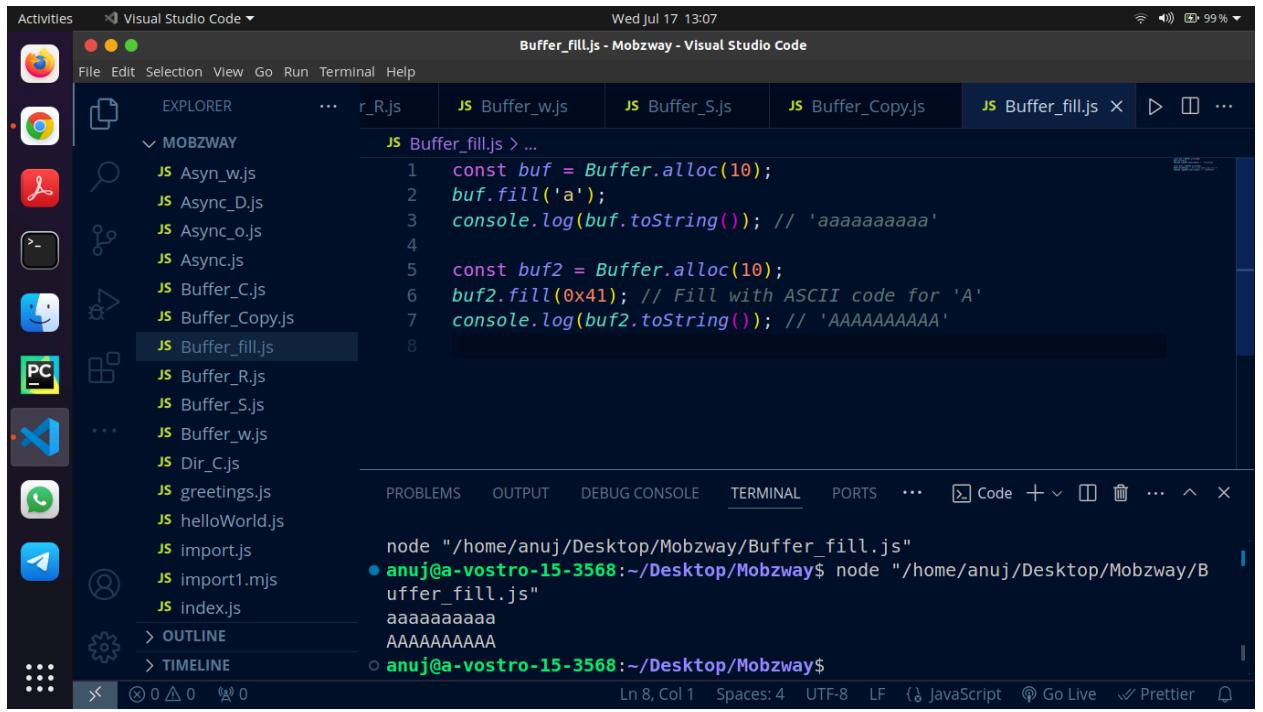
You can copy data from one buffer to another using the `copy` method:



```
const buf1 = Buffer.from('hello world');
const buf2 = Buffer.alloc(11);
buf1.copy(buf2);
console.log(buf2.toString()); // 'hello world'
```

Filling a Buffer

You can fill a buffer with a specific value using the `fill` method:



The screenshot shows a Visual Studio Code interface. The left sidebar has a 'MOBZWAY' folder containing several JavaScript files like 'Asyn_w.js', 'Buffer_C.js', etc. The main editor area shows a file named 'Buffer_fill.js' with the following code:

```
1 const buf = Buffer.alloc(10);
2 buf.fill('a');
3 console.log(buf.toString()); // 'aaaaaaaaaa'
4
5 const buf2 = Buffer.alloc(10);
6 buf2.fill(0x41); // Fill with ASCII code for 'A'
7 console.log(buf2.toString()); // 'AAAAAAAAAA'
```

The terminal at the bottom shows the output of running the script:

```
node "/home/anuj/Desktop/Mobzway/Buffer_fill.js"
anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/Buffer_fill.js"
aaaaaaaaaa
AAAAAAAAAA
anuj@a-vostro-15-3568:~/Desktop/Mobzway$
```

Module 7: Streams

• What are Streams

Streams are a fundamental concept in Node.js for handling I/O operations efficiently. They provide a way to read and write data in a continuous, sequential manner, which is particularly useful for working with large data sets, such as files,

network requests, and real-time communication. Streams are an instance of the `EventEmitter` class and allow you to process data piece by piece (chunks), rather than loading the entire data into memory at once.

Types of Streams

Node.js provides four main types of streams:

1. **Readable Streams:**
 - o Used for reading data.
 - o Examples: `fs.createReadStream`, `http.IncomingMessage` (for HTTP requests).
2. **Writable Streams:**
 - o Used for writing data.
 - o Examples: `fs.createWriteStream`, `http.ServerResponse` (for HTTP responses).
3. **Duplex Streams:**
 - o Both readable and writable.
 - o Examples: `net.Socket`, `zlib.createDeflate`.
4. **Transform Streams:**
 - o A type of duplex stream where the output is computed based on the input.
 - o Examples: `zlib.createGzip`, `crypto.createCipher`.

• **Read and Write Stream API**

The Read and Write Stream API in Node.js provides a way to handle reading from and writing to streams in a more efficient manner, especially for large amounts of data. Here's a comprehensive guide on how to use the Read and Write Stream API with examples.

Readable Streams

Readable streams are used to read data from a source in a sequential manner.

The screenshot shows the Visual Studio Code interface with the title bar "Stream_R.js - Mobzway - Visual Studio Code". The status bar at the bottom indicates "Wed Jul 24 16:24" and "38%". The Explorer sidebar on the left shows a folder named "MOBZWAY" containing various JavaScript files like index6.js, index7.js, etc., and some configuration files. The main editor area displays the following code:

```
const fs = require('fs');
// Create a readable stream
const readableStream = fs.createReadStream('example.txt', {
  encoding: 'utf8'
});

readableStream.on('data', (chunk) => {
  console.log(`Received ${chunk.length} bytes of data.`);
  console.log(chunk);
});

readableStream.on('end', () => {
  console.log('No more data.');
});

readableStream.on('error', (err) => {
  console.error('An error occurred:', err);
});
```

The status bar at the bottom right shows "Ln 18, Col 1" and other settings.

Writable Streams

Writable streams are used to write data to a destination sequentially

The screenshot shows the Visual Studio Code interface with the title bar "Stream_W.js - Mobzway - Visual Studio Code". The status bar at the bottom indicates "Wed Jul 24 16:28" and "42%". The Explorer sidebar on the left shows a folder named "MOBZWAY" containing various JavaScript files like index7.js, index8.js, etc., and some configuration files. The main editor area displays the following code:

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

// Write data to the stream
writableStream.write('Hello, ');
writableStream.write('world!\n');

// End the writable stream
writableStream.end('This is the end.\n');

writableStream.on('finish', () => {
  console.log('All writes are now complete.');
});

writableStream.on('error', (err) => {
  console.error('An error occurred:', err);
});
```

The status bar at the bottom right shows "Ln 20, Col 1" and other settings.

● Flow Control

Flow control in Node.js streams is essential for managing the rate at which data is read from a source and written to a destination, ensuring that neither the readable stream nor the writable stream becomes overwhelmed. This is especially important when dealing with large data sets or when the speed of the source and destination are mismatched.

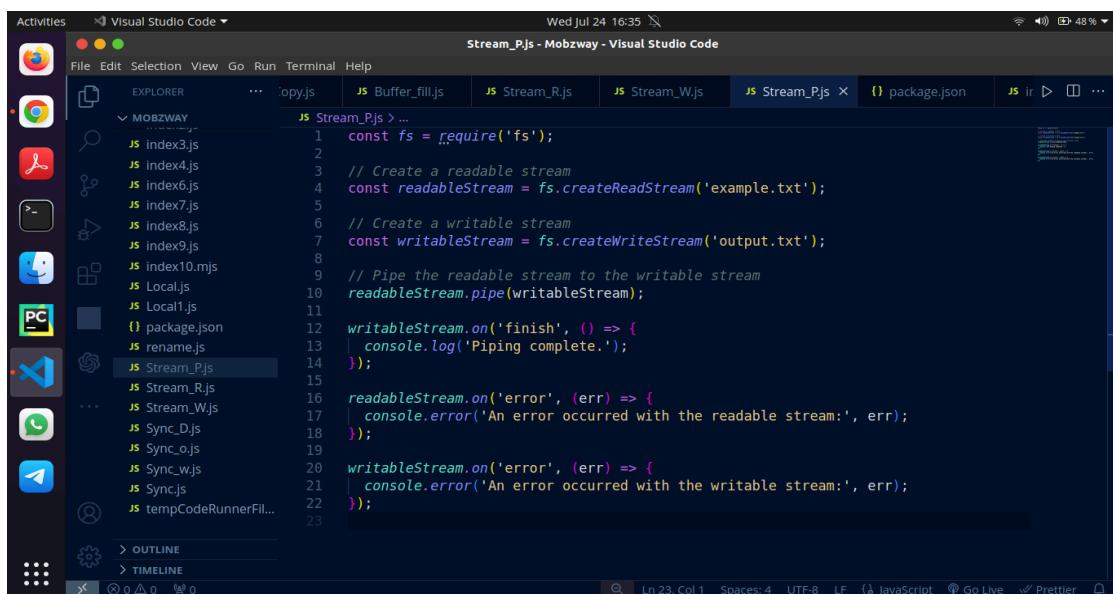
Understanding Flow Control

When working with streams, flow control mechanisms help balance the flow of data. Key aspects include:

1. **Readable Stream:** Emits data as it is read from the source.
2. **Writable Stream:** Writes data to the destination at its own pace.
3. **Backpressure:** Occurs when the writable stream can't handle the rate at which the readable stream is providing data.

● Piping

Piping streams is a powerful feature that allows you to connect a readable stream to a writable stream. This automatically handles the flow of data from the source to the destination.



The screenshot shows a Visual Studio Code interface with the title bar "Stream_Pjs - Mobzway - Visual Studio Code". The status bar indicates "Wed Jul 24 16:35" and "48%". The left sidebar shows a project structure for "MOBZWAY" with files like index3.js, index4.js, index6.js, index7.js, index8.js, index9.js, index10.mjs, Local.js, Local1.js, package.json, rename.js, Stream_Pjs, Stream_R.js, Stream_W.js, Sync_D.js, Sync_o.js, Sync_w.js, Sync.js, and tempCodeRunnerFile... The main editor tab is "Stream_Pjs" containing the following code:

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

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

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

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

writableStream.on('finish', () => {
  console.log('Piping complete.');
});

readableStream.on('error', (err) => {
  console.error('An error occurred with the readable stream:', err);
});

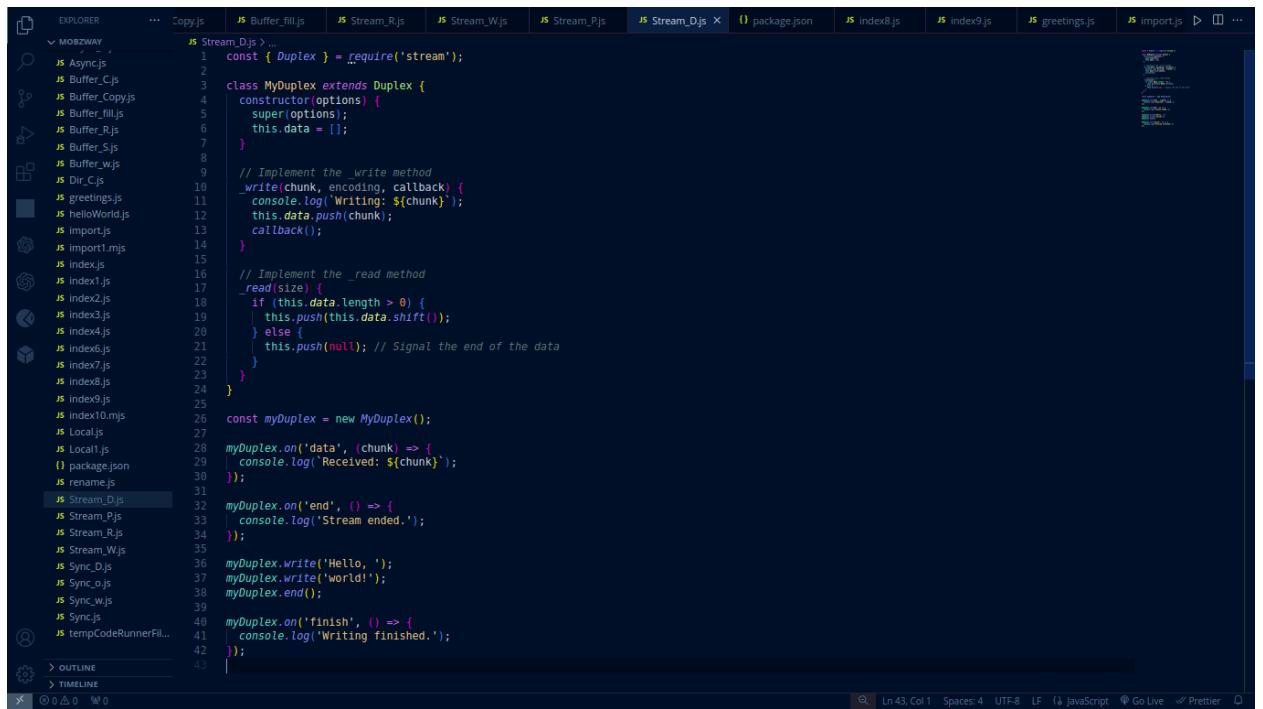
writableStream.on('error', (err) => {
  console.error('An error occurred with the writable stream:', err);
});
```

- **Duplex Stream**

A duplex stream in Node.js is a type of stream that implements both the Readable and Writable interfaces. This means that it can read and write data, making it suitable for use cases where you need to perform both operations, such as in network protocols, file handling, or transformation streams.

Creating a Duplex Stream

To create a duplex stream, you can extend the `stream.Duplex` class and implement the `_read` and `_write` methods. Here is an example of how to create a custom duplex stream.



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

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

  // Implement the _write method
  _write(chunk, encoding, callback) {
    console.log(`Writing: ${chunk}`);
    this.data.push(chunk);
    callback();
  }

  // Implement the _read method
  _read(size) {
    if (this.data.length > 0) {
      this.push(this.data.shift());
    } else {
      this.push(null); // Signal the end of the data
    }
  }
}

const myDuplex = new MyDuplex();
myDuplex.on('data', (chunk) => {
  console.log(`Received: ${chunk}`);
});

myDuplex.on('end', () => {
  console.log(`Stream ended.`);
});

myDuplex.write('Hello, ');
myDuplex.write('world!');
myDuplex.end();

myDuplex.on('finish', () => {
  console.log(`Writing finished.`);
});
```

- **Transform Stream**

A transform stream is a type of duplex stream where the output is computed based on the input. This allows you to transform data as it is being read and

written. Node.js provides a `Transform` class in the `stream` module for creating such streams.

Here's a simple example of a transform stream that converts all input text to uppercase.

```
EXPLORER          Copy.js   JS Buffer_fill.js  JS Stream_R.js  JS Stream_W.js  JS Stream_P.js  JS Stream_D.js  JS Stream_T.js  package.json  JS index8.js  JS index9.js  JS greetings.js
MOBZWAY
JS Buffer_C.js
JS Buffer_Copy.js
JS Buffer_fill.js
JS Buffer_R.js
JS Buffer_S.js
JS Buffer_w.js
JS Dir_C.js
JS greetings.js
JS helloWorld.js
JS import.js
JS import1.mjs
JS index.js
JS index1.js
JS index2.js
JS index3.js
JS index4.js
JS index5.js
JS index6.js
JS index7.js
JS index8.js
JS index9.js
JS index10.mjs
JS Local.js
JS Local1.js
() package.json
JS rename.js
JS Stream_D.js
JS Stream_R.js
JS Stream_T.js
JS Stream_W.js
JS Sync_D.js
JS Sync_C.js
JS Sync_w.js
JS Sync.js
JS tempCodeRunnerFill...
```

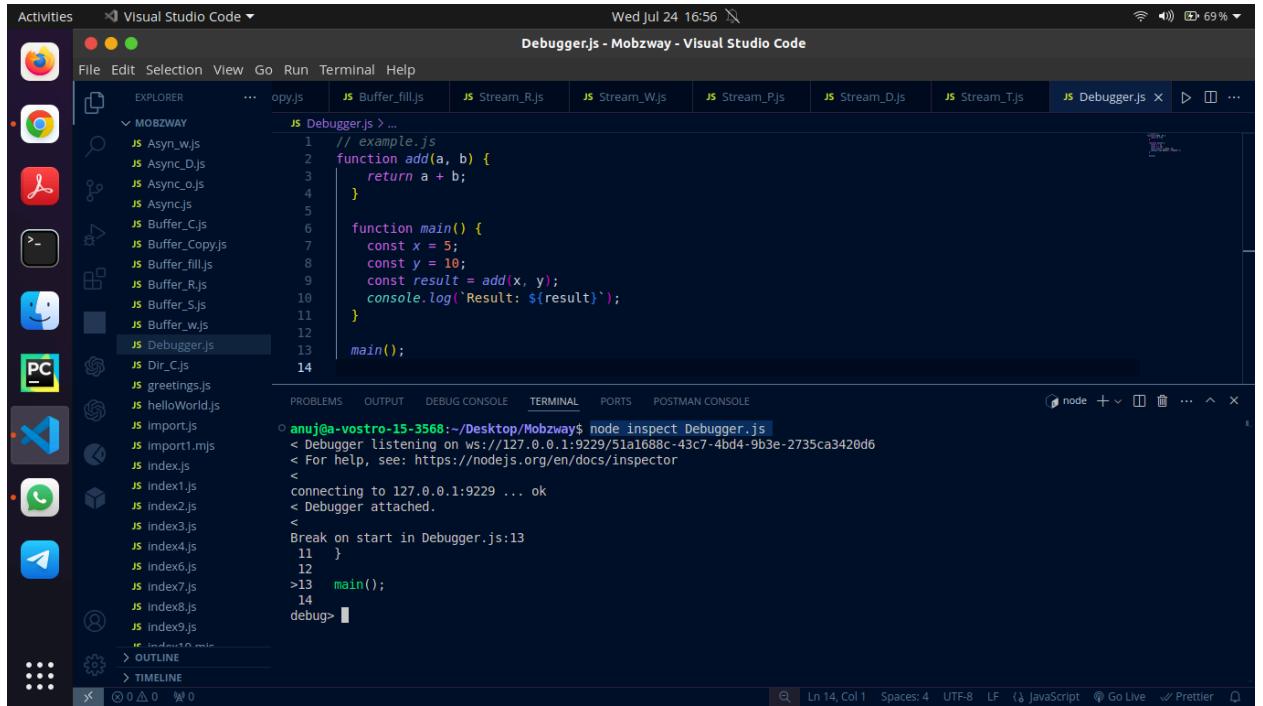
```
node "/home/anuj/Desktop/Mobzway/Stream_T.js"
• anuj@anuj-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/Stream_T.js"
Transformed: HELLO,
Transformed: WORLD!
Transform stream finished.
o anuj@anuj-vostro-15-3568:~/Desktop/Mobzway$
```

Module 8: Debugging Node JS Application

- Core Node JS Debugger

The core Node.js debugger allows you to debug your Node.js applications directly from the command line. This is useful for inspecting your code, setting

breakpoints, and stepping through your code to understand its behavior. The debugger can be accessed using the built-in `inspect` command.



The screenshot shows the Visual Studio Code interface. The left sidebar displays a file tree with several JavaScript files under a 'MOBZWAY' folder. The main editor area shows the content of 'Debugger.js'. The terminal at the bottom has the following output:

```
anuj@anuj-Vostro-15-3568:~/Desktop/Mobzway$ node inspect Debugger.js
< Debugger listening on ws://127.0.0.1:9229/51a1688c-43c7-4bd4-9b3e-2735ca3420d6
< For help, see: https://nodejs.org/en/docs/inspector
<
< connecting to 127.0.0.1:9229 ... ok
< Debugger attached.
<
Break on start in Debugger.js:13
  11 }
  12
>13 main();
  14 debug>
```

1. Start the Debugger: Run your script with the `inspect` flag.

```
node inspect example.js
```

2. Running and Stepping Through Code:

- `cont` or `c`: Continue execution until the next breakpoint.
- `next` or `n`: Step to the next line of code.
- `step` or `s`: Step into a function call.
- `out` or `o`: Step out of the current function.
- `repl`: Open a REPL session to inspect variables and execute code.

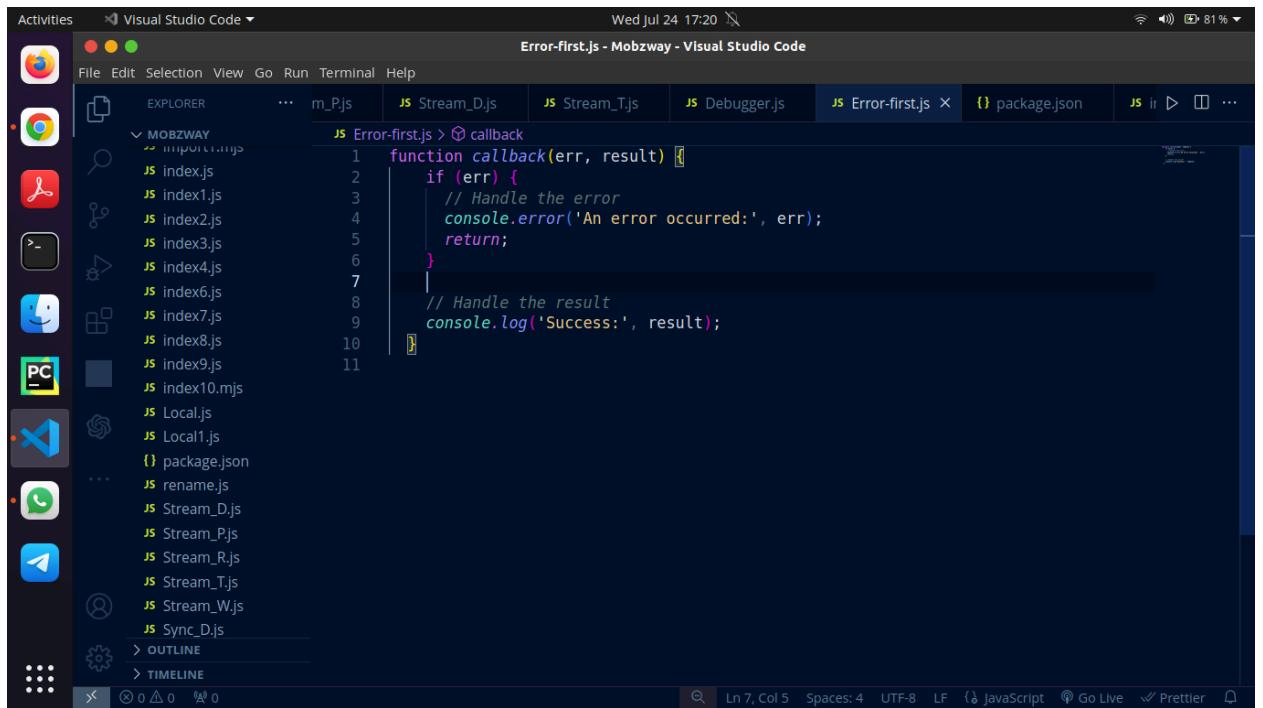
3. Setting Breakpoints:

- `setBreakpoint()` or `sb()`: Set a breakpoint at a specific line.
- `clearBreakpoint()` or `cb()`: Clear a breakpoint.

- **Callbacks: Error-first**

Error-first callbacks are a common pattern in Node.js and JavaScript programming, especially when dealing with asynchronous operations. This pattern ensures that the first argument of the callback function is reserved for an error object (if any error occurs), and subsequent arguments are used for successful results. This approach helps to consistently handle errors and makes the code more predictable and easier to maintain.

Structure of an Error-first Callback



The screenshot shows a Visual Studio Code interface with the following details:

- Title Bar:** Activities, Visual Studio Code, Wed Jul 24 17:20, Error-first.js - Mobzway - Visual Studio Code
- File Menu:** File, Edit, Selection, View, Go, Run, Terminal, Help
- Toolbar:** Includes icons for browser, file, search, and settings.
- Explorer:** Shows a project structure under "MOBZWAY":
 - m_Pjs
 - JS Stream_D.js
 - JS Stream_T.js
 - JS Debugger.js
 - JS Error-first.js (highlighted)
 - {} package.json
 - JS ir
- Editor:** Displays the content of "Error-first.js".

```
function callback(err, result) {
  if (err) {
    // Handle the error
    console.error('An error occurred:', err);
    return;
  }
  // Handle the result
  console.log('Success:', result);
}
```
- Bottom Status Bar:** Ln 7, Col 5, Spaces: 4, UTF-8, LF, {JavaScript}, Go Live, Prettier

