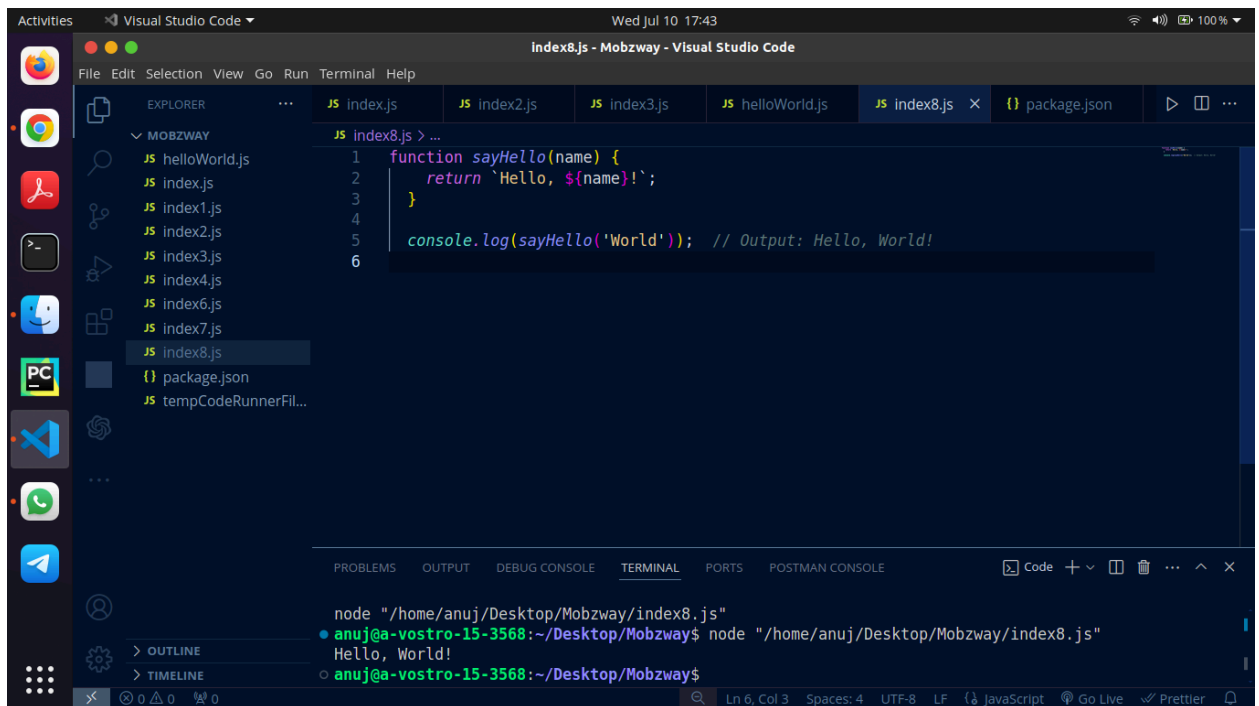


Node.js Documentation

Module 3: Node JS Modules

- **Functions**

Named Functions



The screenshot shows the Visual Studio Code editor interface. The Explorer sidebar on the left displays a project named 'MOBZWAY' with several JavaScript files. The main editor window is open to 'index8.js', which contains the following code:

```
1 function sayHello(name) {  
2   return `Hello, ${name}!`;  
3 }  
4  
5 console.log(sayHello('World')); // Output: Hello, World!  
6
```

The bottom panel shows the TERMINAL output, indicating the command executed and the result:

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

Anonymous Functions

Visual Studio Code interface showing a JavaScript file named `index8.js` in the `MOBZWAY` project. The code defines a function `sayHello` and logs its output.

```
1 const sayHello = function(name) {  
2   return `Hello, ${name}!`;  
3 };  
4  
5 console.log(sayHello('World')); // Output: Hello, World!  
6
```

The terminal output shows the command `node "/home/anj/Desktop/Mobzway/index8.js"` being executed, resulting in the output `Hello, World!`.

Arrow Functions

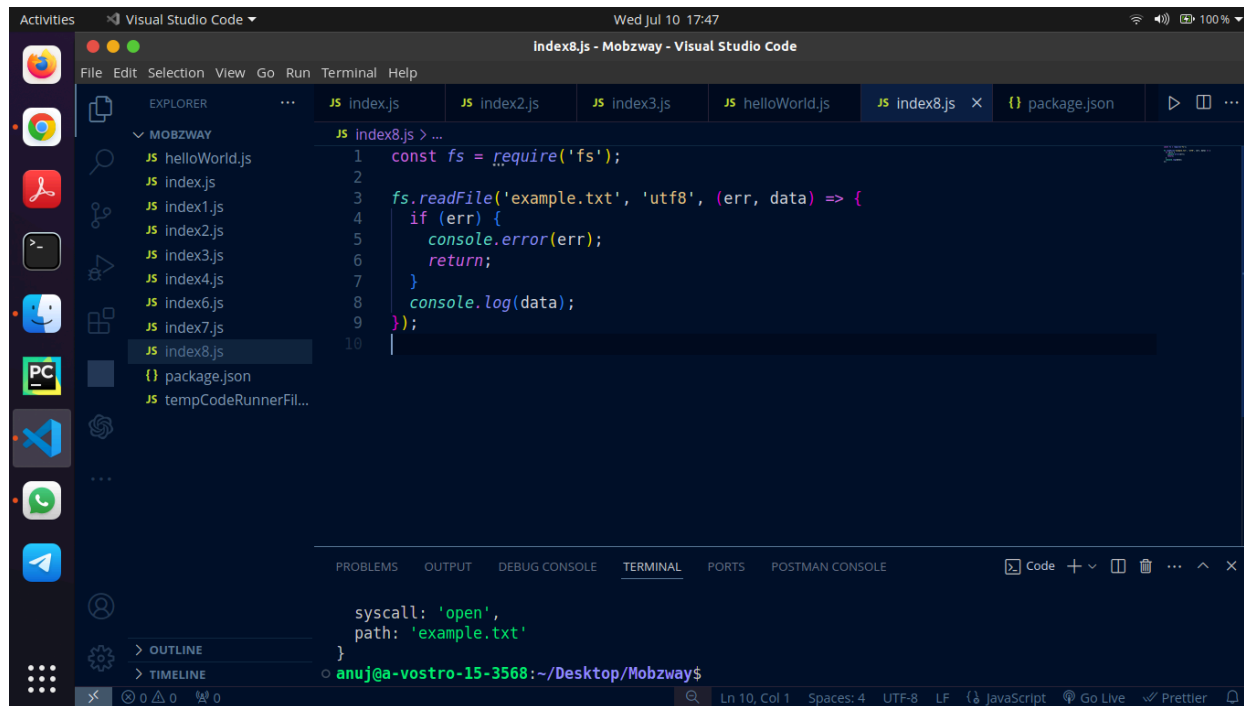
Visual Studio Code interface showing a JavaScript file named `index8.js` in the `MOBZWAY` project. The code defines an arrow function `sayHello` and logs its output.

```
1 const sayHello = (name) => {  
2   return `Hello, ${name}!`;  
3 };  
4  
5 console.log(sayHello('World')); // Output: Hello, World!  
6
```

The terminal output shows the command `node "/home/anj/Desktop/Mobzway/index8.js"` being executed, resulting in the output `Hello, World!`.

Asynchronous Functions

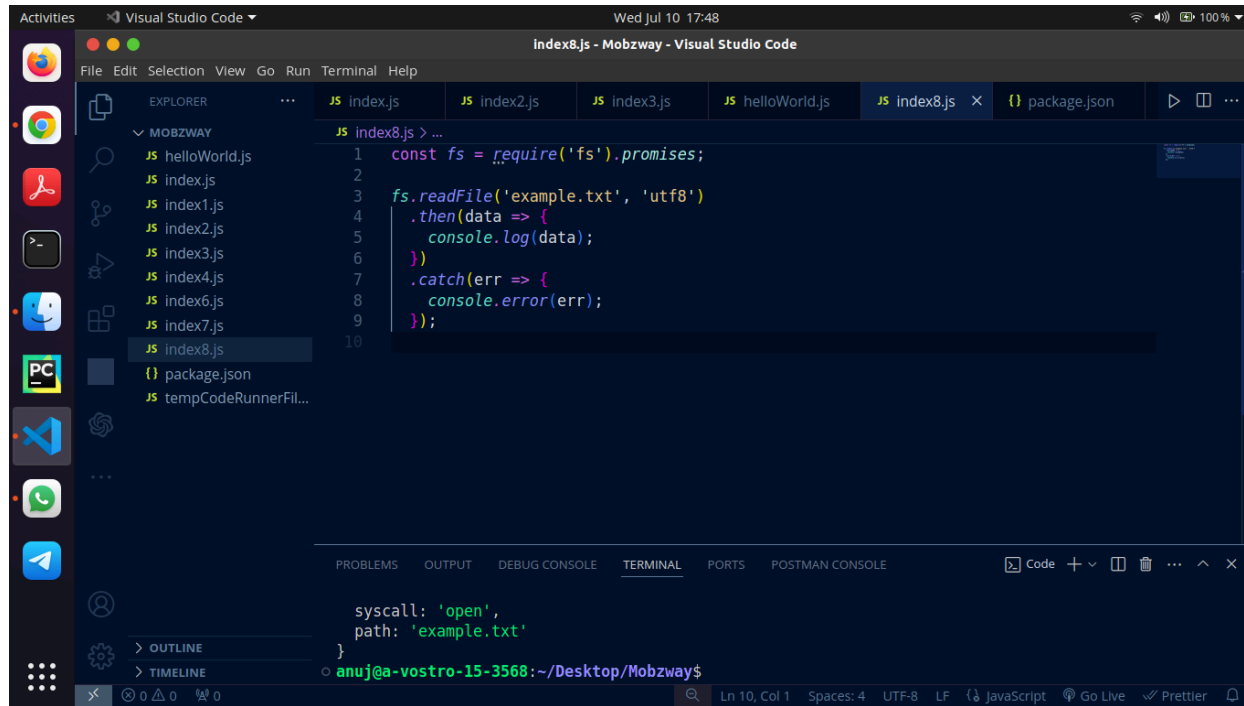
1. Callbacks



The screenshot shows the Visual Studio Code editor with a file named `index8.js` open. The code uses the `fs` module with a callback function to read a file. The Explorer sidebar on the left shows a project named `MOBZWAY` with several `index` files and a `package.json` file. The Terminal at the bottom shows the command `syscall: 'open', path: 'example.txt'` and the user prompt `anuj@a-vostro-15-3568:~/Desktop/Mobzway$`.

```
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
```

2. Promises

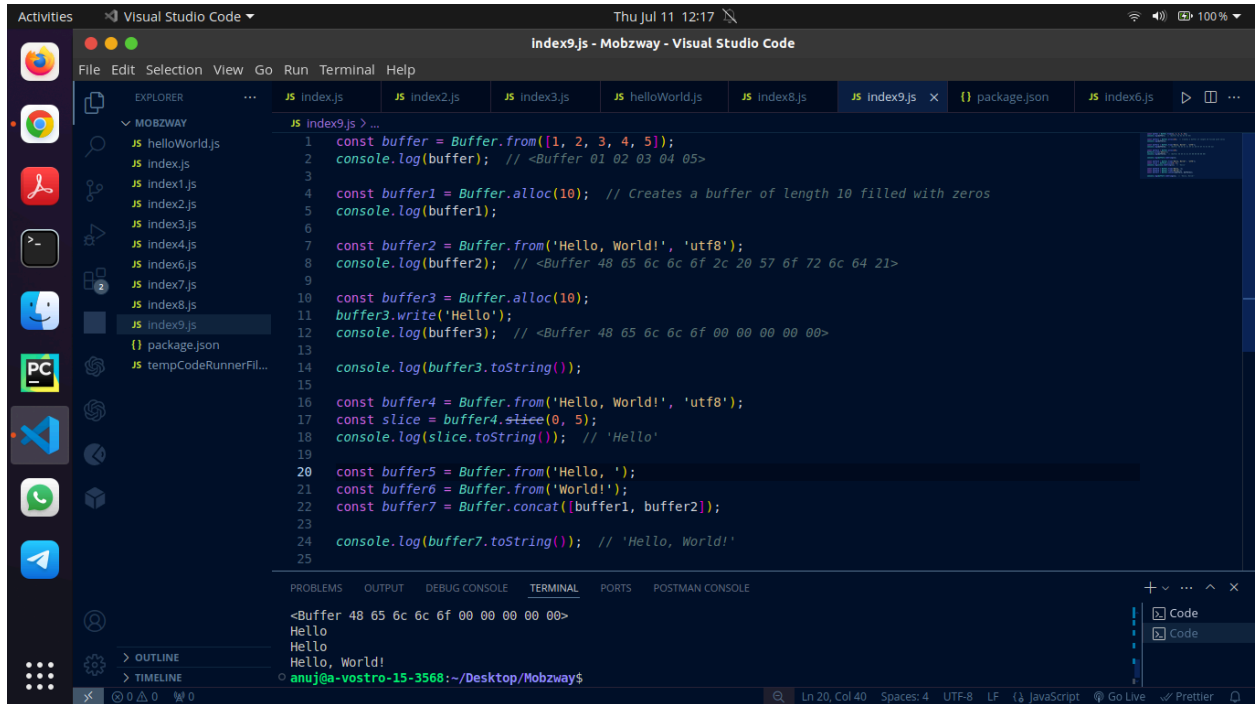


The screenshot shows the Visual Studio Code editor with a file named `index8.js` open. The code uses the `fs` module with Promises to read a file. The Explorer sidebar on the left shows the same project structure as the first screenshot. The Terminal at the bottom shows the same command `syscall: 'open', path: 'example.txt'` and the user prompt `anuj@a-vostro-15-3568:~/Desktop/Mobzway$`.

```
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
```

- **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.



```
1 const buffer = Buffer.from([1, 2, 3, 4, 5]);
2 console.log(buffer); // <Buffer 01 02 03 04 05>
3
4 const buffer1 = Buffer.alloc(10); // Creates a buffer of length 10 filled with zeros
5 console.log(buffer1);
6
7 const buffer2 = Buffer.from('Hello, World!', 'utf8');
8 console.log(buffer2); // <Buffer 48 65 6c 6c 6f 2c 20 57 6f 72 6c 64 21>
9
10 const buffer3 = Buffer.alloc(10);
11 buffer3.write('Hello');
12 console.log(buffer3); // <Buffer 48 65 6c 6c 6f 00 00 00 00 00>
13
14 console.log(buffer3.toString());
15
16 const buffer4 = Buffer.from('Hello, World!', 'utf8');
17 const slice = buffer4.slice(0, 5);
18 console.log(slice.toString()); // 'Hello'
19
20 const buffer5 = Buffer.from('Hello, ');
21 const buffer6 = Buffer.from('World!');
22 const buffer7 = Buffer.concat([buffer1, buffer2]);
23
24 console.log(buffer7.toString()); // 'Hello, World!'
25
```

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

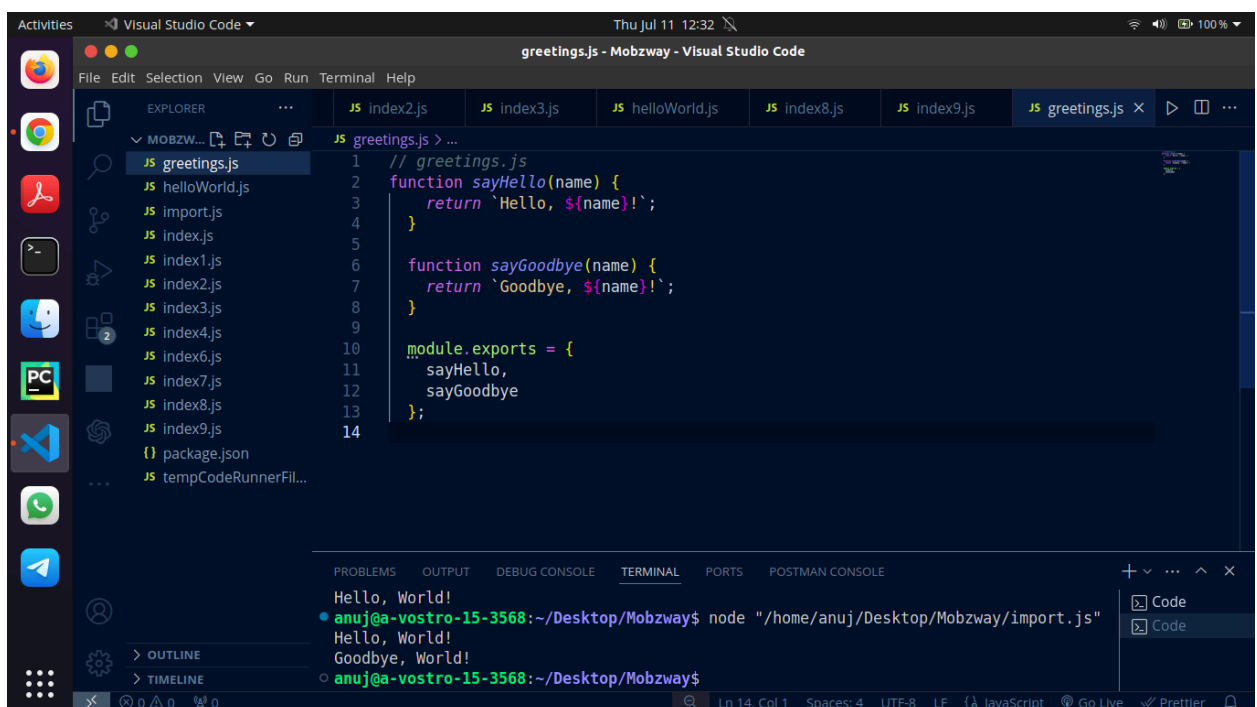
You can concatenate multiple buffers into a single buffer.

- **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`.



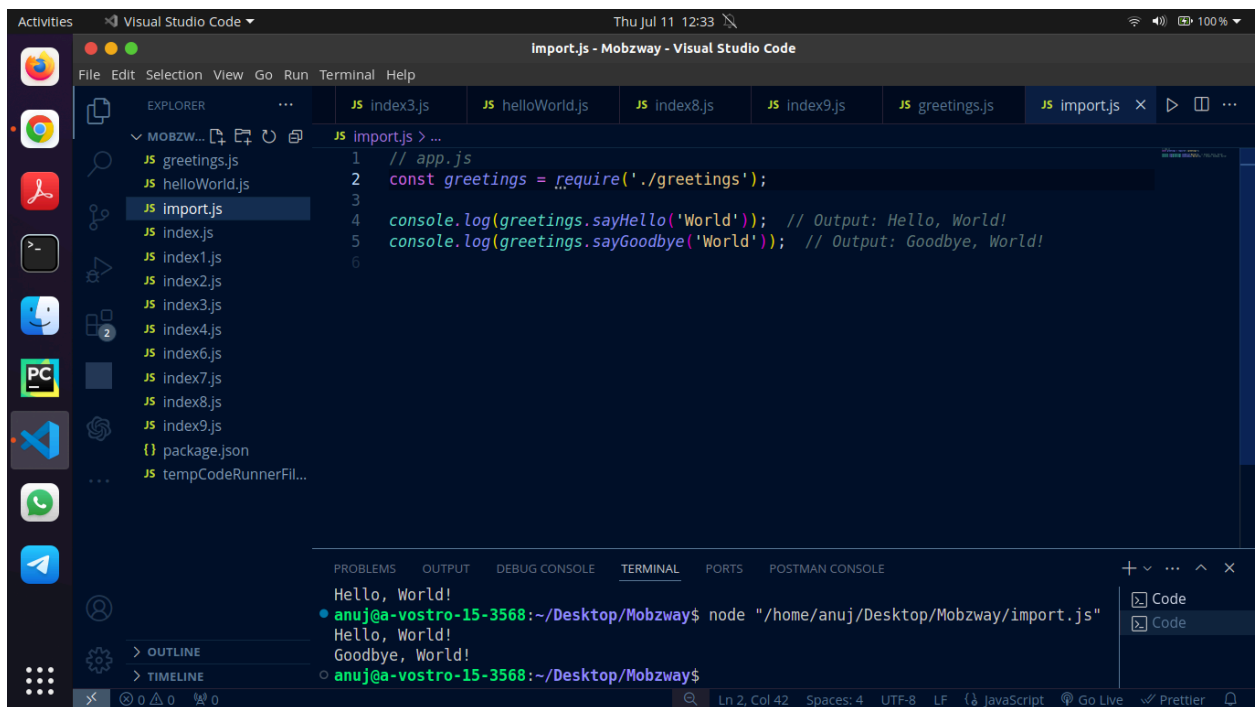
```
greetings.js - Mobzway - Visual Studio Code

1 // greetings.js
2 function sayHello(name) {
3   return `Hello, ${name}!`;
4 }
5
6 function sayGoodbye(name) {
7   return `Goodbye, ${name}!`;
8 }
9
10 module.exports = {
11   sayHello,
12   sayGoodbye
13 };
14
```

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

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 editor with a file named 'import.js' open. The code in the editor is as follows:

```
1 // app.js
2 const greetings = require('./greetings');
3
4 console.log(greetings.sayHello('World')); // Output: Hello, World!
5 console.log(greetings.sayGoodbye('World')); // Output: Goodbye, World!
6
```

The Explorer sidebar on the left shows a project named 'MOBZW...' with several JavaScript files: 'greetings.js', 'helloWorld.js', 'import.js' (selected), 'index.js', 'index1.js', 'index2.js', 'index3.js', 'index4.js', 'index6.js', 'index7.js', 'index8.js', 'index9.js', 'package.json', and 'tempCodeRunnerFil...'. The Terminal at the bottom shows the command 'node "/home/anj/Desktop/Mobzway/import.js"' being executed, resulting in the output: 'Hello, World!', '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.

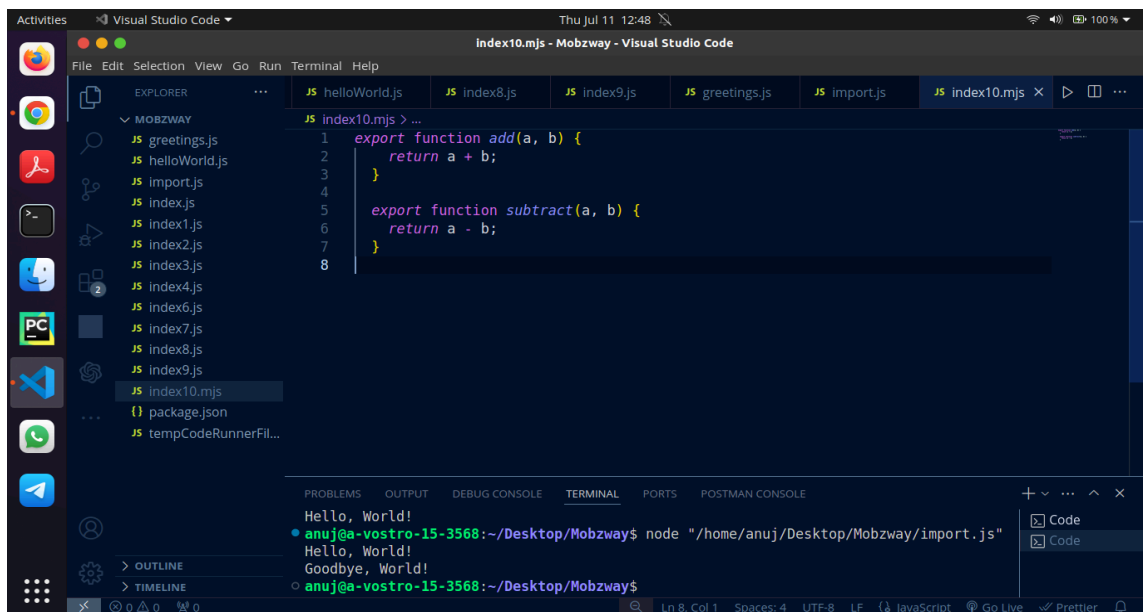
Examples of Popular Third-party Modules:

- **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**.

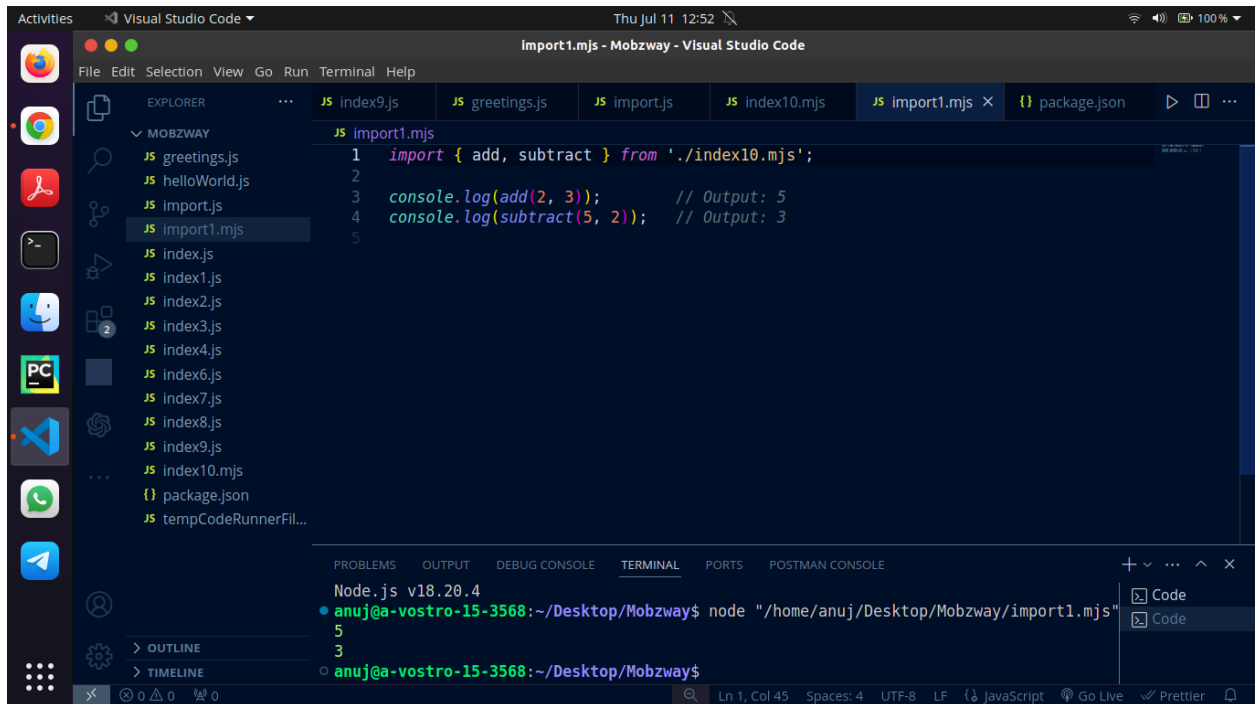
An MJS file is a source code file containing an ES Module (ECMAScript Module) for use with a Node.js application.



The screenshot shows the Visual Studio Code editor with a project named 'Mobzway'. The Explorer sidebar on the left lists files including 'greetings.js', 'helloWorld.js', 'import.js', and 'index10.mjs'. The main editor window displays the code in 'index10.mjs', which defines two exported functions: 'add' and 'subtract'. The terminal at the bottom shows the command 'node "/home/anuj/Desktop/Mobzway/import.js"' being executed, resulting in the output: 'Hello, World!', 'Hello, World!', and 'Goodbye, World!'.

```
1 export function add(a, b) {
2   return a + b;
3 }
4
5 export function subtract(a, b) {
6   return a - b;
7 }
8
```

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



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.

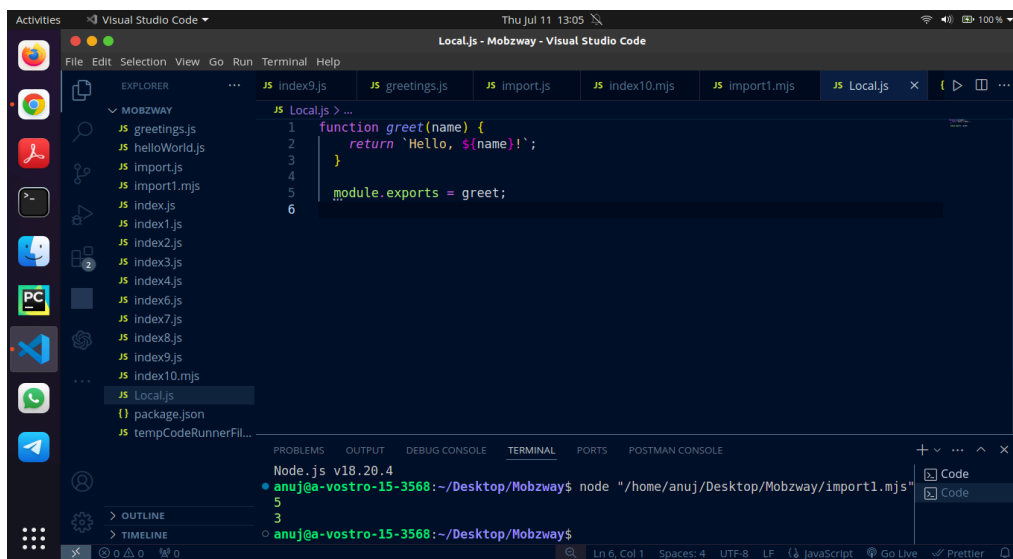
Examples of Built-in Modules:

- **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**

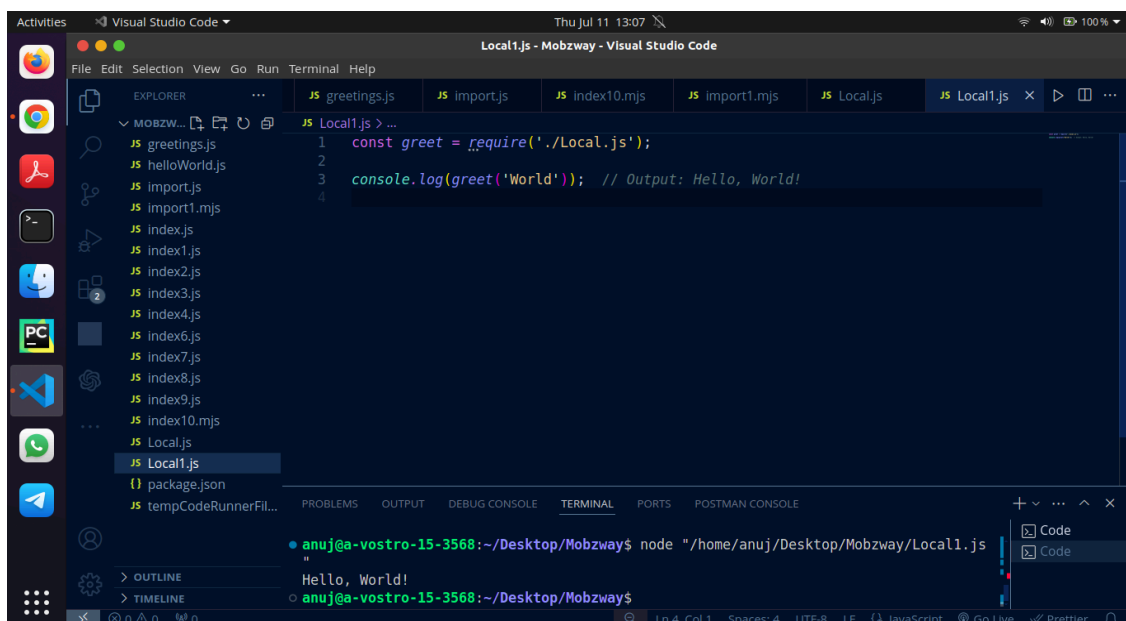
Local modules are user-defined modules that are stored locally in your project and used within the project. They can be reused across different files of the same project.



```
JS Local.js > ...
1 function greet(name) {
2   return `Hello, ${name}!`;
3 }
4
5 module.exports = greet;
6
```

Terminal output:

```
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$
```



```
JS Local1.js > ...
1 const greet = require('./Local.js');
2
3 console.log(greet('World')); // Output: Hello, World!
4
```

Terminal output:

```
anuj@a-vostro-15-3568:~/Desktop/Mobzway$ node "/home/anuj/Desktop/Mobzway/Local1.js"
Hello, World!
anuj@a-vostro-15-3568:~/Desktop/Mobzway$
```

- **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.