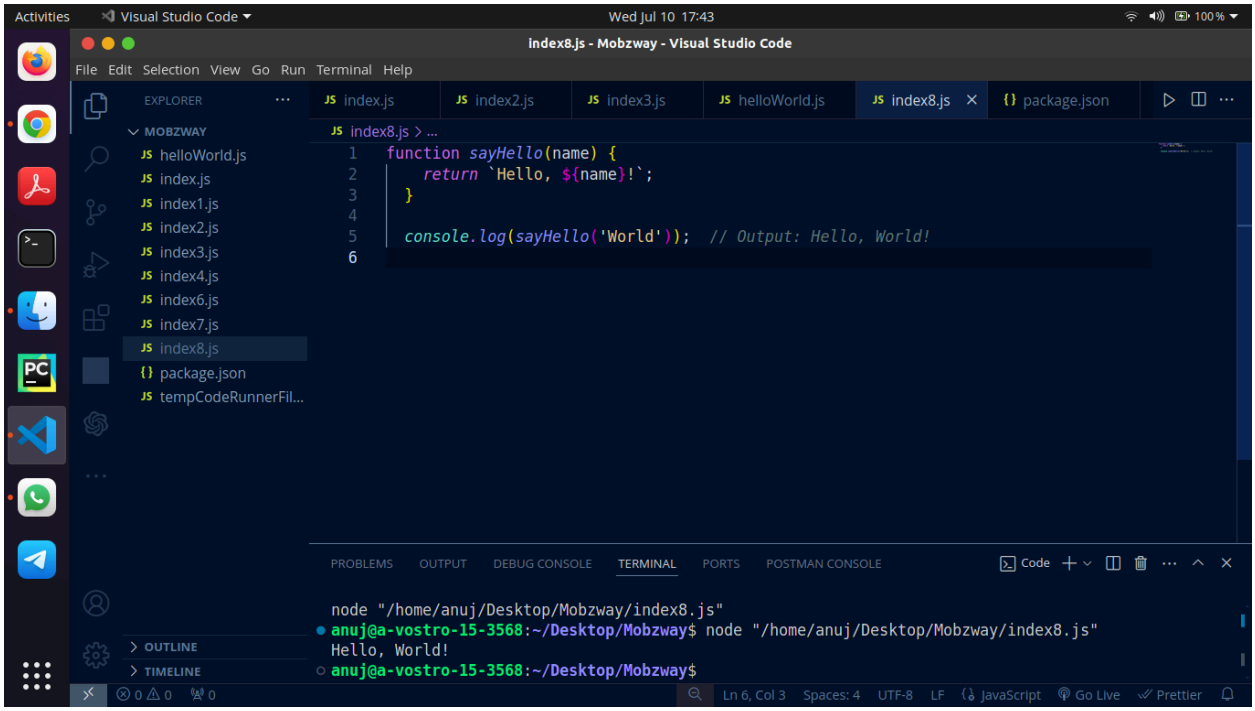
Node.js Documentation

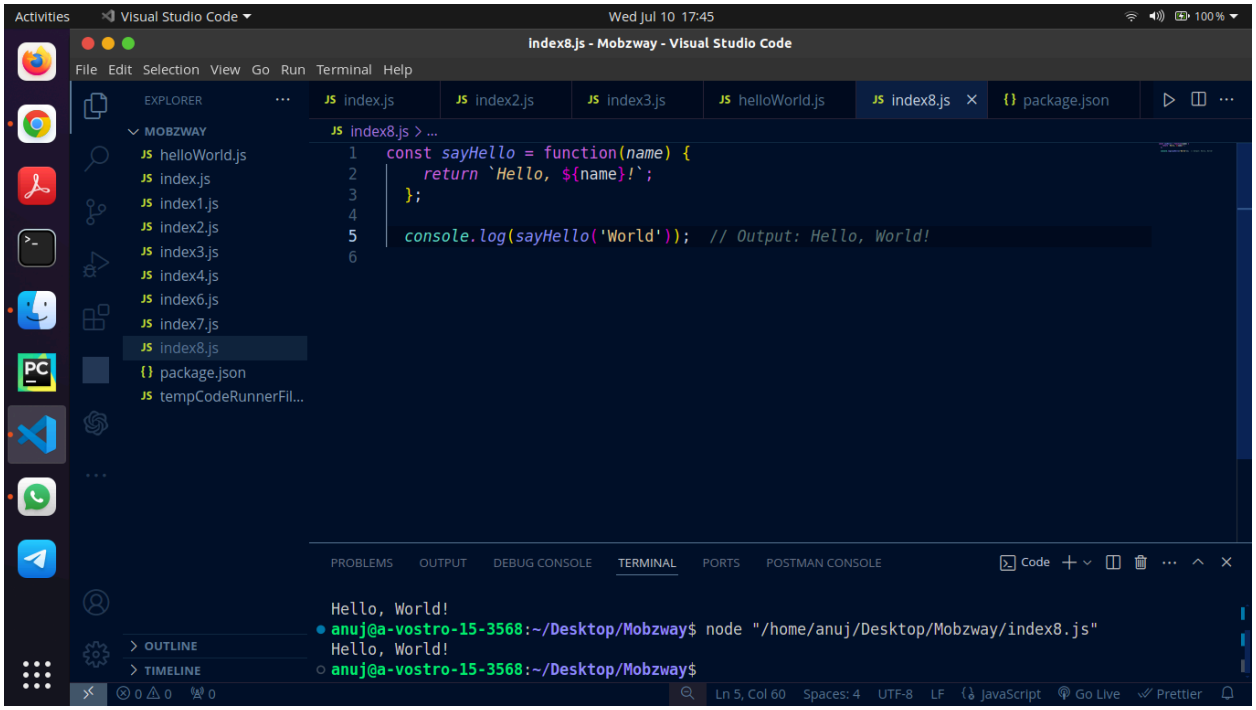
Module 3: Node JS Modules

- **Functions**

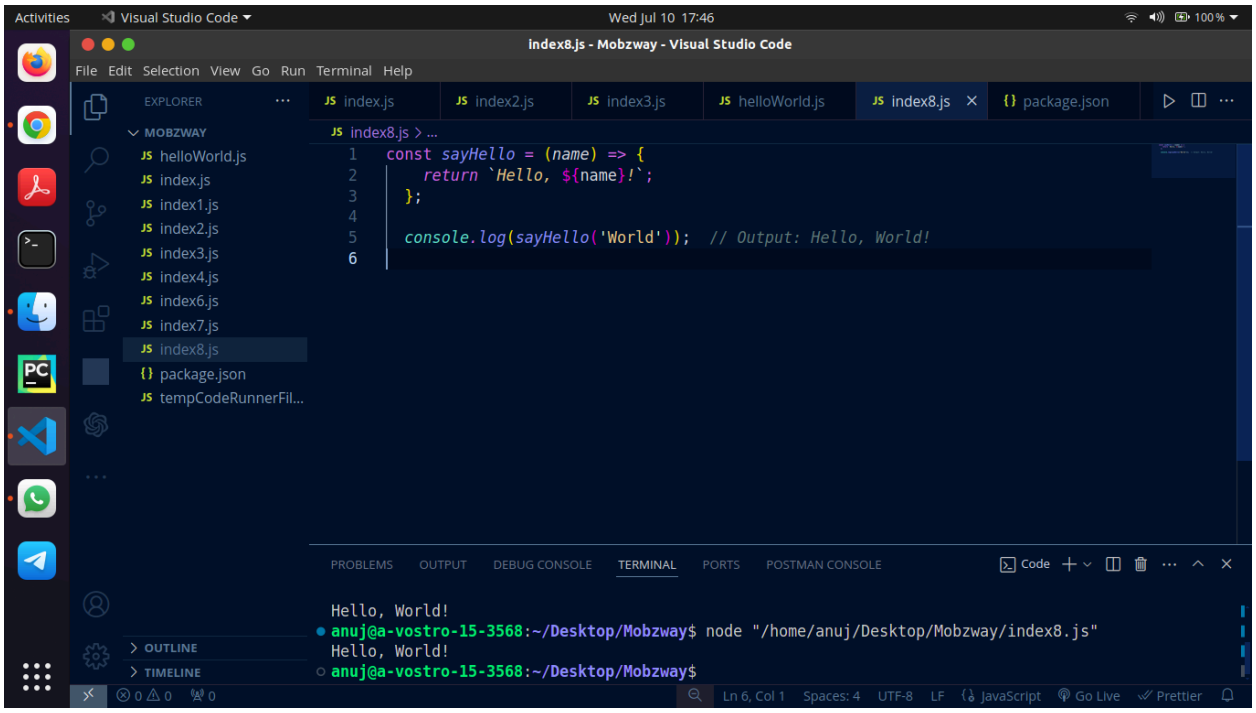
Named Functions



Anonymous Functions

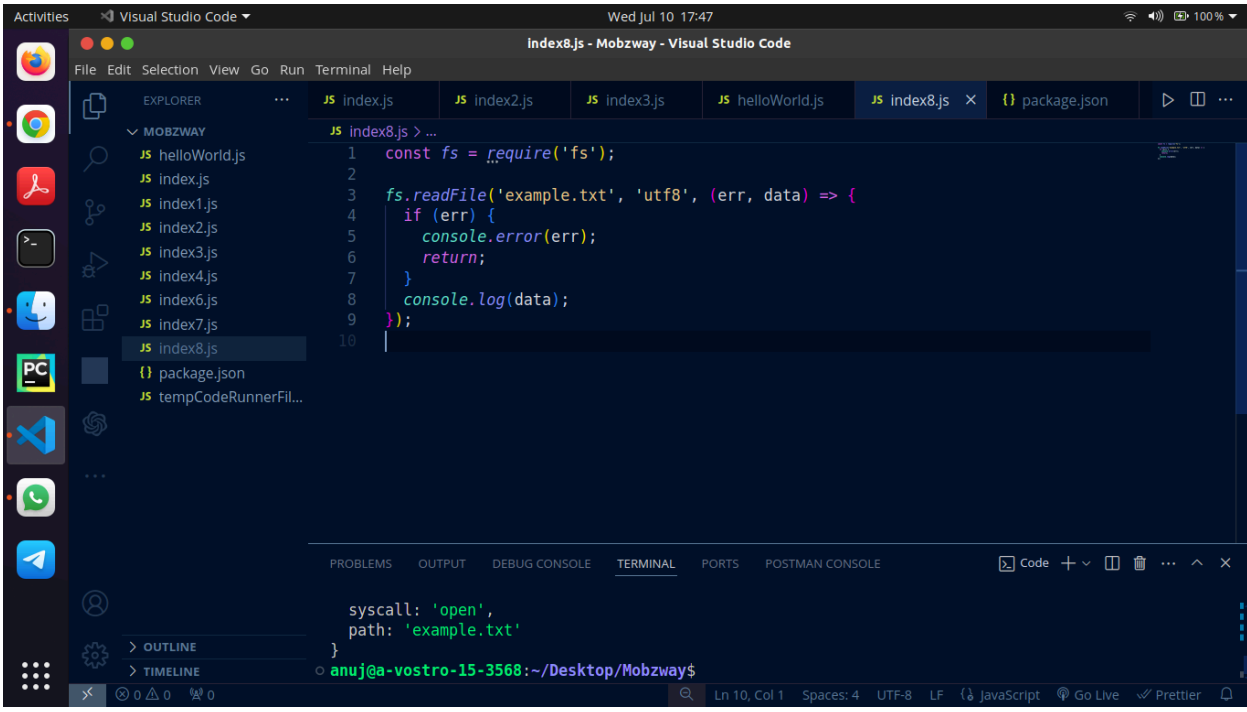


Arrow Functions

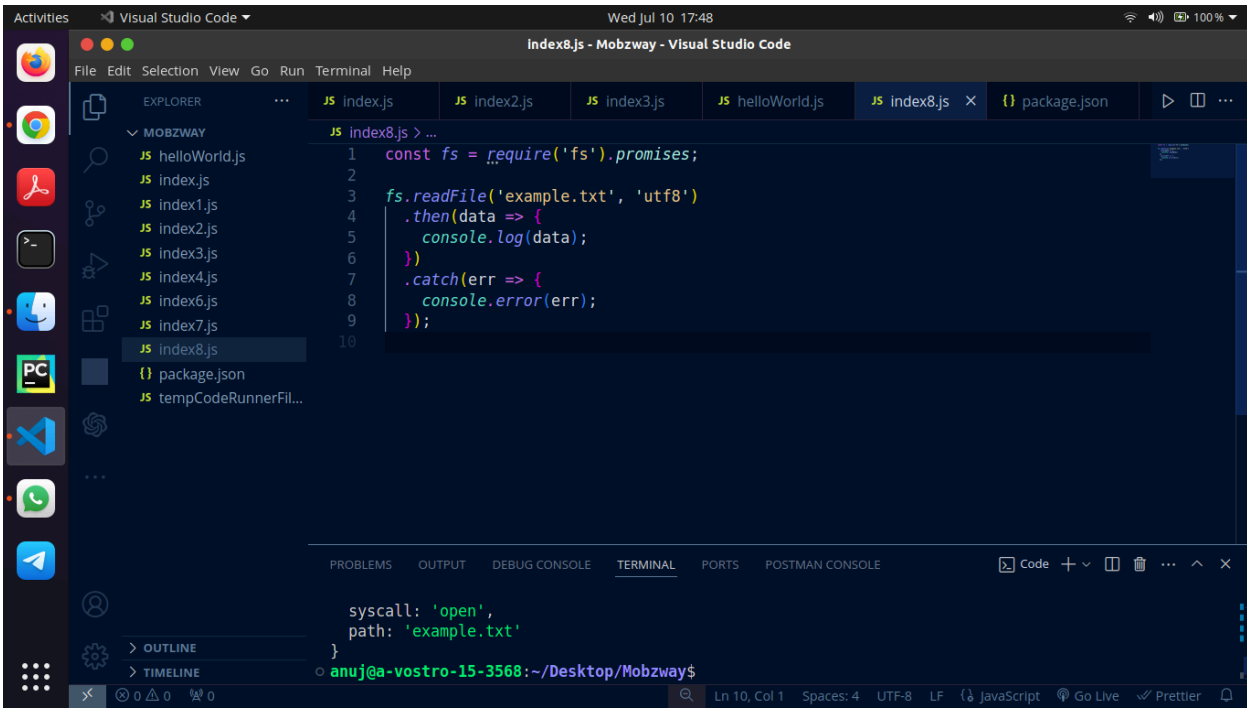


Asynchronous Functions

1. Callbacks

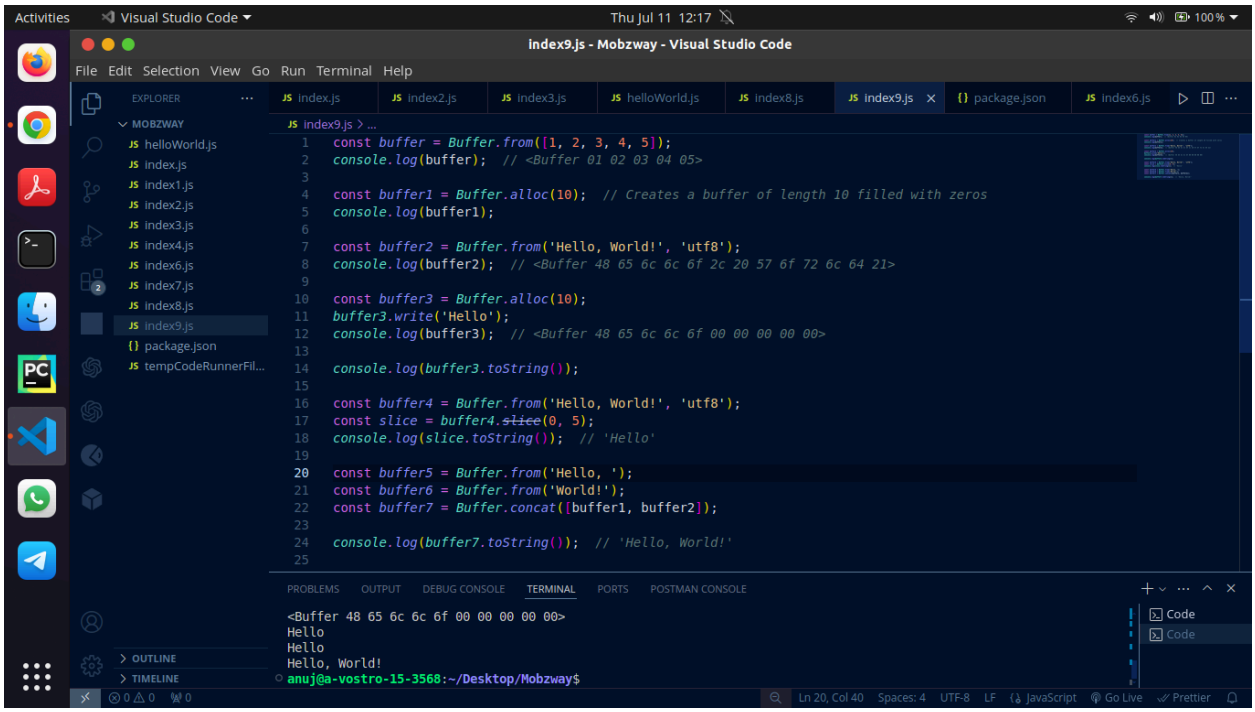


2. Promises



- ## 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.



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

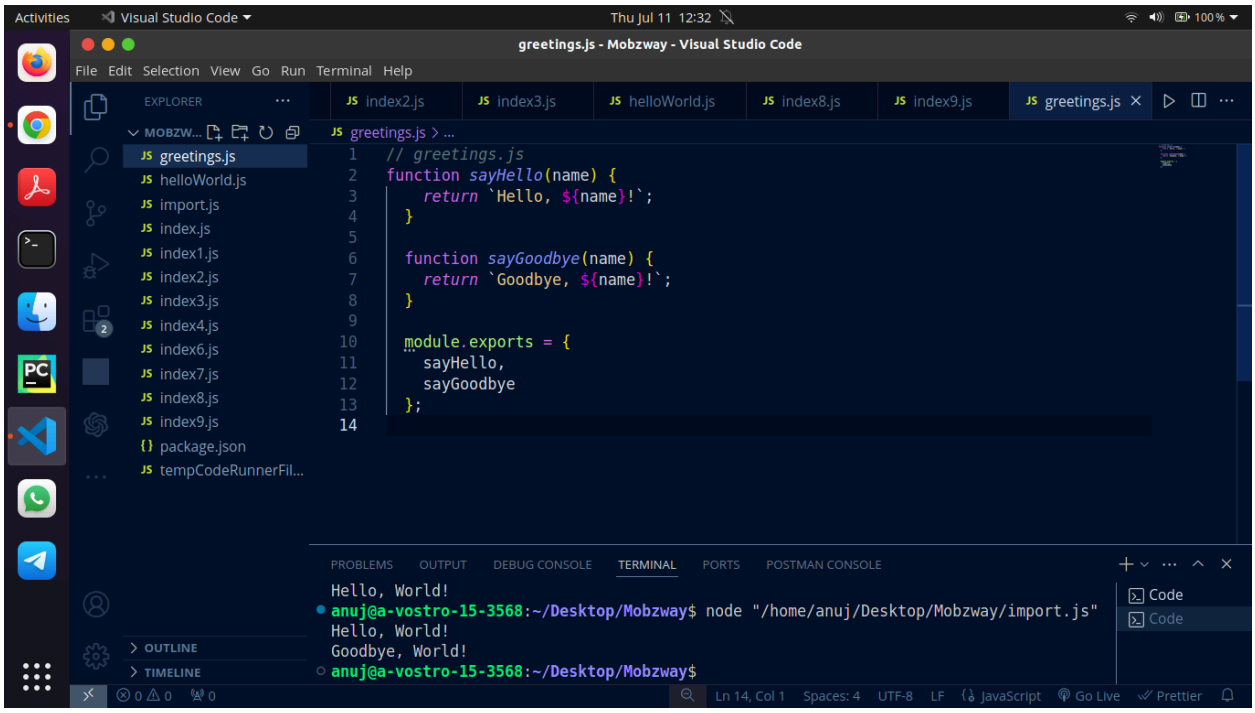
You can concatenate multiple buffers into a single buffer.

- ## 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.

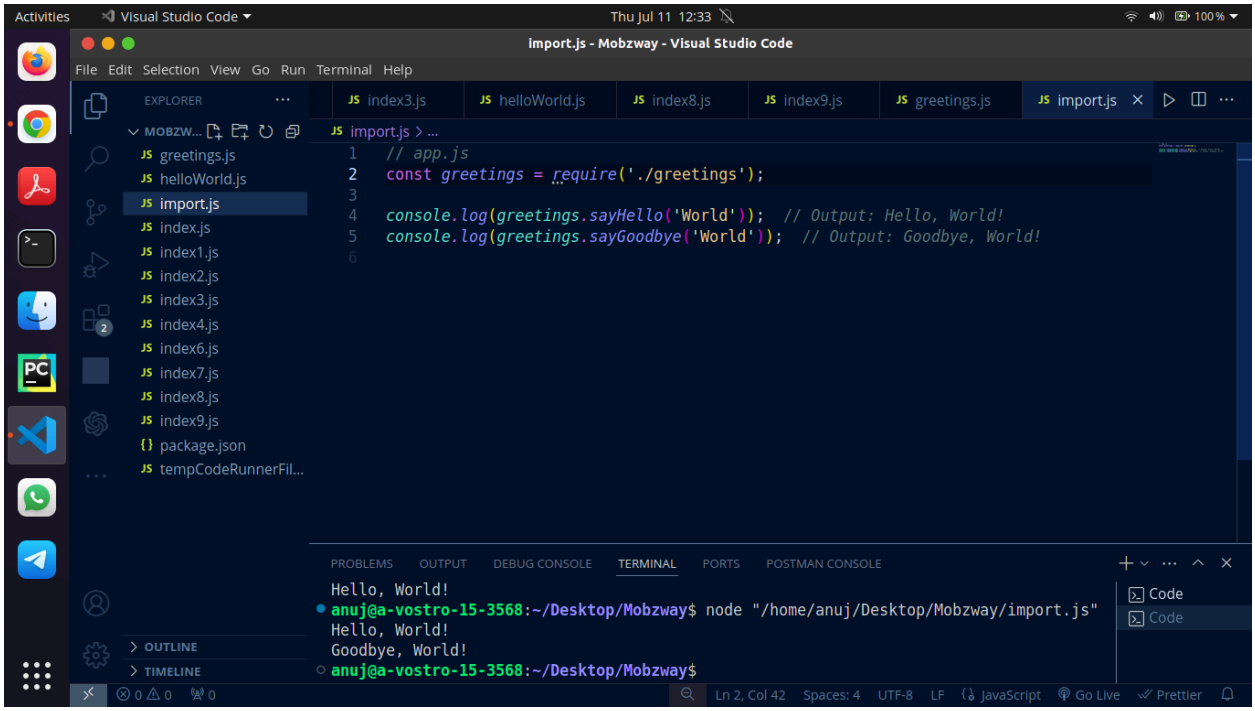
Creating and Exporting a Module

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



Importing a Module

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



- ## Modules Types

<u>Built-in (Core) Modules</u>

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.

<u>User-defined Modules</u>

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.

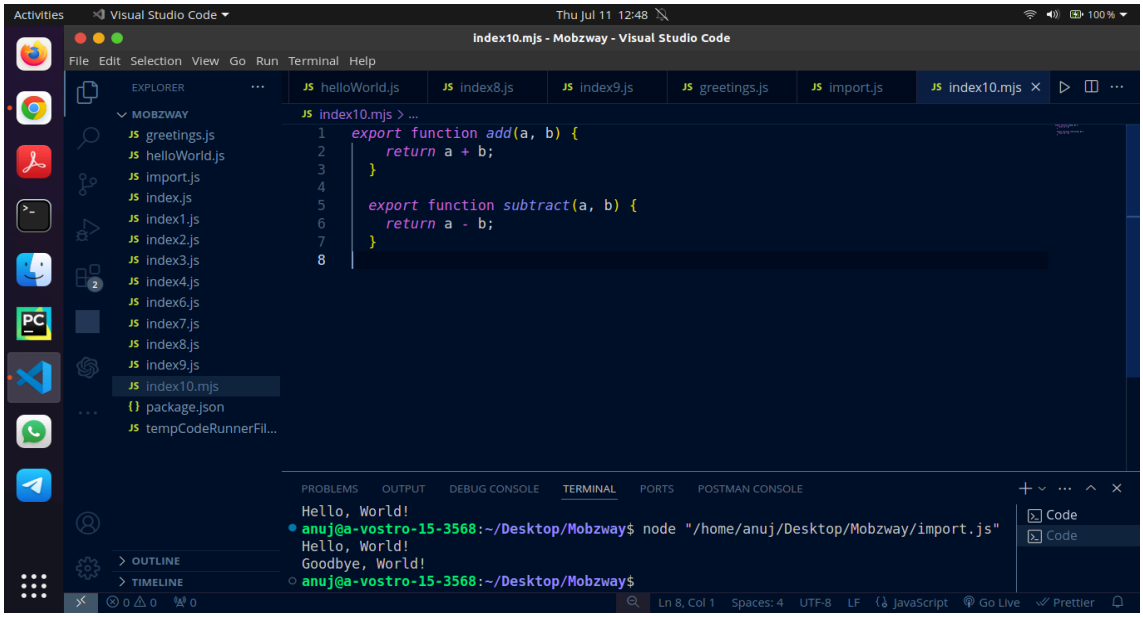
Examples of Popular Third-party Modules:

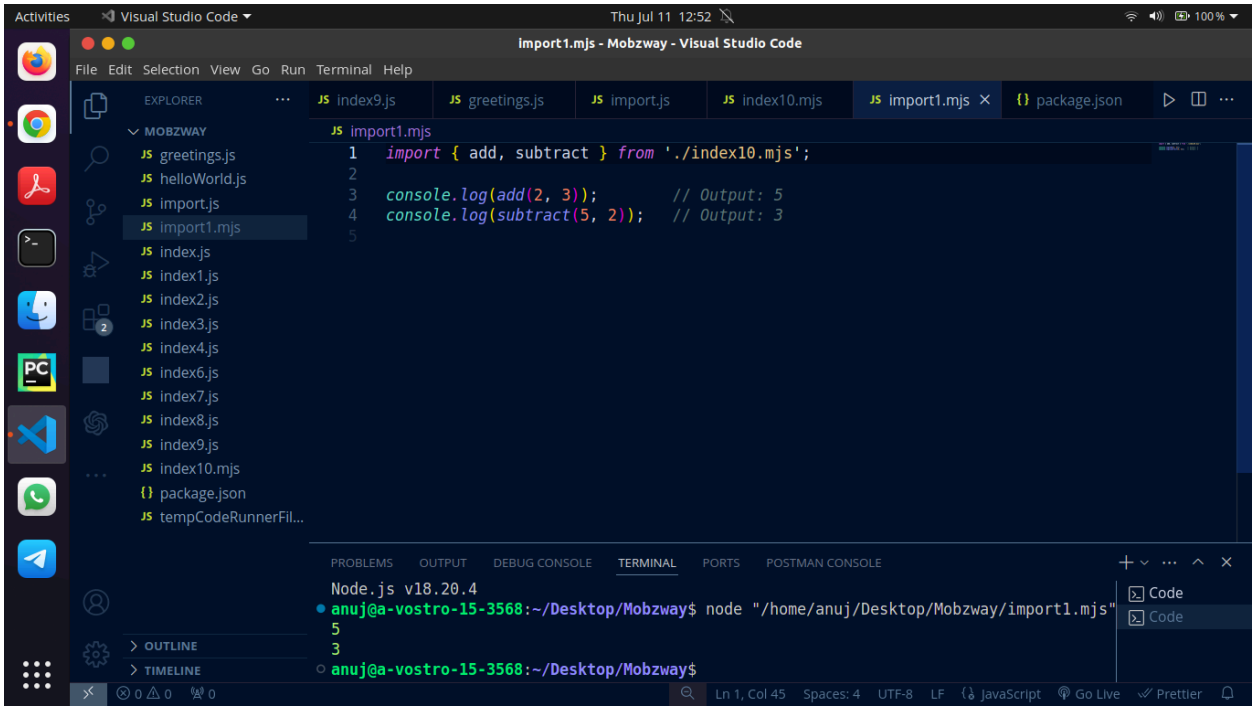
- `express`: Web application framework.
- `lodash`: Utility library for JavaScript.
- `mongoose`: MongoDB object modeling tool.

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.





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.

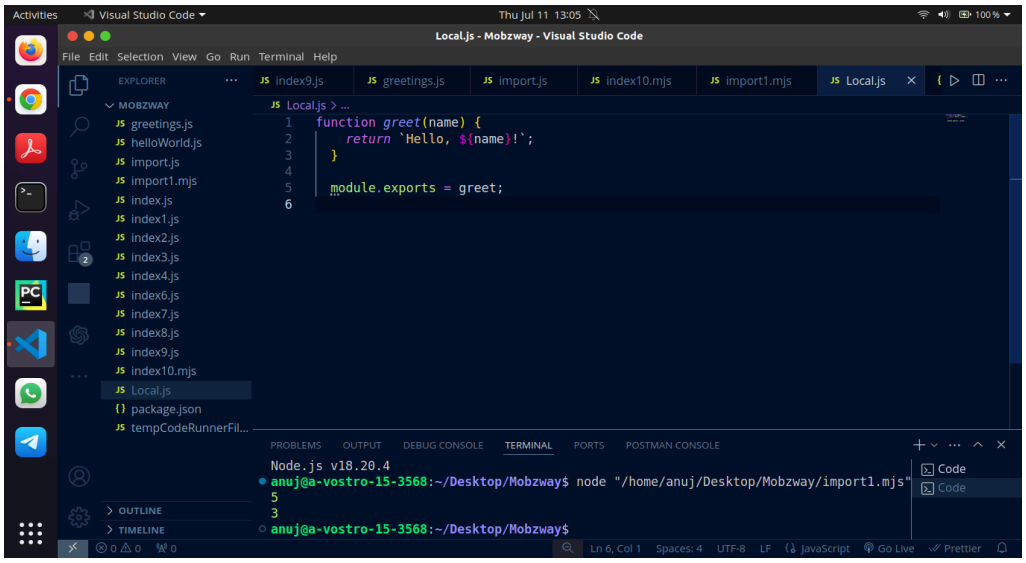
Examples of Built-in Modules:

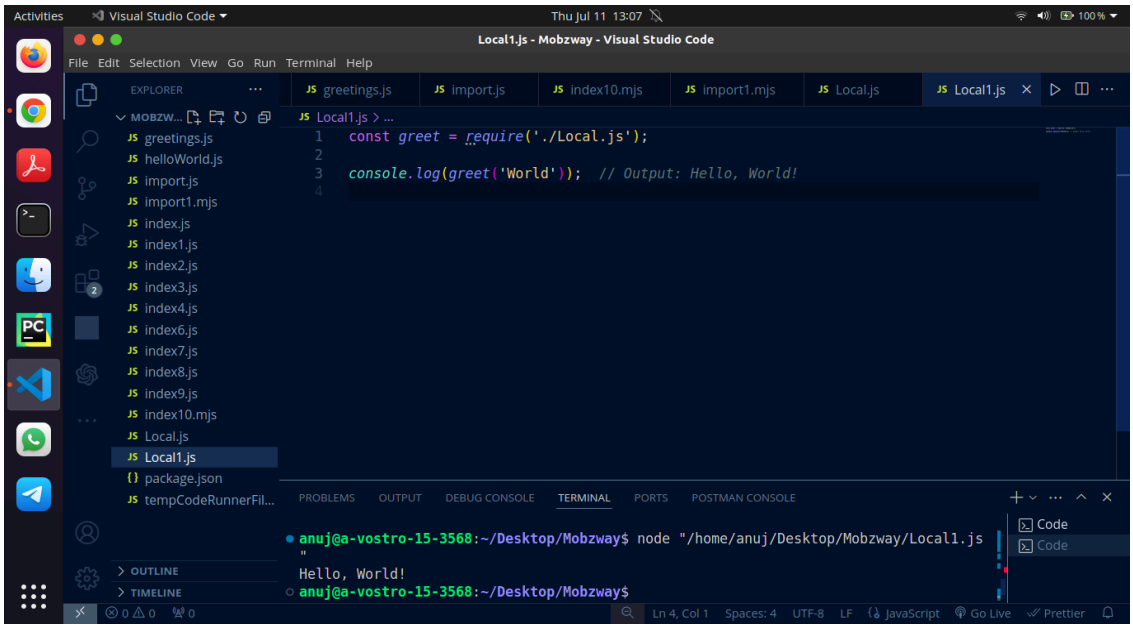
- `fs`: File system operations.
- `http`: HTTP server and client functionality.
- `path`: Utilities for working with file and directory paths.
- `os`: Information about the operating system.

- `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.





- ## 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

  <u>Exporting a Single Function or Object</u>

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

  <u>Exporting Multiple Functions or Objects</u>

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