**Date:24/06/2024**

**Section 2: Introduction to Node JS and NPM**

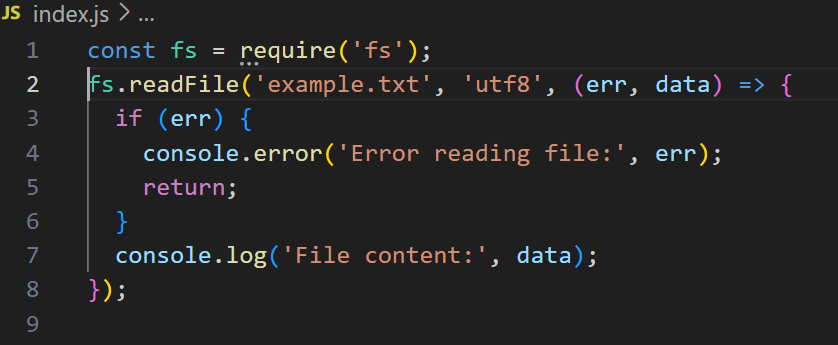
# **What is Node JS and Why use it?**

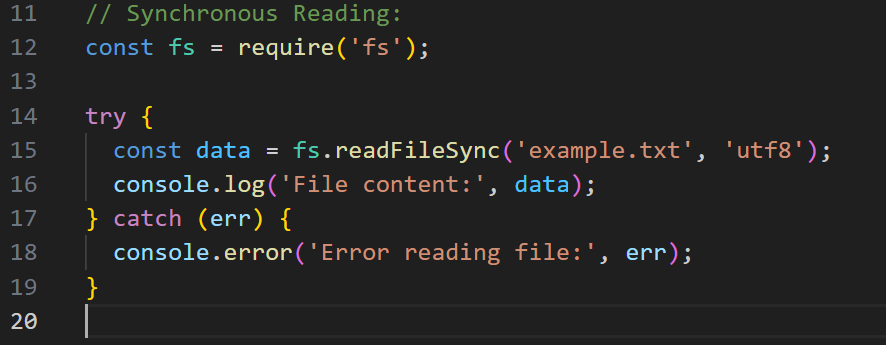
* Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project.
* Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
* Node.js uses JavaScript on the server.
* Node JS is a tool that allows you to run JavaScript on server, making it possible to create fast and efficient web applications.
* Node JS is free and Single threaded based on event driven, on-blocking I/O model.
* Node JS is used because it allows for fast, efficient, and scalable we application, handles many connections simultaneously, and has large library of packages to extend its functionality.
* NPM:-(node package manager) huge library of open-source package available for everyone for free.
* Running JavaScript outside the browser?
* Yes Node.js allows you to run JavaScript code outside of a web browser, making it possible to build server-side application and scripts using JavaScript.

|  |
| --- |
|  |

* Modules:
* Definition: Modules in node are reusable lock of code that encapsulate specific functionality.
* Usage: They are included in other files using “**require**” function.
* Benefits: Improve code organization and maintainability by Breaking down application into smaller, manageable pieces.
* Core Modules in Node.js:
* Definition: Built-in modules that comes packages with node.js.
* Eg. ‘http’ , ’fs’ ,’path’ , ‘os’ , ‘eve’ , ‘util’ , ;stream’ , ‘buffer’ , ‘crypto’
* Usage: Provide fundamental functionalities like creating servers, interacting with file systems handling events, etc.



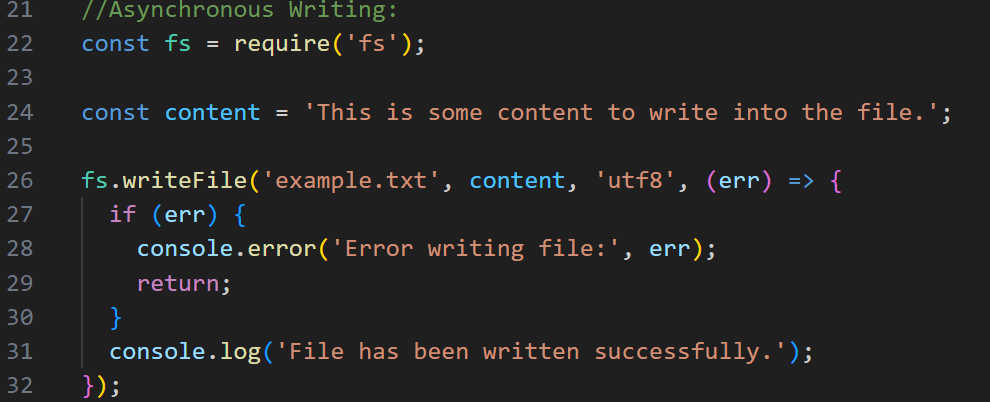
* Reading and Writing files:
* Node.js provides the fs (file system) core module to interact with the file system. This module allows you to read from and write to files easily.
* Reading Files:
* To read files, you can use the **fs.readFile** method, which is asynchronous, or the **fs.readFileSync** method, which is synchronous.
* Asynchronous Reading: 
* ‘fs.readFile’: Reads the file asynchronously.
* Arguments:
* **filename**: The path to the file.
* **encoding**: (optional) The encoding of the file, typically 'utf8'.
* **callback**: A function that gets called after the file is read, with err and data as its arguments.
* Synchronous Reading:



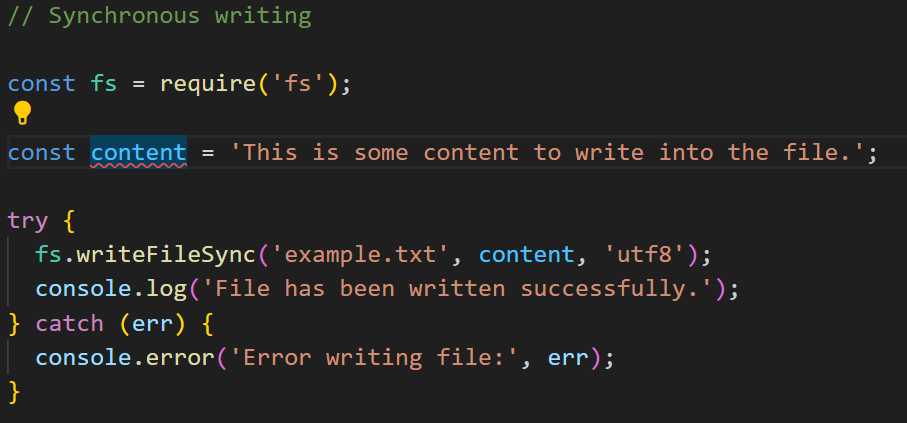
* **fs.readFileSync** : Reads the file synchronously, blocking the execution until the operation is complete.

Writing files:

* To write files, you can use the **fs.writeFile** method, which is asynchronous, or the **fs.writeFileSync** method, which is synchronous.
* Asynchronous Writing:



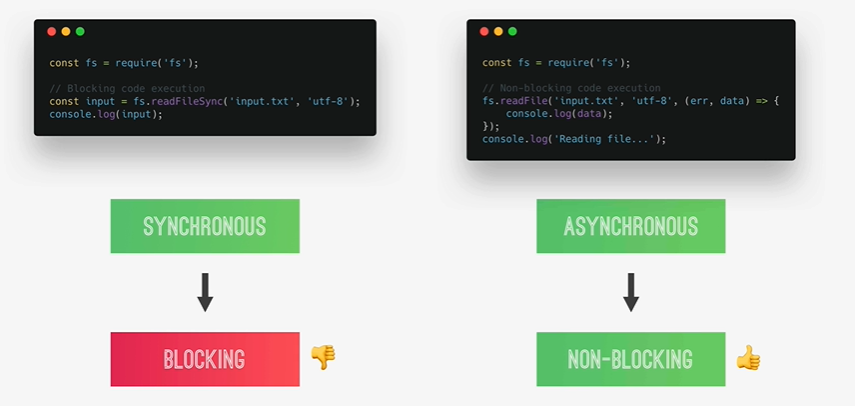
* ‘fs.writeFile’: Writes the content to the file asynchronously.
* Arguments:
* **filename**: The path to the file.
* **Data:** The content to write.
* **encoding**: (optional) The encoding of the file, typically 'utf8'.
* **callback**: A function that gets called after the file is written, with **err** as its arguments.
* Synchronous Writing:



* **fs.writeFileSync**: Writes the content to the file synchronously, blocking the execution until the operation is complete.

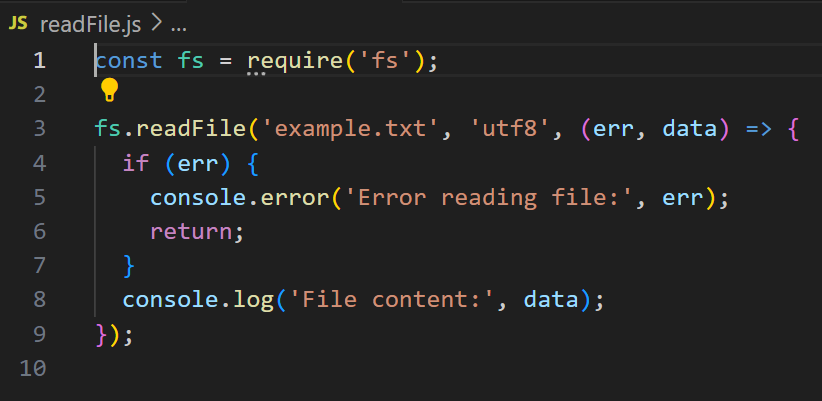
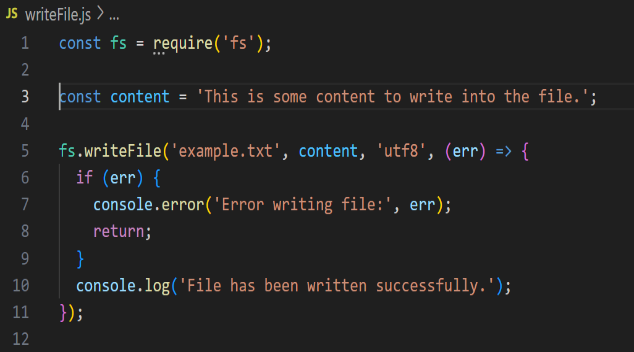
**Date: 25/06/2024**

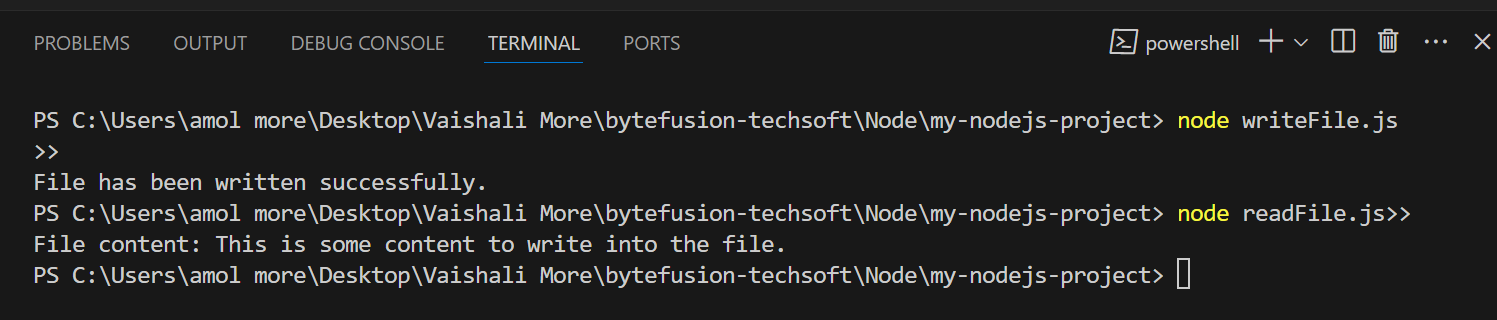
* Blocking and Non-Blocking: Asynchronous Nature in Node.js
* **Blocking**: A blocking operation stops the execution of the program until the operation completes. In other words, the program waits for the operation to finish before moving on to the next task.
* **Non-Blocking**: A non-blocking operation allows the program to continue executing other tasks while the operation is still in progress. The program does not wait for the operation to complete and can handle multiple operations concurrently.



* Reading and Writing files asynchronously in Node.js

Write and Read File:



Output:

* Creating a Simple Web Server:
* Node.js provides a built-in module called http that allows you to create web servers.
* Steps:
* **Step1**: **Create a new directory for your project:**

mkdir **my-web-server**

**cd my-web-server**

**Initialize a new Node.js project: npm init -y**

This command creates a **package.json** file with default settings.

* **Step2**: Create the server file:

Create a new file named **server.js**

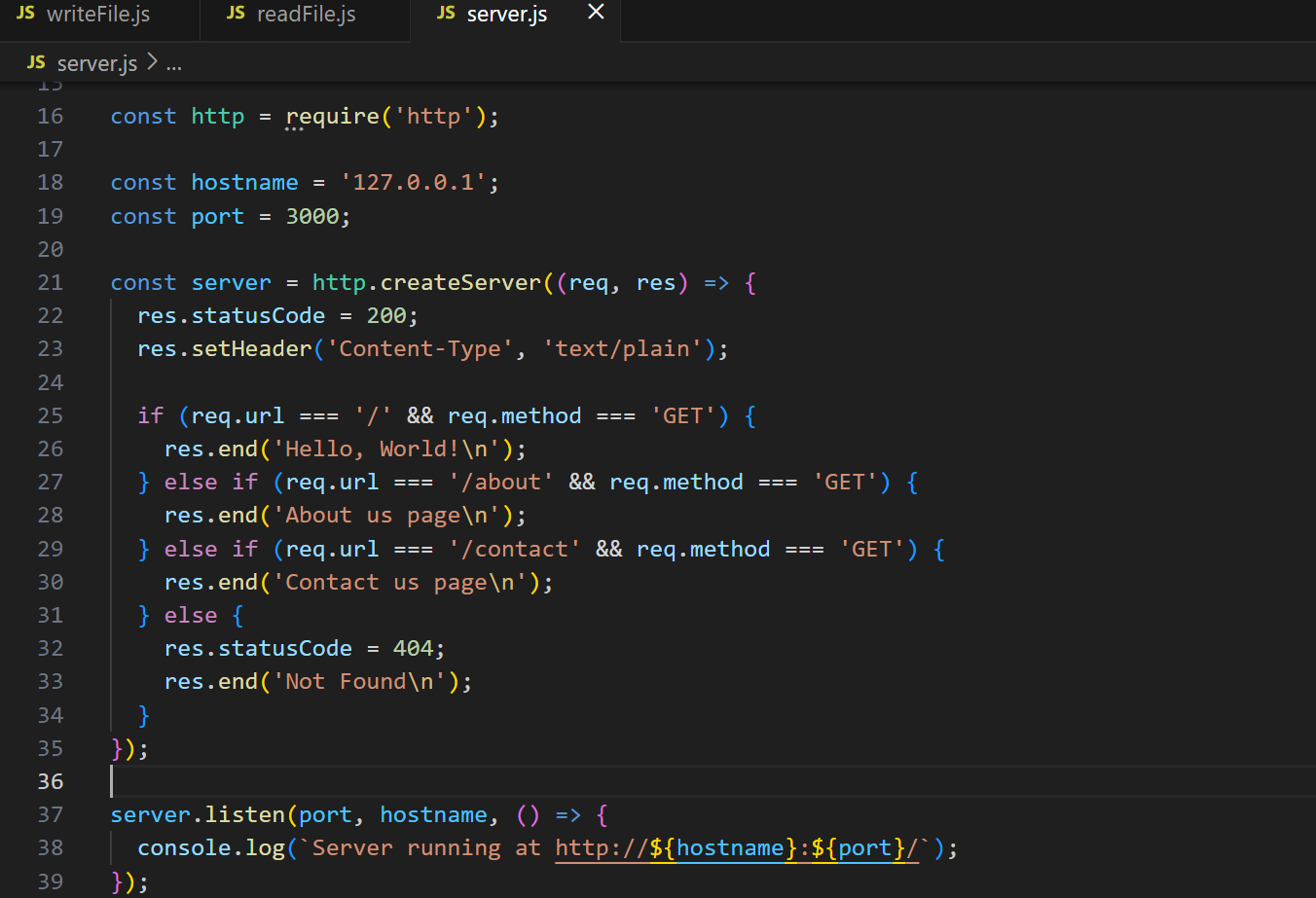
Add the following code to **server.js**

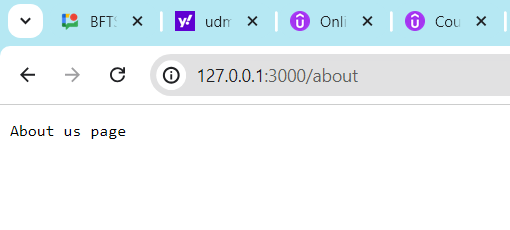
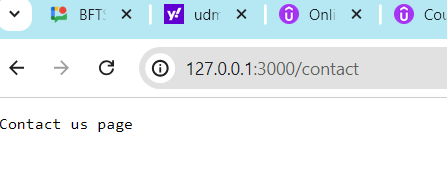
* **Step3**: Run the Server

node **server.js**

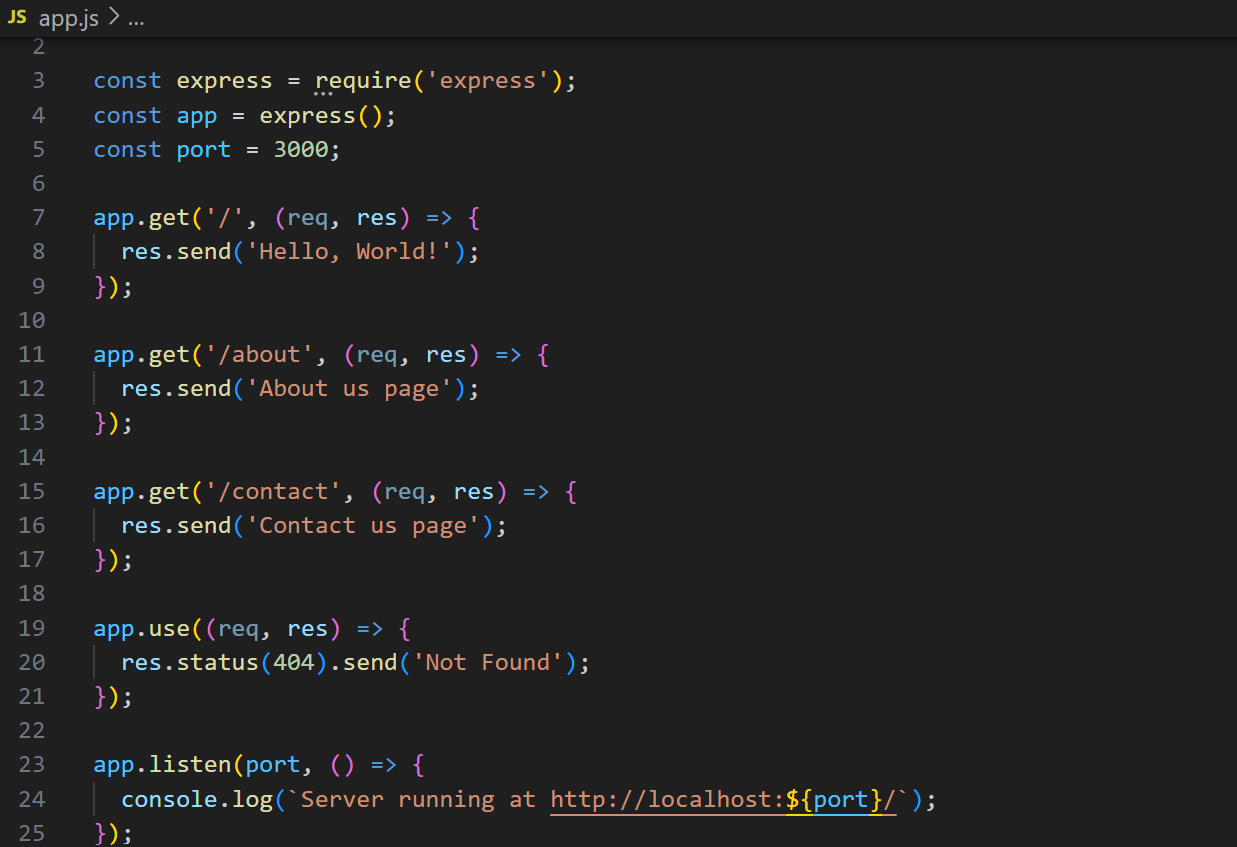
****

* Routing:
* Using both the built-in ‘**http’** module and’**Express**.**js’** for routing.
* Using http module:

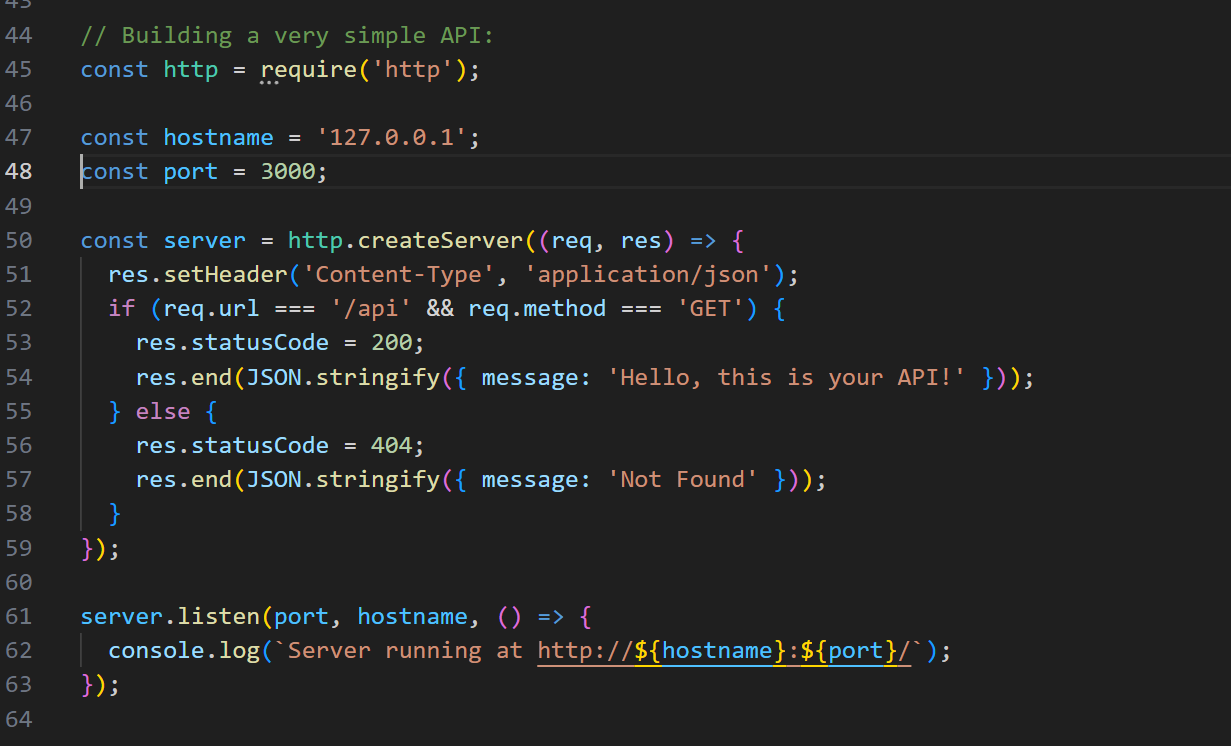




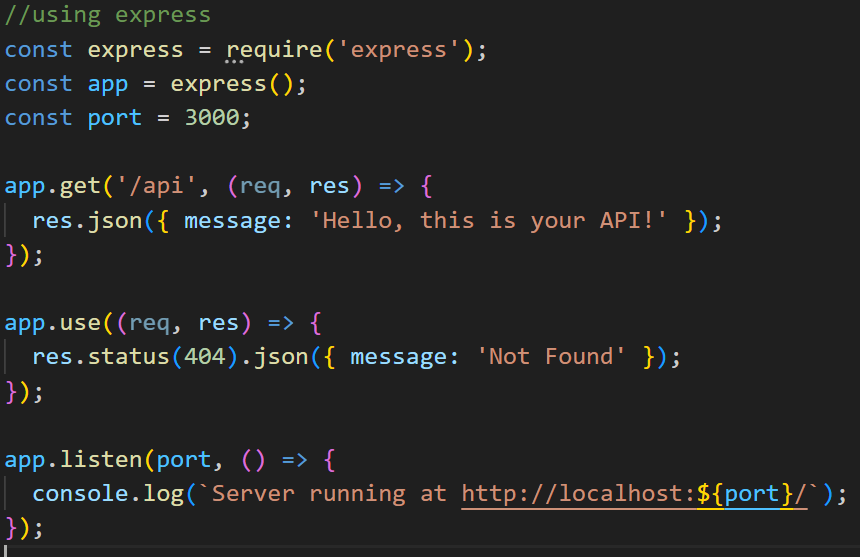
* Using Express.js:
* Install Express: npm install **express**
* Creating a new file named app.js



* Building a (Very) Simple API:
* **Using the http module**: A basic API with manual routing and response handling

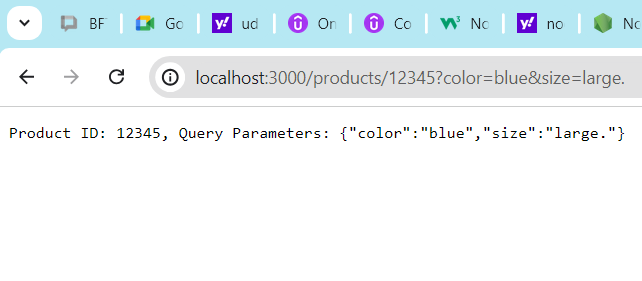


* **Using Express.js**: A cleaner and more efficient way to build APIs with built-in routing and response handling.

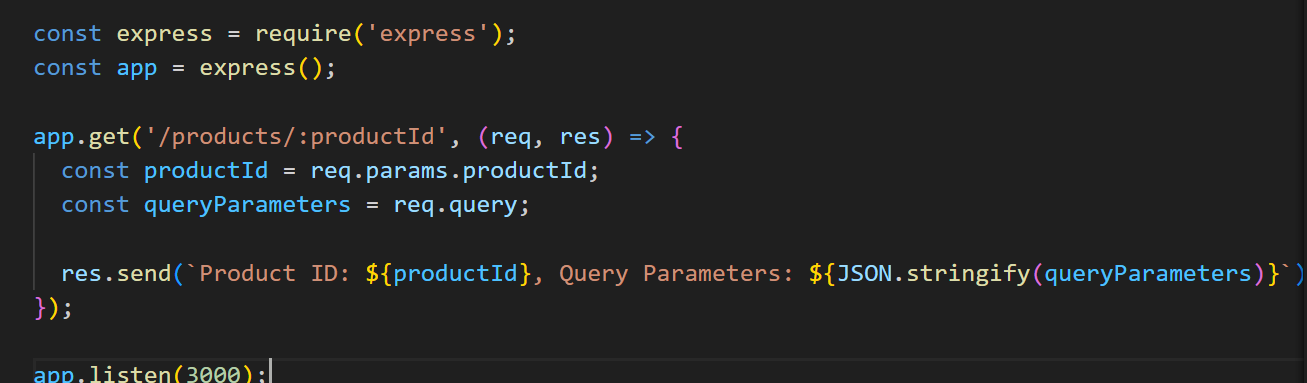


* HTML Templating: Building the Templates:
* HTML templating in Node.js allows you to generate HTML dynamically by combining HTML structure with data from your application. This is particularly useful for rendering dynamic content in web applications. Here are the key points to understand about HTML templating in Node.js:
* **Advantages**:
* **Code Reusability**: Templates help in separating HTML structure from JavaScript logic, promoting cleaner code.
* **Dynamic Content**: Easily inject dynamic data into HTML responses based on application state or user input.
* Filling the Templates:
* Here comes the fun part, filling our templates by replacing our placeholders with actual content.
* When a user requests a URL, the code reads the relevant template file (either template-overview.html or template-card.html) synchronously, fills it dynamically with content from a JSON file, and sends back the relevant content as a response to the user.
* This is achieved through the use of the replace Template function, which replaces the placeholders in the template with actual content.
* There are several other templating engines available in Node.js, such as EJS, Pug (formerly known as Jade), Handlebars, and Mustache, among others.
* Parsing Variables From URLs
  1. **Using the URL Module (Basic Node.js)**
* When a user requests a URL, the code reads the relevant template file (either template-overview.html or template-card.html) synchronously, fills it dynamically with content from a JSON file, and sends back the relevant content as a response to the user.
* Helps extract information from URLs in server-side applications.

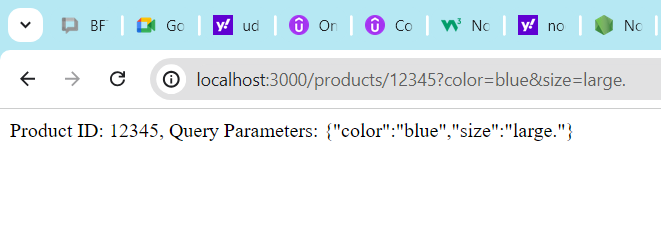


 **Output**:

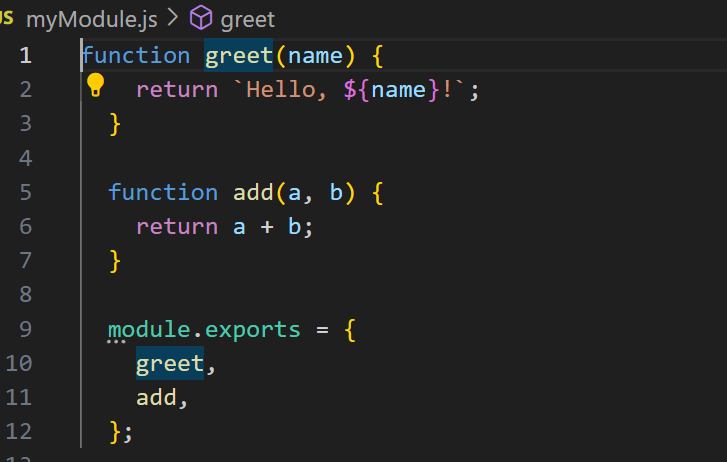
* 1. **Using Express.js**



**Output**:



* Using Modules: Our Own Modules:
* In Node.js, you can create your own modules to organize and reuse code efficiently. Here's a brief guide on how to create and use your own modules:

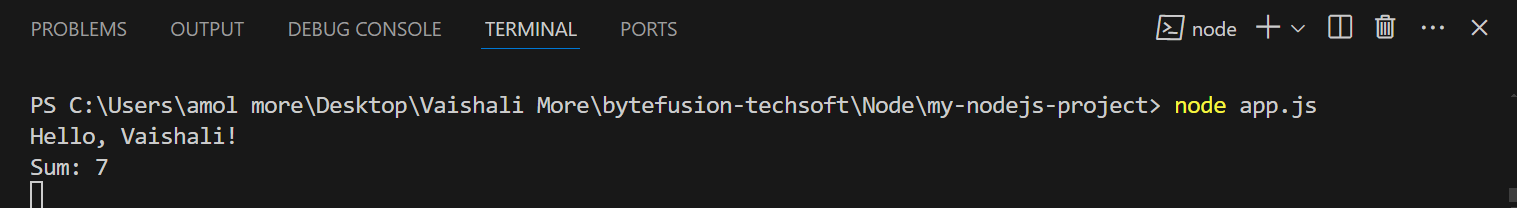
****1.Create a new file for your module: For e.g., create a file named **myModule.js**

2.Define function or variables in ‘**myModule.js’**

3.Used the module in another file

****4. Create another file to use the module. For e.g., **app.js**

5.import and use the module in **‘app.js’**



* Introduction to NPM and the Package. Json:

**1.NPM (Node Package Manager).**

* **Purpose**: Manages packages and dependencies for Node.js projects.

**Key Commands**:

** Initialize a Project:** npm init -y

 **Install a Package:** npm install <package-name>

 **Install Globally**: npm install -g <package-name>

 I**nstall Dev Dependencies**: npm install <package-name> --save-dev

**2.Package.json:**

** Purpose**: Configuration file that stores project metadata and dependencies.

**package. Json**: Tracks project details and dependencies.

**Key Fields**:

* **name**: Project name.
* **version**: Project version.
* **scripts**: Commands to run tasks (e.g., start, test).
* **dependencies**: Packages required for the project.
* **devDependencies**: Packages needed only for development.



* Types of Packages and Installs:
* Theare are Two Types of Packages.

1.Regular Dependencies:

* **Purpose**: Required for your application to run
* **Install Command**: npm install <package-name>
* **Example**: npm install express

2. Development Dependencies.

* **Purpose**: Needed only during development (e.g., testing frameworks, build tools).
* **Install Command**: Needed only during development (e.g., testing frameworks, build tools).
* **Example**: npm install nodemon --save-dev

3. Global Package:

* **Purpose**: Installed globally to be used across multiple projects, usually CLI tools.
* **Install Command**: npm install -g <package-name>
* **Example:** npm install -g create-react-app

4.Optional Dependencies.

* **Purpose**: Optional packages that do not prevent the main package from working if they fail to install.
* **Install Command**: npm install <package-name> --save-optional
* **Example**: npm install fsevents --save-optional

5. Peer Dependencies.

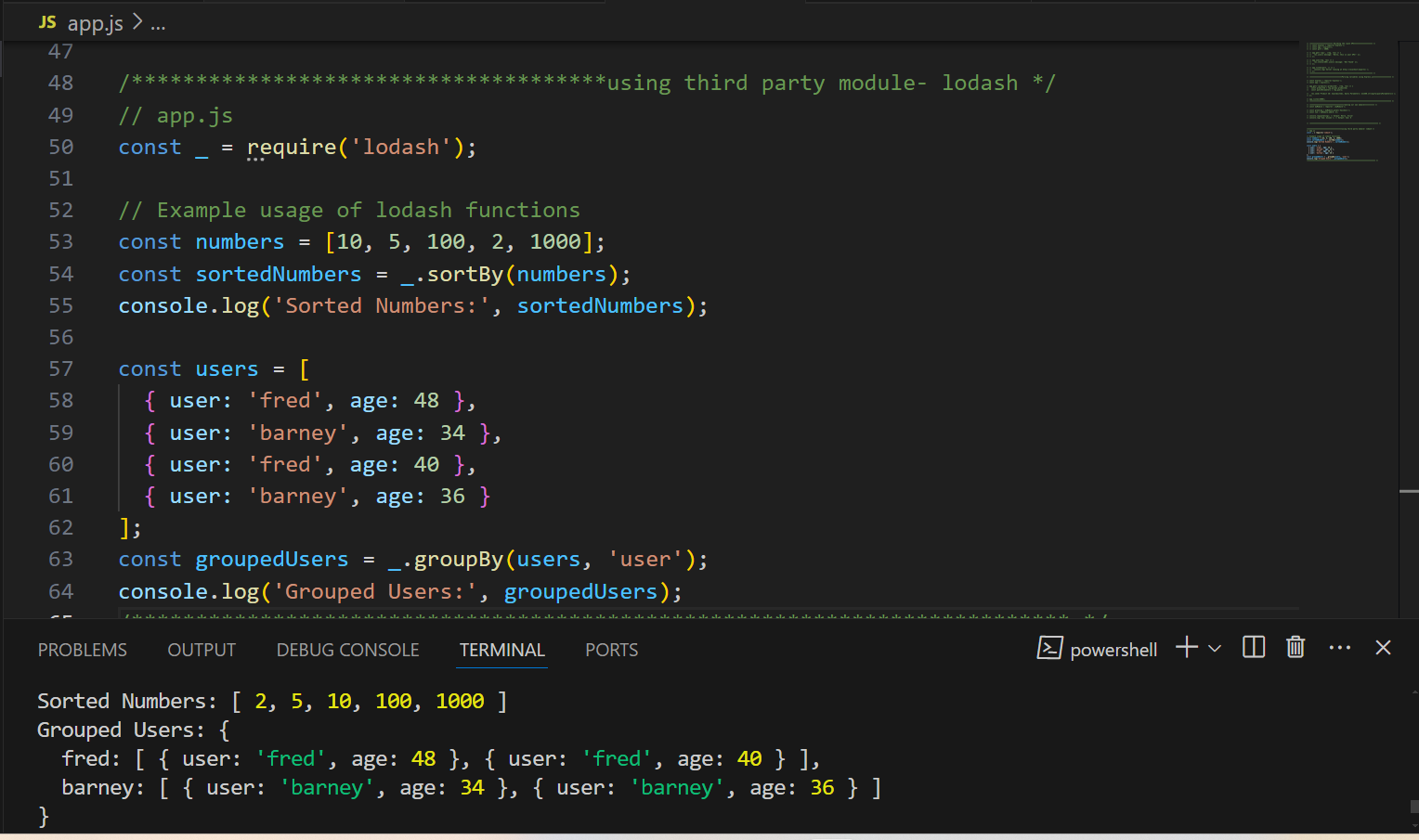
* **Purpose**: Packages that your package expects the consuming project to also have installed.
* **Install Command**: Not installed automatically, but listed in package. Json to indicate they should be installed.
* **Example**:

"peerDependencies": {

"react": "^16.0.0"

}

* Using Modules 3: Third Party Modules:
* These modules can be used to add functionality to your Node.js applications without having to write Third-party modules are packages created by the community and shared via NPM (Node Package Manager). everything from scratch.
* **Example**: Using the **lodash** Module:
* lodash is a popular utility library that provides helpful functions for common programming tasks.
* Install the ‘lodash’ Module: **npm install lodash.**
* Create a JavaScript File and Use **lodash.**

****

* Package Versioning and Updating:
* Proper package versioning and updating are crucial to maintaining a stable and secure Node.js application. Here's a brief guide on how to handle these tasks using NPM (Node Package Manager).
* Understanding Semantic Versioning (SemVer) :

1. **MAJOR**: Breaking changes.
2. **MINOR**: New features, but backward-compatible.
3. **PATCH**: Bug fixes, backward-compatible.

* Install Specific Version**: npm install <package-name>@<version>**
* **Example**: npm install [express@4.17.1](mailto:express@4.17.1)
* Checking for Updates:
* Check for outdated Package**: npm outdated**
* Updating Package:
* Update Specific Package: **npm update <package-name>**
* Update All Package: **npm update**
* Update to the latest version: **npm install <package-name>@latest**
* Setting Up Prettier in VS Code:
* Prettier is a popular code formatter that helps maintain consistent code style across your project. Here’s how to set up Prettier in Visual Studio Code (VS Code):
* Recap and what’s Next:
* Prettier is a popular code formatter that helps maintain consistent code style across your project. Here’s how to set up Prettier in Visual Studio Code (VS Code):

**Section 3: Introduction to Backend Web Development**

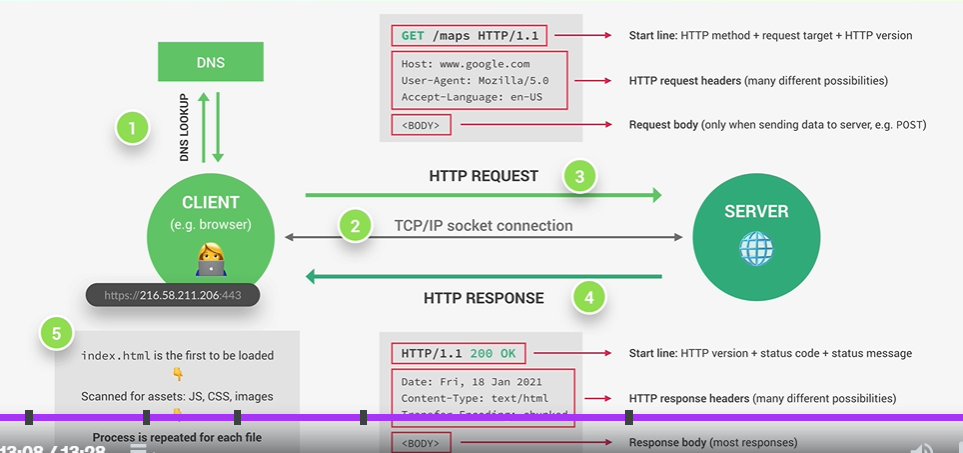
# **Section Intro:**

* Backend web development focuses on the server side of web applications, involving server-side programming, database management, API creation, server management, and security. Key components include:
* Backend development ensures robust, secure, and efficient web applications by managing the server-side functionality and communication with the frontend.

1. **Languages:** Python, JavaScript, Ruby, PHP, Java, C#.
2. **Databases:** Relational (MySQL, PostgreSQL), NoSQL (MongoDB, Redis).
3. **APIs:** RESTful, GraphQL.
4. **Server Management:** Apache, Nginx, AWS, Azure.
5. **Security:** Authentication, HTTPS, protection against vulnerabilities.
6. **Version Control:** Git (GitHub, GitLab).
7. **Development Tools:** Visual Studio Code, unit testing, CI/CD (Jenkins, GitHub Actions).
8. **Frameworks:** Express.js, Django, Ruby on Rails, Spring Boot.

# **An How Overview of How the Web Works:**

* Backend web development focuses on the server side of web applications, involving server-side programming, database management, API creation, server management, and security. Key components include:
* Backend development ensures robust, secure, and efficient web applications by managing the server-side functionality and communication with the frontend.



# **HTTP in Action:**

* Node.js provides powerful capabilities for handling HTTP requests and building web servers. Here are the essential points to understand HTTP in Node.js:
  1. **HTTP Module**: Node.js includes a built-in http module that allows you to create HTTP servers and handle HTTP requests and responses.
  2. **Creating an HTTP Server**: Use http.createServer() method to create an HTTP server.
  3. **Handling Request and Responses**:
* The callback function (req, res) inside createServer handles each incoming HTTP request.
* req is an instance of http.IncomingMessage that contains request details (e.g., URL, headers).
* res is an instance of http.ServerResponse used to send the response back to the client.

**4.Routing:**

* Implement custom routing using req.url and req.method to handle different HTTP methods (GET, POST, etc.) and URLs.

**5.Sending Different Content Types:**

* Set different Content-Type headers for serving HTML, JSON, plain text, etc.

**6.HTTP Methods:**

* Handle different HTTP methods (GET, POST, PUT, DELETE) using req.method.

**7.Error Handling:**

* Implement error handling to manage unexpected errors and send appropriate error responses.

**6.Listening to Server Events:**

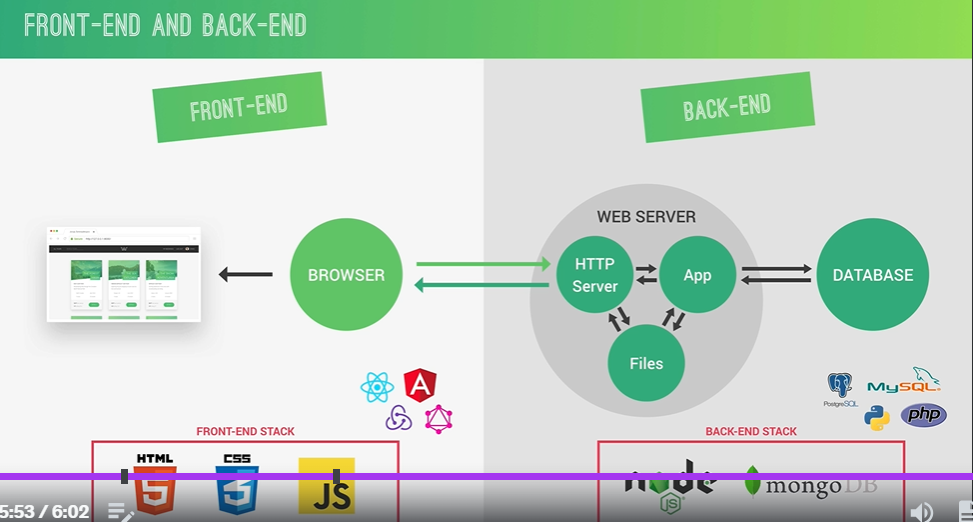
* Use server.on('event', callback) to listen to server events like listening, close, error, etc.

# **Front End vs Back End Web Development:**

#### **Frontend Development:**

* **Focus:** Client-side (what users see and interact with).
* **Technologies:** HTML, CSS, JavaScript.
* **Frameworks/Libraries:** React, Angular, Vue.js.
* **Role:** Design and implement user interfaces, ensure responsive design, handle user interactions.
* **Tools:** Web browsers, IDEs like Visual Studio Code, browser developer tools.
* **Responsibilities:** Creating layouts, styling, and ensuring a smooth user experience

#### **Backend Development:**

* **Focus:** Server-side (behind the scenes operations).
* **Technologies:** Python, JavaScript (Node.js), Ruby, PHP, Java, C#.
* **Frameworks:** Django, Express.js, Ruby on Rails, Spring Boot.
* **Role:** Manage databases, server logic, APIs, and application security.
* **Tools:** Web servers (Apache, Nginx), database systems (MySQL, MongoDB), version control (Git).
* **Responsibilities:** Ensuring data integrity, server performance, and secure data handling.

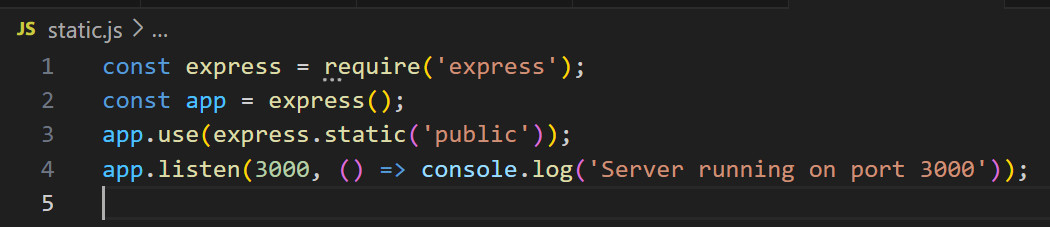
#### **Comparison:**

* **Frontend:** Directly interacts with users, emphasizes aesthetics and usability.
* **Backend:** Handles data processing, application logic, and server-side functionality, emphasizes security and performance.
* **Interaction:** Frontend sends requests to the backend; backend processes these requests and sends data back to the frontend.

# **Static vs Dynamic vs API:**

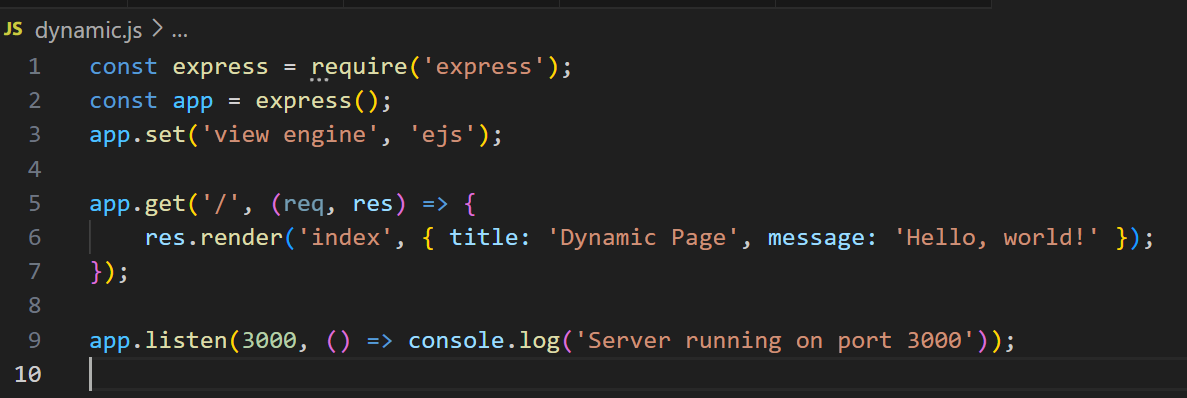
#### **Static:**

* **Definition:** Serves fixed content (HTML, CSS, JS files) that doesn't change per request.
* **Example:** Serving an HTML file with express.static().

.

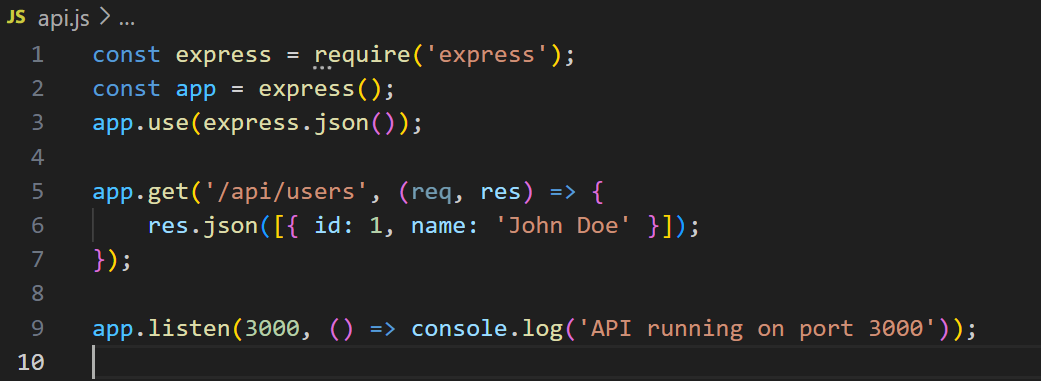
#### **Dynamic:**

* **Definition:** Generates content on the fly based on requests, often using templates and database queries.
* **Example:** Rendering a template with dynamic data using EJS.

.

#### **API:**

* **Definition:** Provides endpoints for client-server communication, often using JSON.
* **Example:** Creating a RESTful API endpoint.



### . **Comparison:**

* **Static:** Fixed content, quick to serve, suitable for simple websites.
* **Dynamic:** Content changes based on request, suitable for personalized content and complex applications.
* **API:** Facilitates data exchange, used for client-server communication in web and mobile apps.

# **Processes, Thread and Thread Pool:**

#### **a) Processes:**

* **Definition:** Independent execution units with their own memory space.
* **Example:** Forking a process using child process.
* **Use:** Running separate tasks in parallel, like CPU-intensive operations.

#### **b) Threads:**

* **Definition:** Lighter execution units within a process, sharing the same memory space.
* **Example:** Node.js primarily uses a single main thread for event-driven, non-blocking I/O operations.

#### **c)Thread Pool:**

* **Definition:** A pool of worker threads that handle asynchronous operations.
* **Example:** Managed by the libuv library, used for file I/O, DNS operations, etc.
* **Use**: Efficiently managing concurrent tasks without blocking the main thread.

### **Comparison**:

* **Processes:** Independent, suitable for isolated tasks, higher memory usage.
* **Threads:** Shared memory within a process, lightweight, Node.js primarily single-threaded.
* **Thread Pool:** Handles async operations, offloads work from the main thread, managed by libuv.

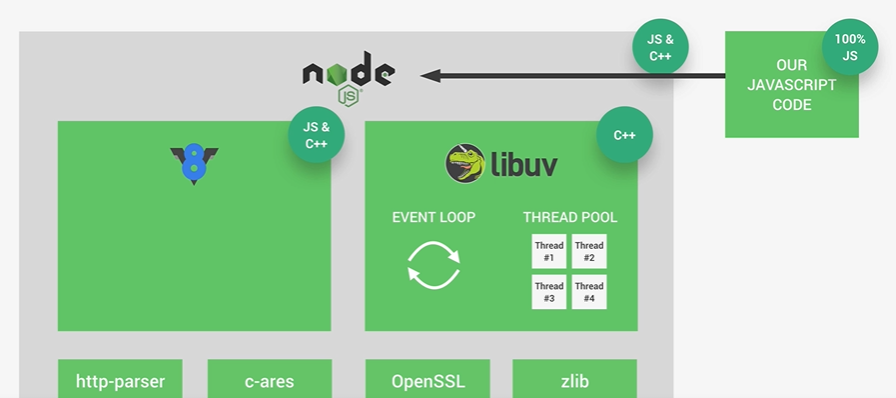
**Section 4: How Node.js Works: A Look Behind the Senses**

# **Section Intro:**

* Node.js is built on the V8 JavaScript engine and uses an event-driven, non-blocking I/O model. Here’s a brief overview of how it works:
* **Event-Driven**: Node.js uses a single-threaded event loop to handle multiple connections concurrently.
* **Non-Blocking I/O**: Asynchronous operations allow Node.js to perform tasks without waiting.
* **Libuv**: Manages the event loop and asynchronous operations.
* **Custom Routing**: Use req.url and req.method to implement routing and handle different requests.

This model allows Node.js to be highly efficient and scalable for I/O-bound applications.

# **Node v8 Libuv and C++:**



* Node.js is built on the V8 JavaScript engin.

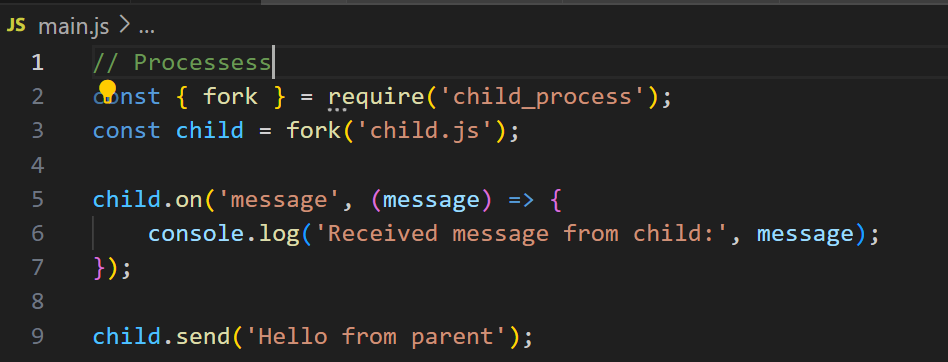
# **Date:26/04/2024**

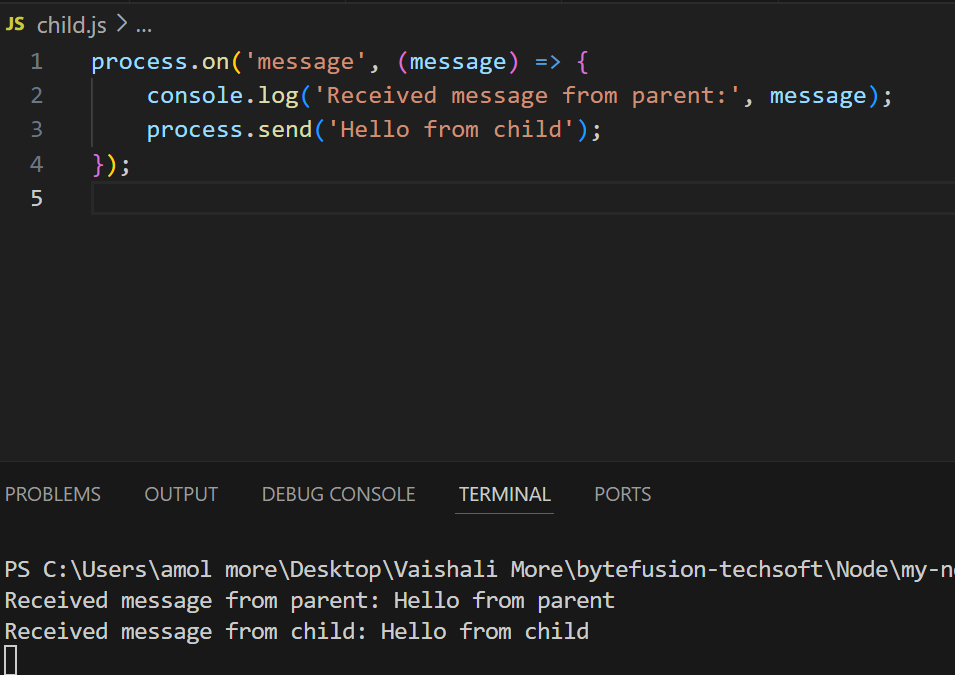
# **Processes, Threads and Thread Pool:**

* Backend web development focuses on the server side of web applications, involving server-side programming, database management, API creation, server management, and security. Key components include

**1**.**Processes:**

* **Definition:** Independent execution units with their own memory space.
* **Example:** Forking a process using child\_process.
* **Use:** Running separate tasks in parallel, like CPU-intensive operations.



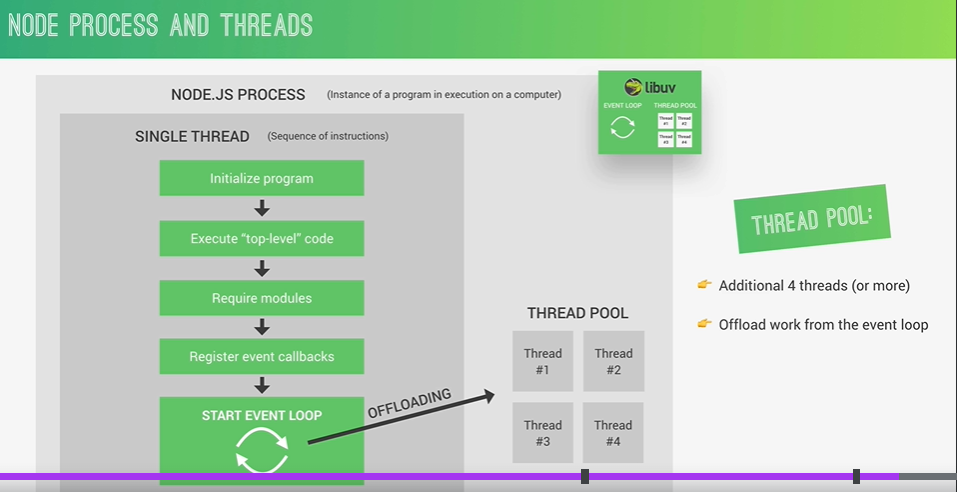


#### **2.Threads**

* **Definition:** Lighter execution units within a process, sharing the same memory space.
* **Example:** Node.js primarily uses a single main thread for event-driven, non-blocking I/O operations.

#### **3.Thread Pool**

* **Definition:** A pool of worker threads that handle asynchronous operations.
* **Example:** Managed by the libuv library, used for file I/O, DNS operations, etc.
* **Use:** Efficiently managing concurrent tasks without blocking the main thread.

****

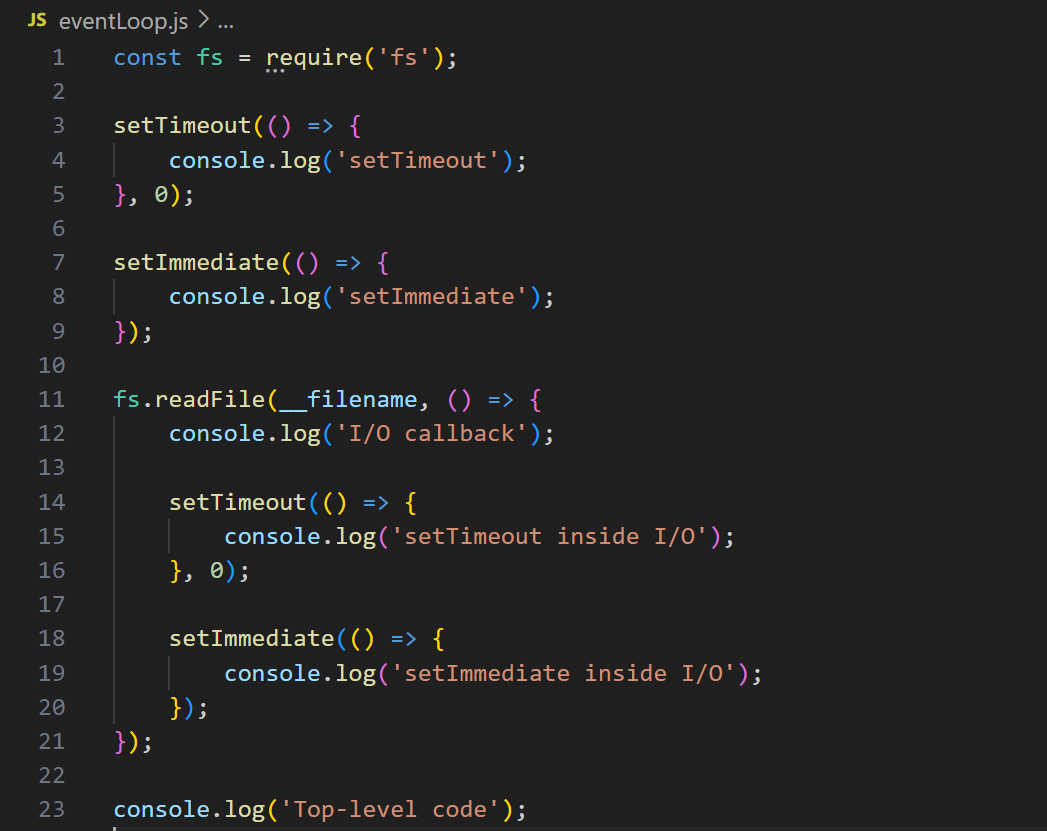
# **The Node.js Event Loop:**

* The event loop is what allows Node.js to perform non-blocking I/O operations — despite the fact that JavaScript is single-threaded — by offloading operations to the system kernel whenever possible.:

#### **a) Event Loop**

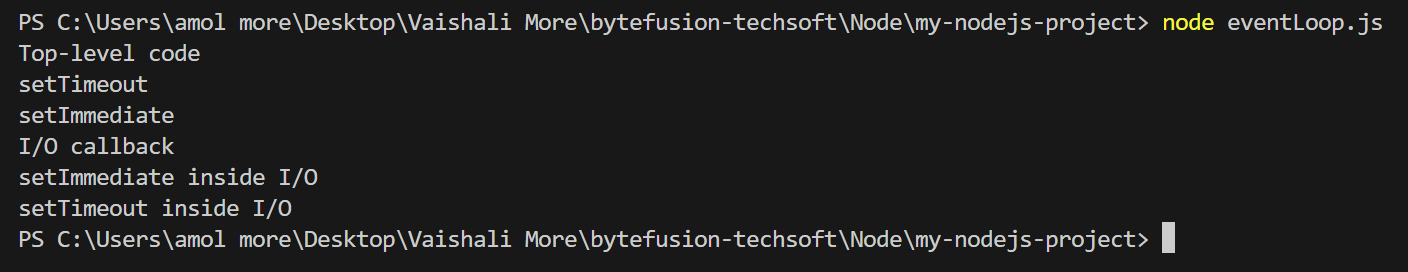
* **Definition:** The mechanism that allows Node.js to perform non-blocking I/O operations by offloading operations to the system kernel whenever possible.
* **Phases:** The event loop has several phases: timers, pending callbacks, idle/prepare, poll, check, and close callbacks.

# **The Event Loop in Practice:**



#### **Execution Order:**

1. **Top-level code:** Executes first.
2. **Timers Phase:** Executes setTimeout callbacks.
3. **Poll Phase:** Processes I/O operations, then runs their callbacks.
4. **Check Phase:** Executes setImmediate callbacks.
5. **Close Callbacks Phase:** Handles closed events.



# **The Events and Event-Driven Architecture:**

# **1.Events**

* **Definition:** Actions or occurrences that happen in the system, which the system can respond to.
* **Core Module:** events module provides the EventEmitter class to handle events.

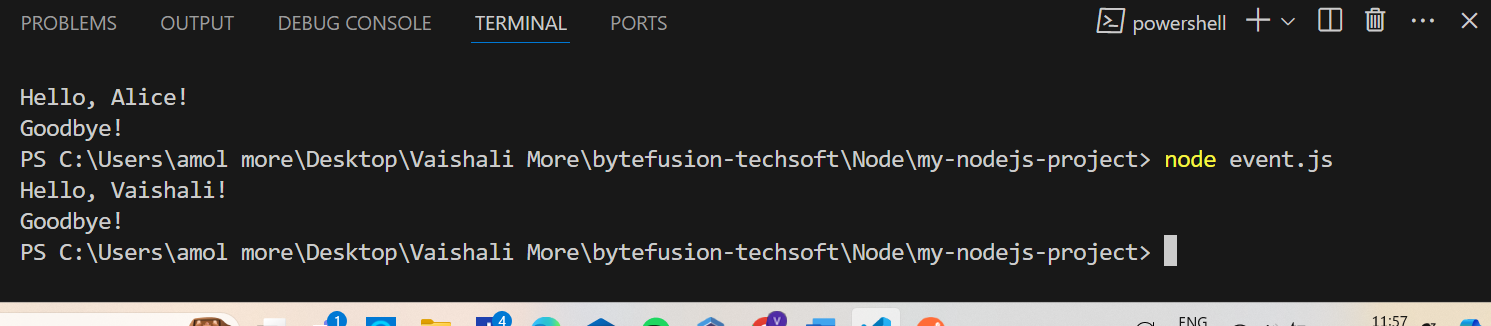


# **The Events Practice:**

# 

**Output:**

This example demonstrates how to create an EventEmitter instance, register event listeners using the .on() method, and trigger events using the .emit() method. This basic event handling is fundamental to understanding more complex event-driven programming in Node.js.

****

# **Introduction to Stream**

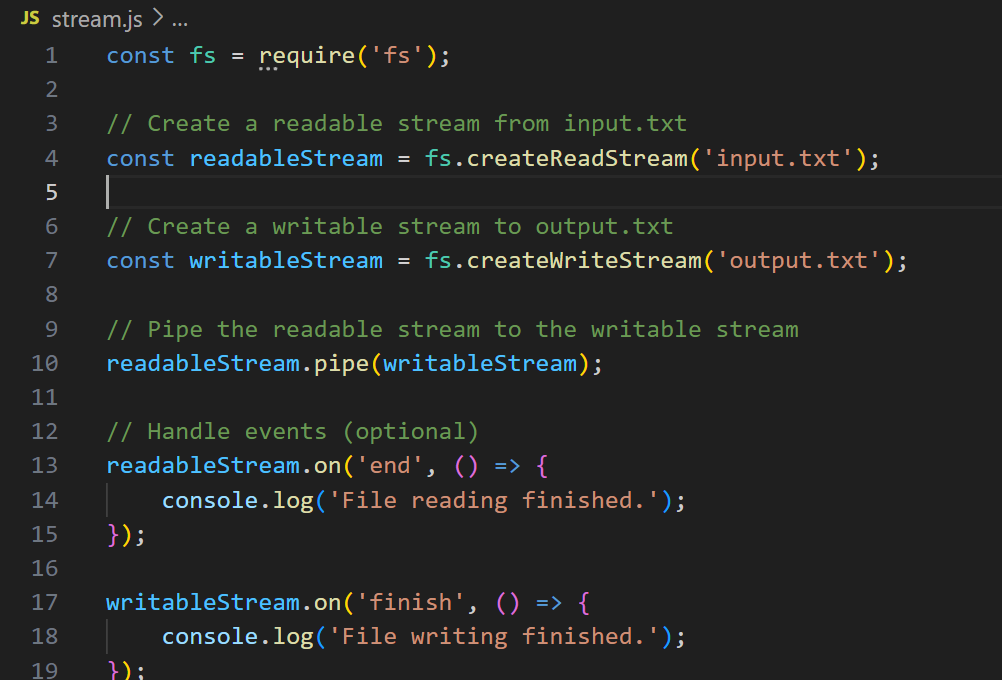
# **Streams**

* **Definition:** Streams are abstract interfaces for working with streaming data in Node.js. They allow reading or writing data piece-by-piece instead of all at once, which is efficient for handling large amounts of data.

#### **Types of Streams**:

1. **Readable Streams:** For reading data (e.g., fs.createReadStream()).
2. **Writable Streams:** For writing data (e.g., fs.createWriteStream()).
3. **Duplex Streams:** For both reading and writing (e.g., TCP sockets).
4. **Transform Streams:** For modifying or transforming data while reading and writing (e.g., zlib.createGzip()).

# **Stream in Practice**

****

# **How Requiring Modules Really Works:**

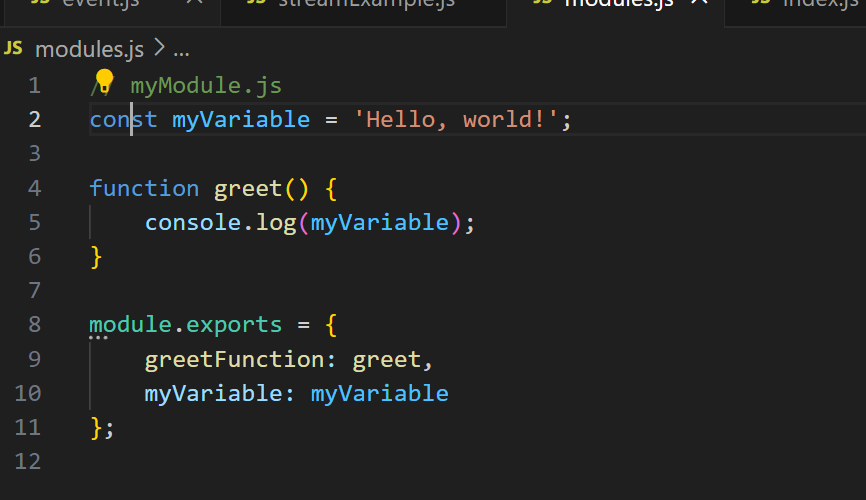
#### **a) Basics of Module Loading**

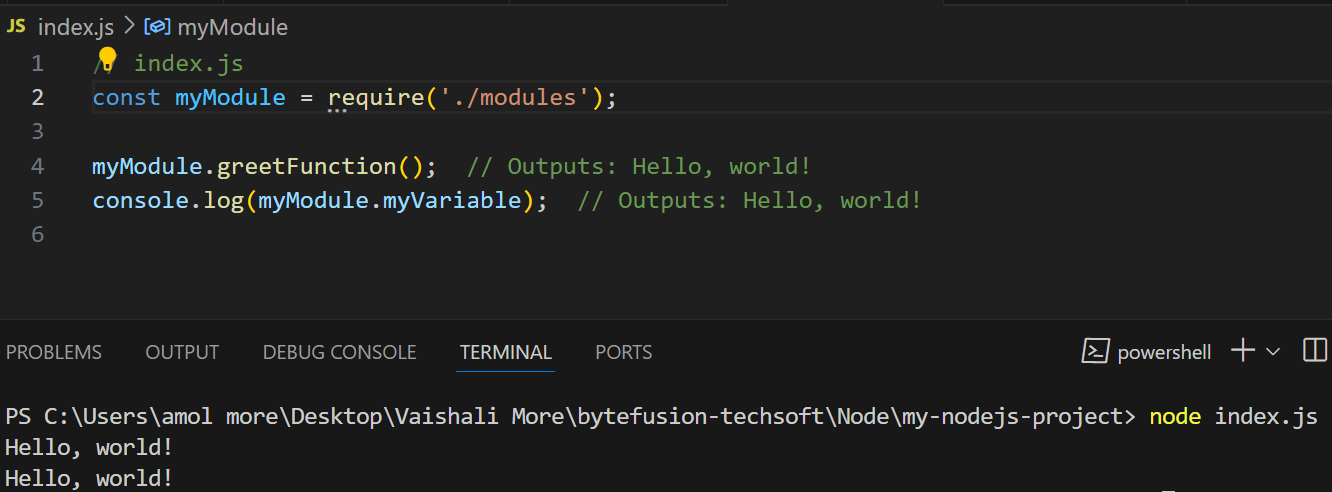
* **Definition:** Node.js uses the CommonJS module system, where each file is treated as a module.
* **Require Function:** require() is a built-in function in Node.js used to import modules.

#### **b) Steps Involved**

1. **Module Identification:**
   * When require('module-name') is called, Node.js tries to locate the module-name.
2. **File Extension Resolution:**
   * If the module path doesn't include a file extension (like .js, .json, .node), Node.js appends .js, .json, or tries to load as a native .node extension.

# **Requiring Modules in Practice:**





**Section 5: (Optional)Asynchronous JavaScript: Promise and Async/Await**

# **Section Intro:**

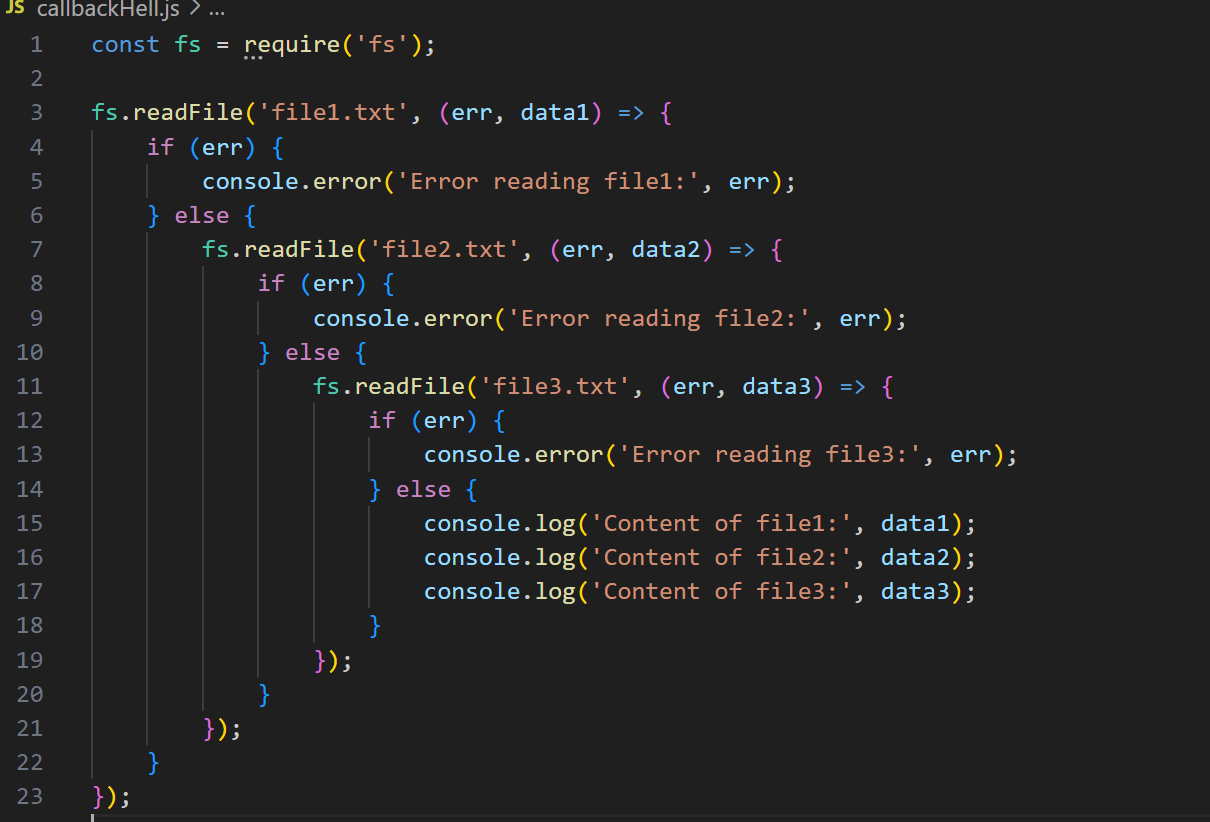
 **Promise:** Handles asynchronous operations, providing better control flow with .**then**() and .**catch**() for success and error handling.

 **Async/Await:** Simplifies asynchronous code further by using **async** functions and **await** keywords to wait for promises to resolve, making code more readable and maintainable.

# **The Problem with Callback: Callback Hell:**

#### **Callback Hell**

* **Issue:** Occurs when multiple nested callbacks are used, leading to code that is hard to read, debug, and maintain.
* **Causes:** Deeply nested callbacks due to sequential or dependent asynchronous operations.

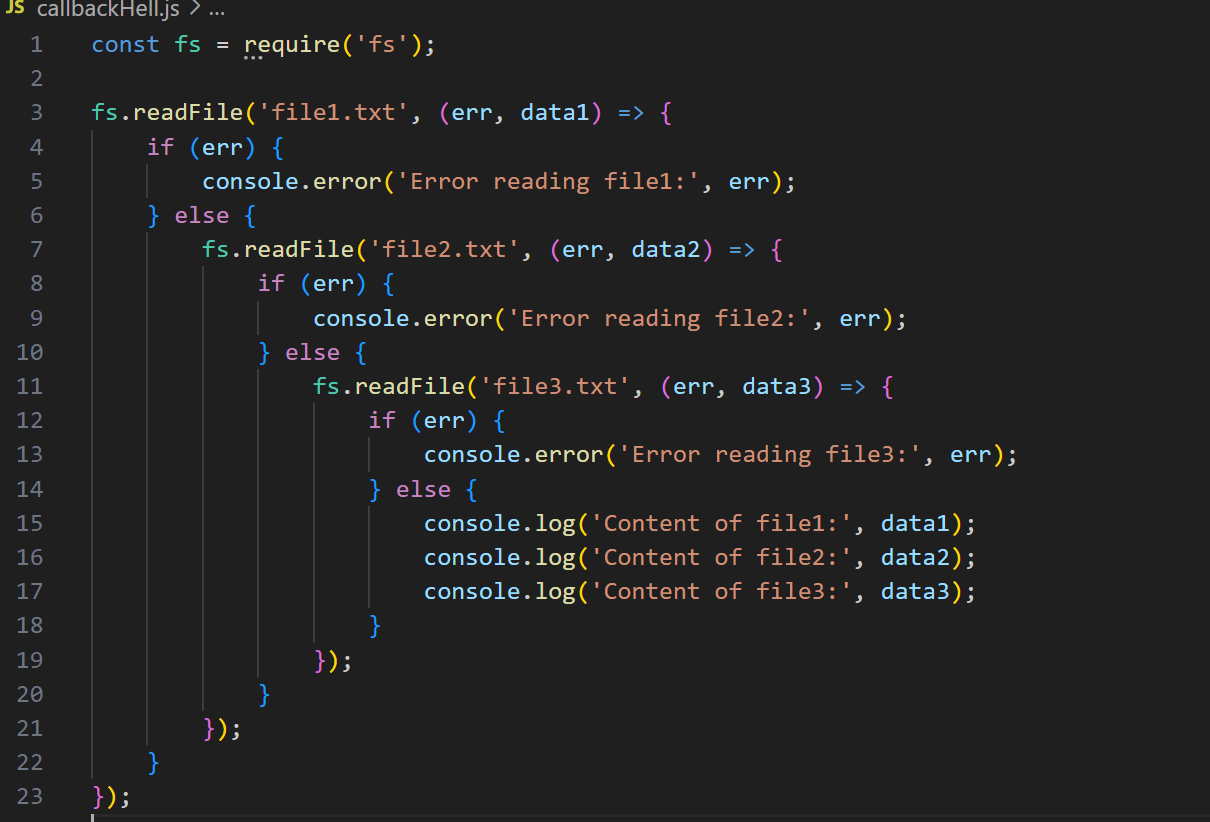


# **From Callback Hell to Promises:**

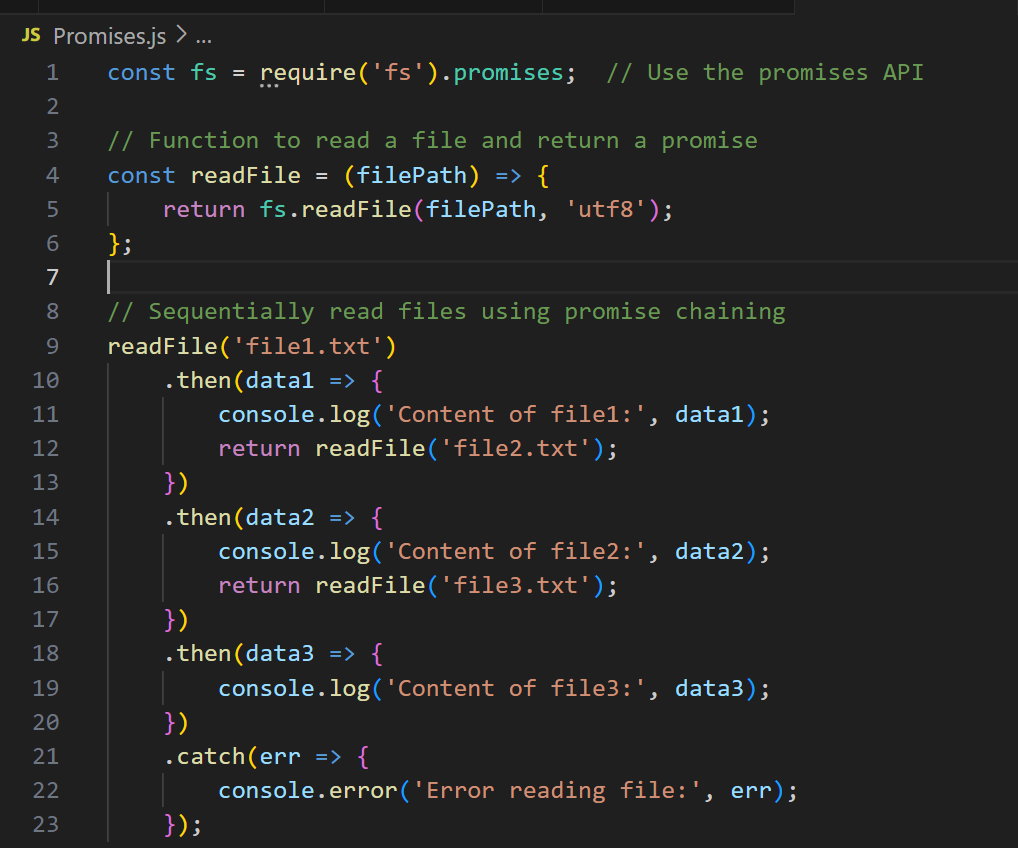
#### **Callback Hell to Promises**

Using Promises can make asynchronous code more manageable and readable by avoiding deeply nested callbacks. Promises allow chaining of asynchronous operations and centralized error handling.

**Callback Hell:**

****

**Promises:**

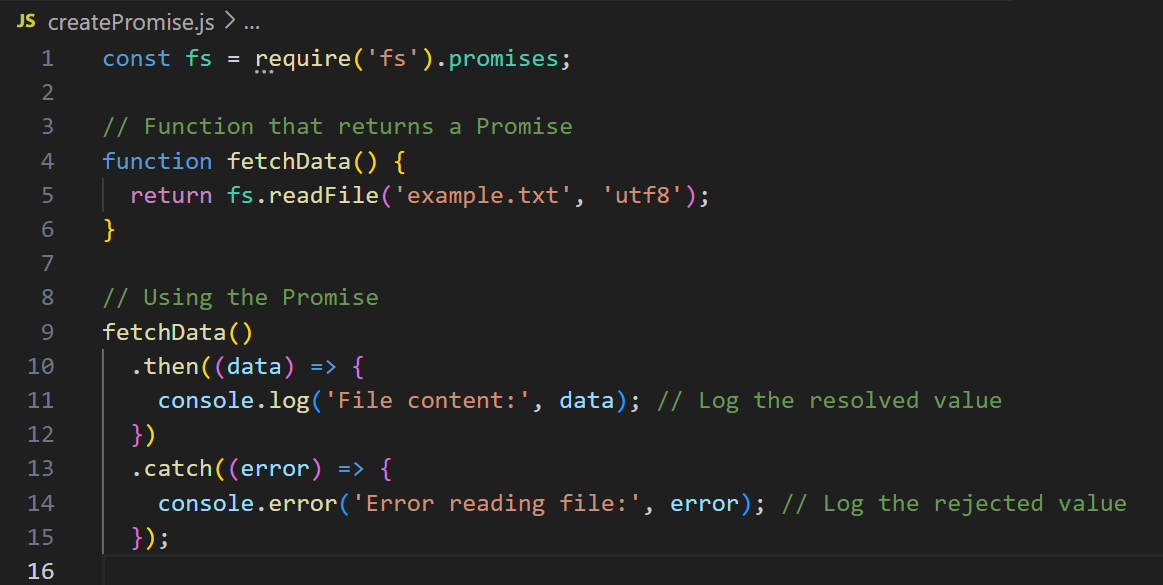
****

### **Benefits of Using Promises**

* **Improved Readability:** Flat, linear structure compared to nested callbacks.
* **Modularity:** Easier to isolate and reuse asynchronous functions.
* **Centralized Error Handling:** Simplifies error management with a single .catch().

Using Promises transforms callback hell into more readable and maintainable code, enhancing the structure and handling of asynchronous operations in Node.js.

# **Building Promises:**



### **Benefits of Using Promises**

* **Improved Readability:** Flat, linear structure compared to nested callbacks.
* **Modularity:** Easier to isolate and reuse asynchronous functions.
* **Centralized Error Handling:** Simplifies error management with a single .catch().

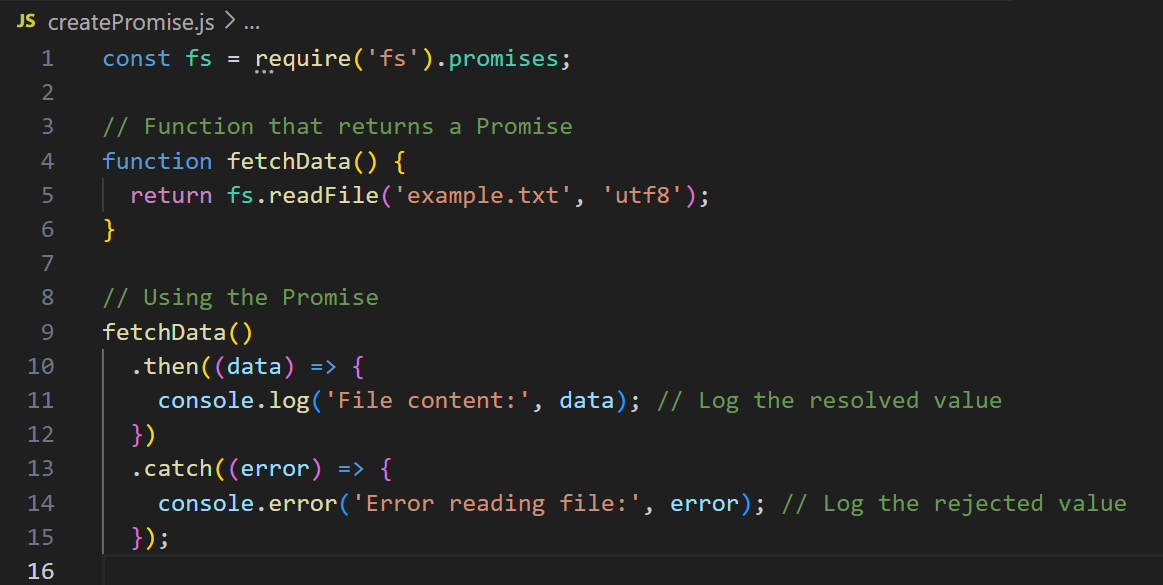
# **Consuming Promises with Async/Await:**

#### **Key Points**

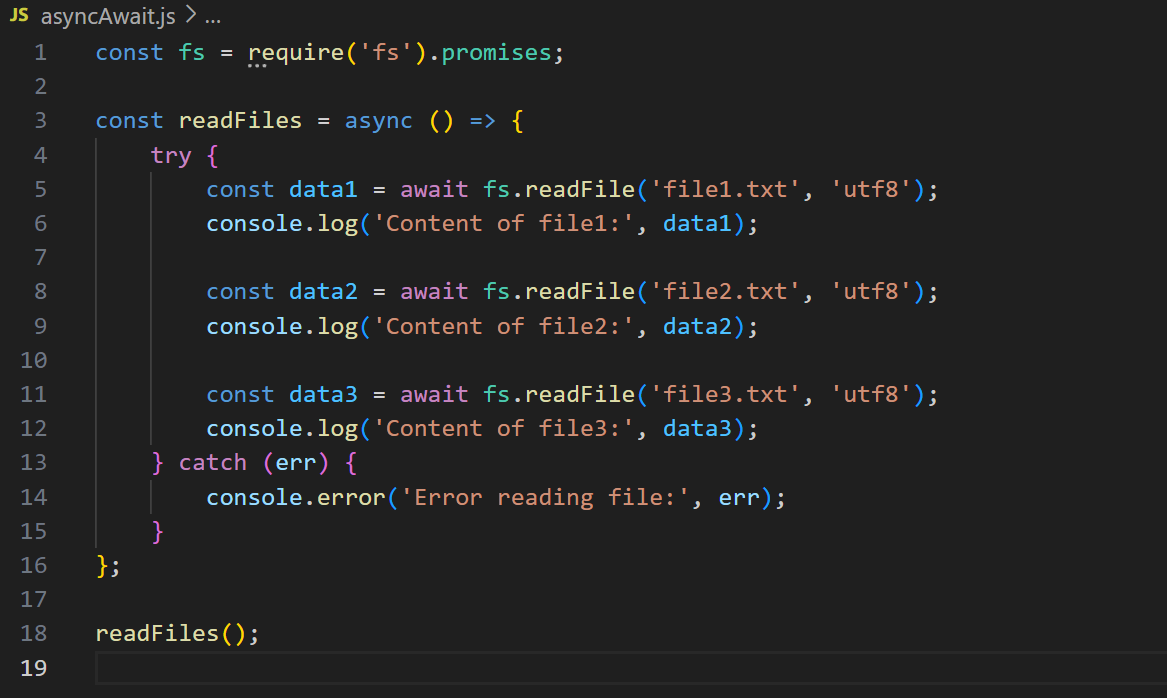
* **Async Functions:** Declared using the async keyword, they return a promise.
* **Await Keyword:** Used within async functions to pause execution until a promise is resolved or rejected, making the code appear synchronous.
* **Error Handling:** Use try...catch blocks to handle errors within async functions.

**Example: Using Async/Await:**

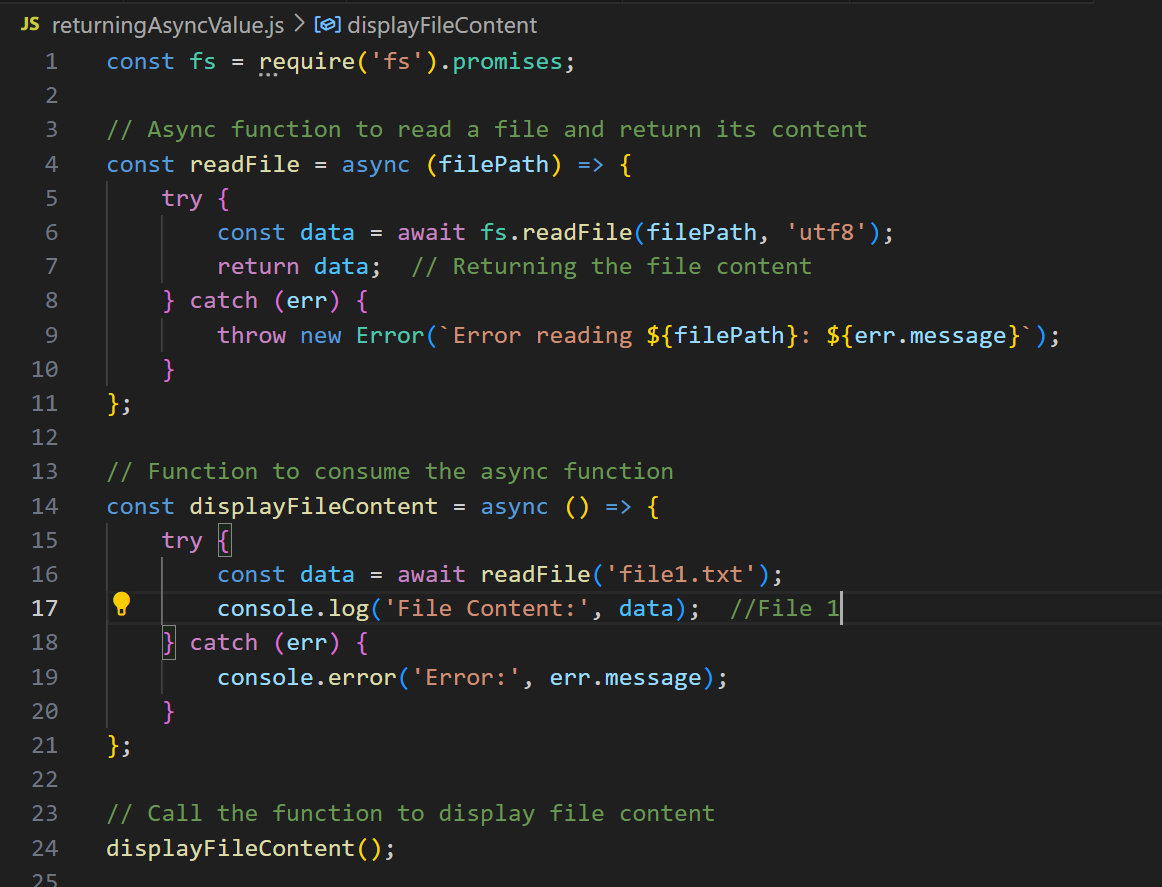
**Using Promises:**

****

**Using Async/Await:**

****

# **Returning Values from Async Function:**



 **Async Function:**

* readFile is an async function that reads a file and returns its content as a string. If there is an error, it throws an error with a descriptive message.

 **Consuming the Async Function:**

* displayFileContent is another async function that calls readFile and uses await to get the content of the file.
* The content is then logged to the console. If an error occurs, it is caught and logged.

 **Executing the Function:**

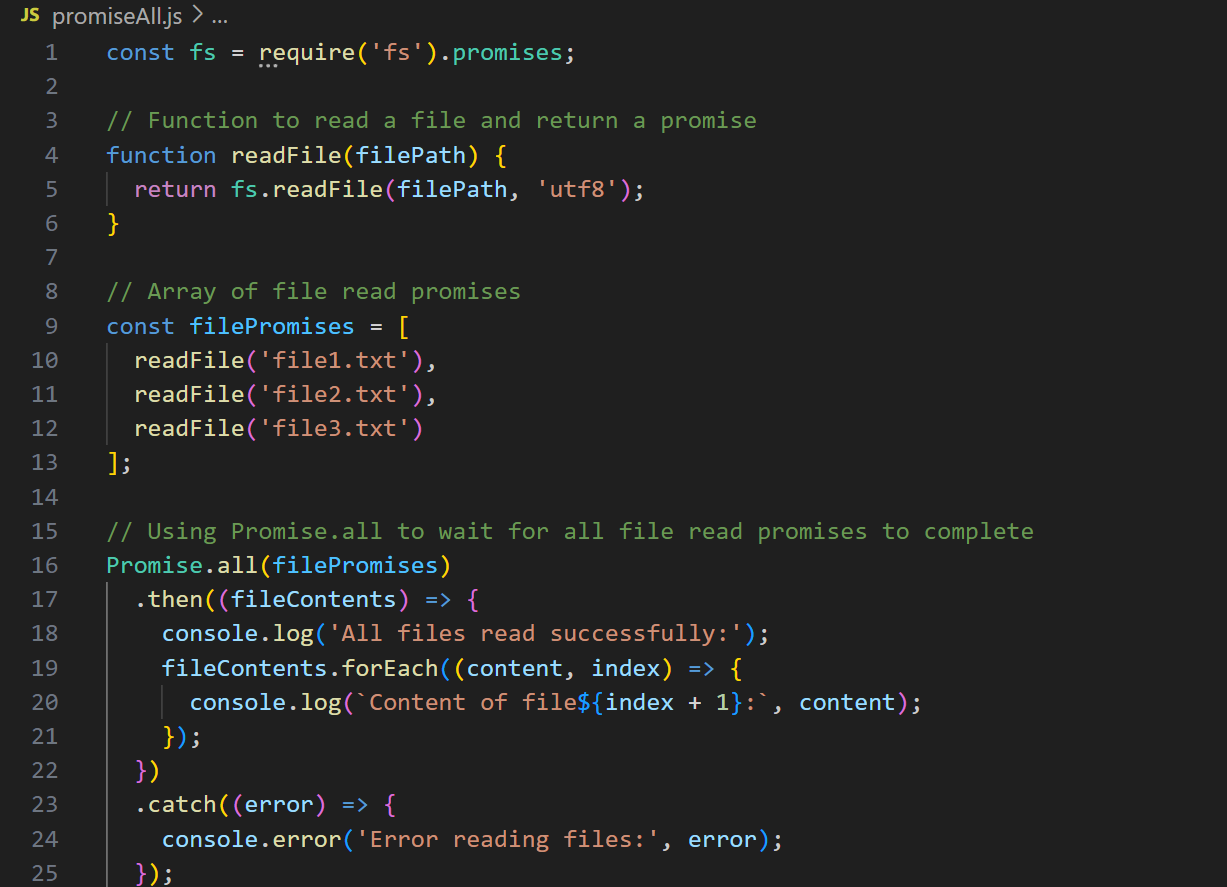
* displayFileContent is called to execute the asynchronous operations.

# **Waiting for Multiple Promises Simultaneously:**

When you need to wait for multiple asynchronous operations to complete, you can use Promise.all. This method takes an array of promises and returns a single promise that resolves when all the promises in the array have resolved, or rejects if any of the promises reject.

### **Key Points**

* **Promise.all**: Takes an array of promises and returns a single promise.
* **Resolves**: When all promises in the array have resolved.
* **Rejects**: If any promise in the array rejects.



**Section 6: Express: Let’s Start Building the Natours API**

# **Section Intro:**

* This introduction sets the stage for developing a full-fledged API, illustrating the ease of using Express for building scalable and maintainable web services.

# **What is Express?**

**Express** is a minimal and flexible web application framework for Node.js. It provides a robust set of features for building web and mobile applications. Express facilitates the development of APIs and web servers by simplifying the process of handling HTTP requests and responses.

#### **Key Features**

* **Middleware:** Functions that execute during the request-response cycle. They can perform tasks like parsing request bodies, handling cookies, and providing authentication.
* **Routing:** Simplifies the definition of route handlers for various HTTP methods and URL patterns.
* **Extensibility:** A vast ecosystem of plugins and middleware to extend functionality.
* **Performance:** Lightweight and fast, making it suitable for scalable applications.

# **Installing Postman:**

* Postman is a popular tool for testing APIs. Here are the short steps to install Postman:

**1.On Windows**:

1. **Download:**
   * Visit the Postman website.
   * Click on the "Download for Windows" button.
2. **Install:**
   * Once the download is complete, open the installer.
   * Follow the installation prompts to install Postman on your system.

#### **2.On macOS**:

1. **Download:**
   * Visit the Postman website.
   * Click on the "Download for Mac" button.
2. **Install:**
   * Once the download is complete, open the .dmg file.
   * Drag the Postman app to your Applications folder.

#### **3.On Linux:**

1. **Download:**
   * Visit the Postman website.
   * Download the appropriate version for your Linux distribution (e.g., .tar.gz file for most distributions).
2. **Install**:
   * tar -xzf Postman-linux-x64.tar.gz
   * sudo mv Postman /opt/Postman
   * sudo ln -s /opt/Postman/Postman /usr/bin/postman

#### **Post-Installation**:

1. **Launch Postman:**
   * Open Postman from the Start menu (Windows), Applications folder (macOS), or by typing postman in the terminal (Linux).
2. **Sign In or Create an Account:**
   * You can sign in with an existing account or create a new one to sync your work across devices.

# **Setting up Express and Basic Routing:**

* **Express** is a minimal and flexible Node.js web application framework that provides robust features for building web and mobile applications.

#### **Key Points**

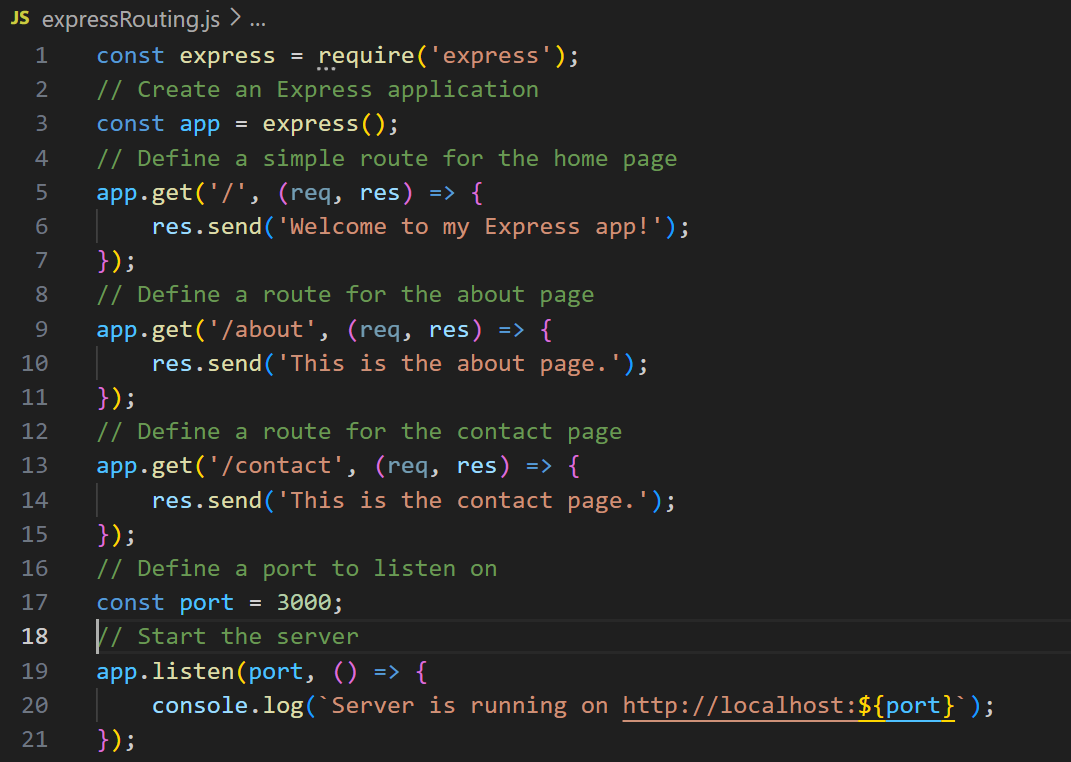
* **Installation**: Install Express using npm.
* **Creating a Server**: Use Express to create an HTTP server.
* **Routing**: Define routes to handle different HTTP requests.

### **Installation**

To install Express, use npm (Node Package Manager**):**

**npm install express**

**Basic Setup and Routing:**

****

# **API’s and RESTful API Design:**