INDIRA GANDHI NATIONAL TRIBAL UNIVERSITY

Lalpur, Amarkantak, Distt. Anuppur (M.P.) – 484887

(A Central University established by an Act of Parliament)

FACULTY OF TECHNICAL, VOCATIONAL EDUCATION
AND SKILL TRAINING

NODE JS WITH EXPRESS JS ASSIGNMENT

COURSE	**: M.Voc in software development**
SEMESTER	**: 2nd**
PAPER-CODE	**: MSD-201**
DEPARTMENT	**: Department of Vocational Education**

Submitted to:

Kamlesh Pandey

(Assistant professor,

Department of

Vocational Education)

Submitted by:

Name- Shakti Barnwal

Enrollment No-2301233005

INDEX

Q1. Explain any 10 advantages of Node JS through example.

Node.js is a cross-platform runtime environment and library for running JavaScript applications outside the client's browser. Node.js is a powerful tool for web development, offering performance benefits and flexibility for building server-side applications. It is open source and free to use.

Advantages of Node JS:

1. **Cost Effective Solution:**
   - Node.js advantage is the use of a single programming language (JavaScript) for front-end and back-end development. Among Node.js's biggest advantages is its ability to eliminate remuneration requirements by allowing two resource teams to work together.
   - Also, Node.js require fewer files and less code. It helps save a lot of time, money, and energy in startup product development.

2. **Cross-Platform Compatibility:**
   - Node.js's cross-platform compatibility exemplifies its remarkable versatility, enabling the creation of applications that run seamlessly on Windows, macOS, and Linux, ensuring a broad reach and smooth user experience across diverse operating systems.
   - Node.js eliminates the need for platform-specific code and reduces the complexity of managing multiple codebases. Node.js allows for development across platforms of Electron and NW.Js for creating interactive web applications on any platform.

3. **Front-end and Back-end Development:**
   - Beyond back-end development, Node.js thrives in the front-end realm, empowering developers to build full-stack applications using frameworks like Express.js and Socket.IO.
   - This unified approach streamlines development and diminishes the necessity for separate front-end and back-end teams, ultimately saving both time and resources. This might be the biggest benefit of using Node js. Fast development time, ease of use, and the ability to scale with increasing traffic.

4. **Flexible:**
    - Flexibility is another advantage of Node.js. At the point when you roll out an improvement in Node.js, just that node is impacted. Where other run-time conditions or structures might expect you to make changes as far as possible back to the core programming, it requires nothing except a change to the node.
    - Furthermore, the best part is that when you consolidate JSON with Node Js, you can undoubtedly trade data between client servers and the web server.

5. **Scalability:**
    - Scalability is built into the core of Node.js. It is one of advantages of Node js for startups planning to grow over time. App-based startups choose to develop lightweight and fast systems with good real-time responses that can be scaled up later and add more modules to the existing ones easily. Node's apps support both vertical and horizontal scaling.
    - What Node is good for is its compatibility with microservices architecture, which is beneficial for projects that will scale and grow in the future. Also, it is possible to create a separate microservice for any functionality and then scale it separately from all other parts.

6. **IoT Mobile Apps:**
    - IoT is the future of technology and the 5th industrial revolution. Companies require unified IoT software to manage related devices, sensors, and equipment with IoT's interface.
    - Programmers prefer using Node Js in IoT mobile apps because it integrates with IoT protocols, provides higher performance, data protection, and secured authentication, and makes it easier to build API for these apps.

7. **Efficient caching:**

    - In a debate over the pros and cons of Node.js, caching always comes up as a key Node.js benefit. It has a powerful ability to cache data. When requests are made to the app, they are cached in-app memory. Consequently, when requests cycle through execution and re-execution, the nodes are still able to run efficiently and not be bogged down by historical data.

- The developers don’t have to re-execute the codes as caching allows applications to load the web pages faster and responds more swiftly to the user.

8. **Speed and performance:**
   - Its non-blocking, input-output operations make the environment one of the speediest options available. Code runs quickly, and that enhances the entire run-time environment. This is largely due to its sectioned-off system. But it also has to do with the fact that it runs on Google’s V8 JavaScript engine. Its apps are more likely to be programmed end-to-end in Javascript, and that plug and play interoperability contributes to speed and performance.

9. **Asynchronous Programming:**
   - Node.js relies on an event-driven, non-blocking I/O mechanism, programmers can create extremely responsive and effective programs. Because of this, it is perfect for creating real-time applications like chat programs and gaming platforms. Node.js is best for creating asynchronous APIs, servers, and real-time applications.

10. **Single-Threaded:**
    - Node.js is single-threaded, it can process numerous requests concurrently without starting a new thread for each one. Compared to other programming languages that need several threads to handle numerous requests, this makes it more effective and simpler to manage.
    - Node.js is single-threaded as it has a single main event loop that processes JavaScript operations and handles all I/O. However, Node.js provides us with additional features that, if properly used, can give the advantages that multithreading has.

Q2. Explain disadvantages of traditional web server model.

The traditional web server model, also known as the client-server architecture. A client server architecture is a system that hosts, provides, and manages most of the resources and services that the client requests. This approach, also known as the networking computing model or client server network, involves the delivery of all requests and services across a network.

Disadvantages:

The client-server architecture, has several disadvantages-

1. **Centralized system:** The traditional web server model is a centralized system, where all data is stored in a single location, making it vulnerable to single points of failure and potential data loss. In other words we can say, It increased security risks, as a vulnerability in one component can potentially compromise the entire system.

2. **High Cost:** The traditional web server model can be cost-efficient. it requires regular maintenance and updates to ensure the server remains secure and functional. In client-server networks, the cost of setting up and maintaining the server is typically higher than the cost of running the network.

3. **Complex:** In the event of data loss or corruption, data recovery can be a complex and time-consuming process. If, main server gets halt then entire system will be failed.

4. **Capacity limitations:** The capacity of the client and server can be changed separately, but this can be a complex process and may require significant resources.

5. **Network complexity:** The traditional web server model can introduce network complexity, as multiple clients may be connected to the same server, which can lead to network congestion and potential performance bottlenecks.

6. **Dependency on network stability:** The traditional web server model relies on a stable network connection, which can be unreliable and may cause issues if the connection is lost.

7. **Performance bottlenecks:** The traditional web server model can experience performance bottlenecks, particularly if the server is handling a large number of requests simultaneously.

8. **Require dedicated server infrastructure:** The traditional web server model requires a dedicated server infrastructure, which can be expensive and resource-intensive.

9. **Network traffic congestion:** The main disadvantage of a client-server model is the danger of a system overload owing to a lack of resources to service all of the clients. If too many different clients try to connect to the shared network at the same time, the connection may fail or slow down. Additionally, if the internet connection is down, any website or client in the world will be unable to access the information. Large businesses may be at risk if they are unable to get important information.

Q3. Explain the process of traditional web server model through example.

The traditional web server model is a client-server architecture where a client (usually a web browser) requests a resource from a server. The traditional web server model is a common architecture used to design and deploy web applications. In this model, each request is handled by a dedicated thread from a thread pool. If no thread is available in the thread pool at any point in time, the request waits until the next available thread becomes free. This approach is often used in web servers like **Apache and IIS**.

A thread is freed only when the request is processed, a response is returned, and the thread is returned back to the thread pool. This makes the traditional web server model synchronous and blocking.

Process of Traditional Web Server Model:

Step 1: Client Request A user types a URL (Uniform Resource Locator) in their web browser, for example, https://www.youtube.com. The browser breaks down the URL into its components: protocol (https), domain name (www.youtube.com), and path (/).

Step 2: DNS Resolution The browser sends a request to a Domain Name System (DNS) server to resolve the domain name (www.youtube.com) to an IP address. The DNS server returns the IP address associated with the domain name.

Step 3: Connection Establishment The browser establishes a connection with the web server at the resolved IP address using the Hypertext Transfer Protocol (HTTP). The HTTP request is sent to the web server, which is running on a specific port (usually 80 for HTTP).

Step 4: Web Server Processing The web server receives the HTTP request and processes it. The web server checks if the requested resource exists on the server and is accessible. If it does, the web server retrieves the resource and prepares it for transmission.

Step 5: Resource Retrieval The web server retrieves the requested resource, which can be a file (e.g., HTML, CSS, JavaScript, or image), a database query result, or a dynamically generated response.

Step 6: Response Generation The web server generates a response to the client, which includes the requested resource and any additional information (e.g., headers, metadata).

Step 7: Response Transmission The web server sends the response back to the client over the established connection.

Step 8: Client Rendering The client receives the response and renders the requested resource. For example, if the requested resource is an HTML file, the browser parses the HTML and displays the content to the user.

Step 9: Connection Closure The client and server close the connection, and the process is complete.

This traditional web server model is a simple example of the client-server architecture, where the client (web browser) requests a resource from the server, and the server responds with the requested resource.

Q4. Explain Node JS process model.

The Node.js process model differs from traditional web servers in that Node.js runs in a single process with requests being processed on a single thread. One advantage of this is that Node.js requires far fewer resources. Contrary to the traditional web server model, NodeJS uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

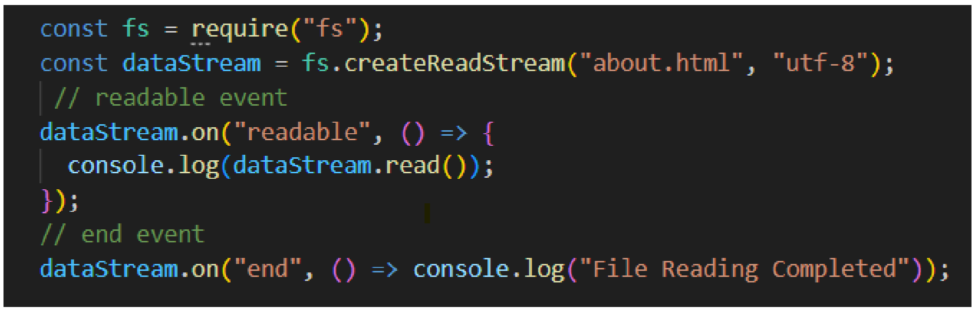
The NodeJS process model can be explained with three architectural features of NodeJS.

1. Single-threaded event loop
2. Non-Blocking I/O Model
3. Event-driven and Asynchronous by default

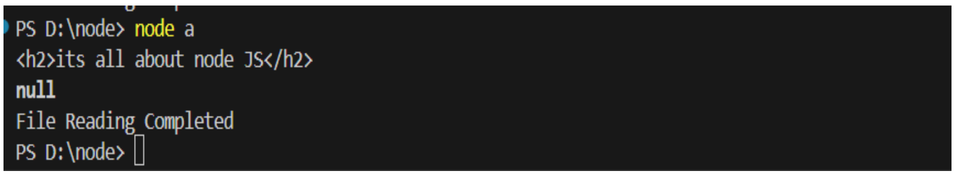
Single Threaded Event Loop:

- NodeJS runs on a single-threaded environment which means each user request processes on a single thread only. Events are a crucial paradigm of the NodeJS process. Events are actions that instruct the runtime what and when something needs to be completed. Event Emitters are response object instances that can be subscribed to and acted upon to perform operations. Event Emitters emit events based on certain predefined events accepting a callback.
- NodeJS has two types of threads: one Event loop also referred to as the main thread, and the k Workers also referred to as the background thread. When a new user request comes in, it is placed in an event queue. Every request consists of a synchronous and asynchronous part. The synchronous part of the request is handled on the main thread while the asynchronous part is handled in the background via the k Workers/ background threads.

Example: Listening to events and asynchronous operations-



```
const fs = require("fs");
const dataStream = fs.createReadStream("about.html", "utf-8");
 // readable event
dataStream.on("readable", () => {
  console.log(dataStream.read());
});
// end event
dataStream.on("end", () => console.log("File Reading Completed"));
```

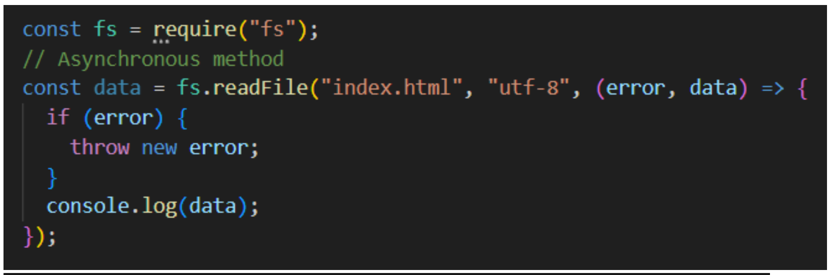


```
PS D:\node> node a
<h2>its all about node JS</h2>
null
File Reading Completed
PS D:\node> 
```

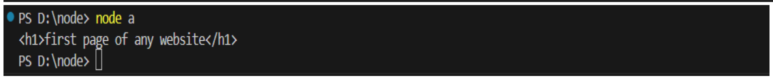
Non-Blocking I/O Model:

- Blocking codes or operations are the ones that need to be completed entirely before moving on to another operation. Non-blocking codes are asynchronous and accept callback functions to operate.
- As mentioned, every request has a synchronous and asynchronous part. The main thread of NodeJS does not keep waiting for the background thread to complete the asynchronous I/O operations. The main thread keeps switching between other requests to process their synchronous part while the background thread process the asynchronous part.

Example: The asynchronous method always uses callbacks and events to resolve its operation-



```
const fs = require("fs");
// Asynchronous method
const data = fs.readFile("index.html", "utf-8", (error, data) => {
  if (error) {
    throw new error;
  }
  console.log(data);
});
```



```
PS D:\node> node a
<h1>first page of any website</h1>
PS D:\node> 
```

Event-Driven and Asynchronous by Default:

- Once the execution of the background thread is complete, the background thread emits events to notify the main thread. Callback functions are associated with asynchronous processes. If the main thread is not free, the request waits for the main thread to be free and then takes up the callback request for further execution.
- To provide concurrency, I/O events and callbacks, and other time-consuming operations are asynchronous by default. Node architecture uses libuv, which is a C library built specifically for NodeJS for handling most asynchronous I/O operations.

Q5. Explain advantages of Node JS process model over the traditional web server.

Node.js offers several advantages over the traditional web server model, primarily due to its process model.

Lightweight and Efficient:

- Node.js uses an event-driven, non-blocking I/O model, which makes it lightweight and efficient.
- Unlike the traditional web server model, where each user request is assigned to a dedicated thread (resulting in thread pools), Node.js processes requests using a single-threaded event loop.
- This single-threaded approach reduces resource consumption and allows Node.js to handle a large number of concurrent connections without creating additional threads.

Event-Driven and Asynchronous:

- Node.js leverages an event-driven architecture.
- Events are actions that instruct the runtime when and what needs to be completed.
- The event loop is responsible for executing code, processing events, and executing queued sub-tasks.
- When a new user request arrives, it is placed in an event queue.
- Node.js processes the synchronous part of the request on the main thread and the asynchronous part in the background via worker threads.
- Despite the use of multiple threads internally, Node.js is considered single-threaded because all requests are received on a single thread.

Non-Blocking I/O Model:

- Blocking operations wait for completion before moving to the next operation.
- Node.js, on the other hand, performs non-blocking I/O operations.
- Asynchronous code execution allows Node.js to handle multiple requests concurrently without waiting for each operation to complete.
- Callback functions are used to manage asynchronous operations effectively.

Resource Utilization:

- Traditional web servers create a separate thread (often around 2MB in size) for each incoming connection.
- In contrast, Node.js allocates a small heap per connection, resulting in efficient memory usage.
- Node.js processes run in a single thread, requiring fewer resources compared to other platforms.

Developer Productivity:

- Node.js simplifies development by removing silos between frontend and backend developers.
- It provides an ecosystem of over 140,000 npm packages, making it easy to handle dependencies.
- The lightweight and fast nature of Node.js enhances developer productivity.

Q6. Explain the installation process of Node JS in your system.

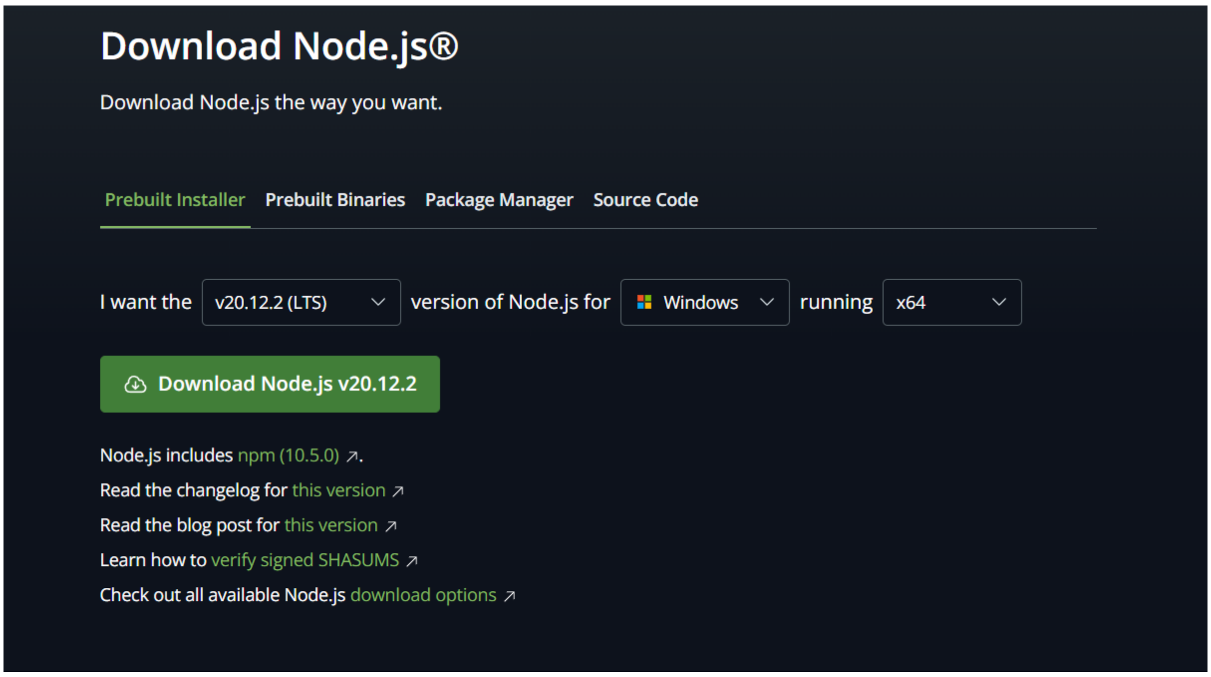
As we know that Node JS is free, open-source framework of java script that upgrades its functionality. Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. Node.js has grown into a vital element for server-side programming

Node.js installation: For Node.js installation you have to follow below given steps-

Step 1: Download Node.js from its official site: **https://nodejs.org/en/download/**.

Step 2: In the downloads page, you will see various versions of Node. All you need to do is, click on the box, suitable to your system configurations.

Select your operating system, On the homepage of the Node.js website, you'll see buttons for different operating systems such as Windows, macOS, and Linux. Click on the button corresponding to your operating system.



Download Node.js®

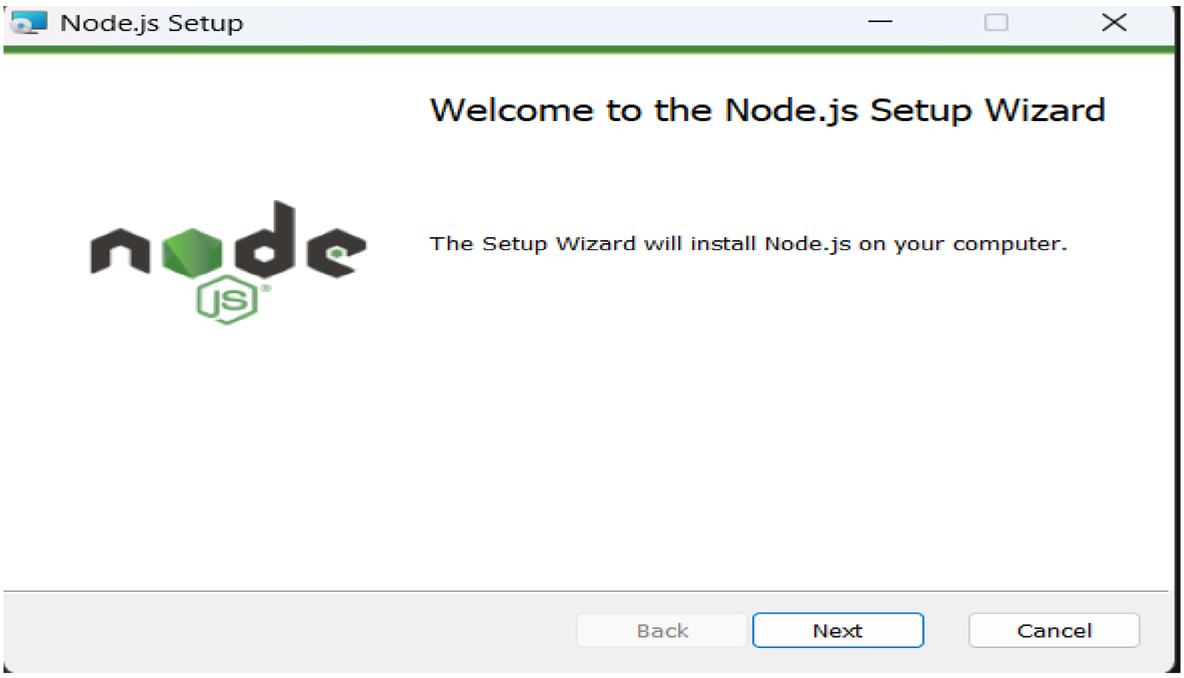
Download Node.js the way you want.

I want the v20.12.2 (LTS) version of Node.js for Windows running x64

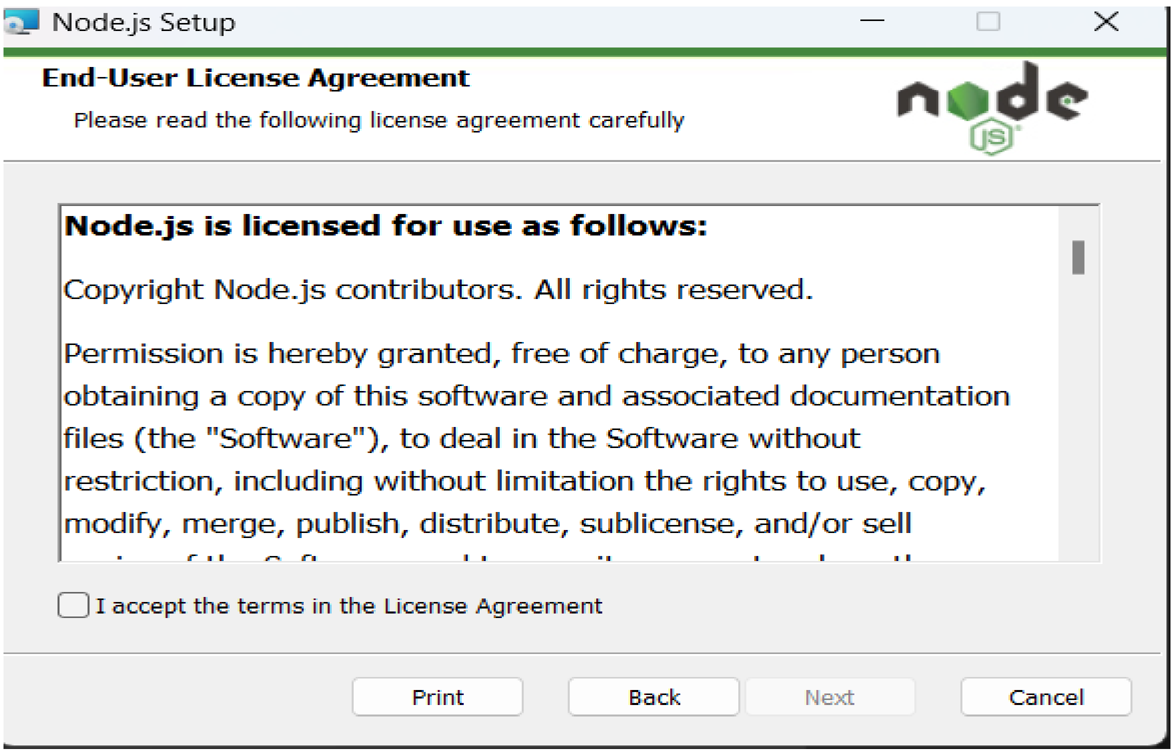
Node.js includes npm (10.5.0) ↗.
Read the changelog for this version ↗
Read the blog post for this version ↗
Learn how to verify signed SHASUMS ↗
Check out all available Node.js download options ↗

Step 3: Once you have successfully downloaded the software, go to the downloaded file, and double-click on it.

Step 4: As soon as you click on the file, an installation wizard will come up. Select 'Next' and move further with the installation.


Node.js Setup
Welcome to the Node.js Setup Wizard
The Setup Wizard will install Node.js on your computer.
Back
Next
Cancel

Step 5: Checkmark on the "I accept" checkbox and click on the 'Next' button.

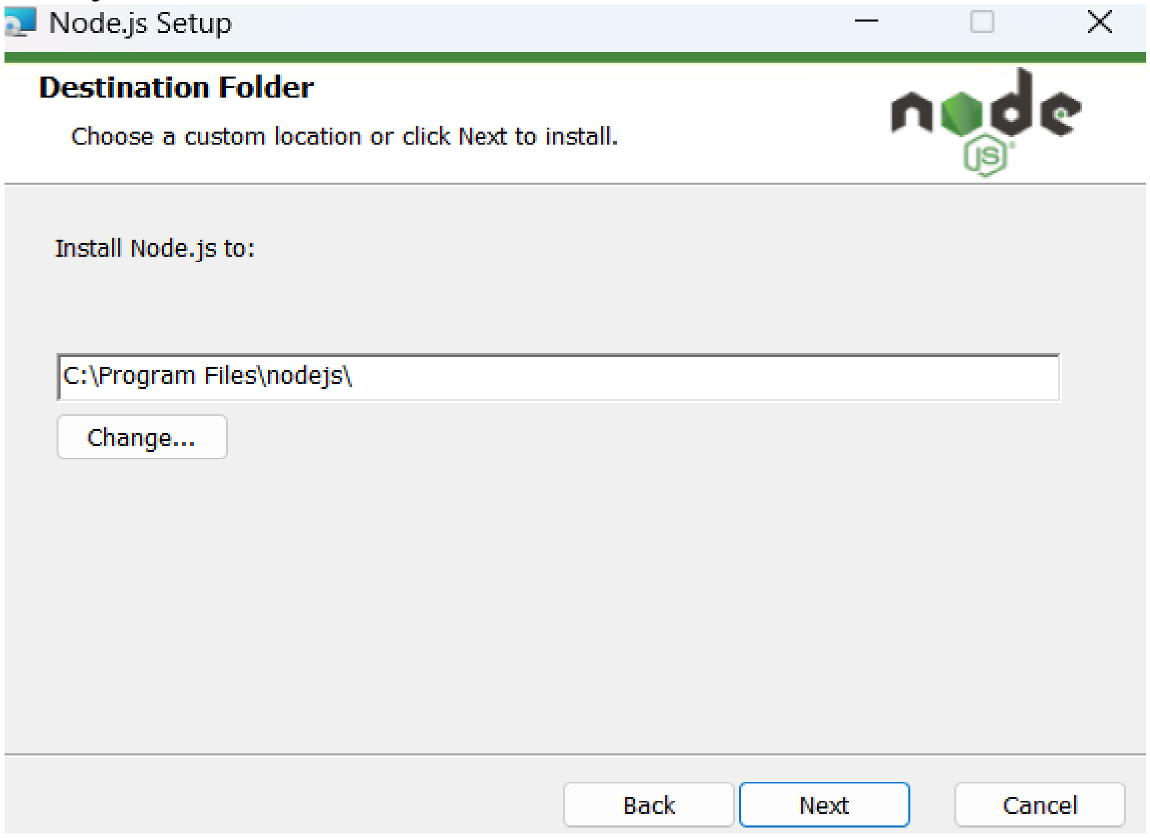

Node.js Setup
End-User License Agreement
Please read the following license agreement carefully

Node.js is licensed for use as follows:

Copyright Node.js contributors. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell

Step 6: By clicking on ‘Change’, set the path, where you want to install the Node.js file and then click on next.



Node.js Setup

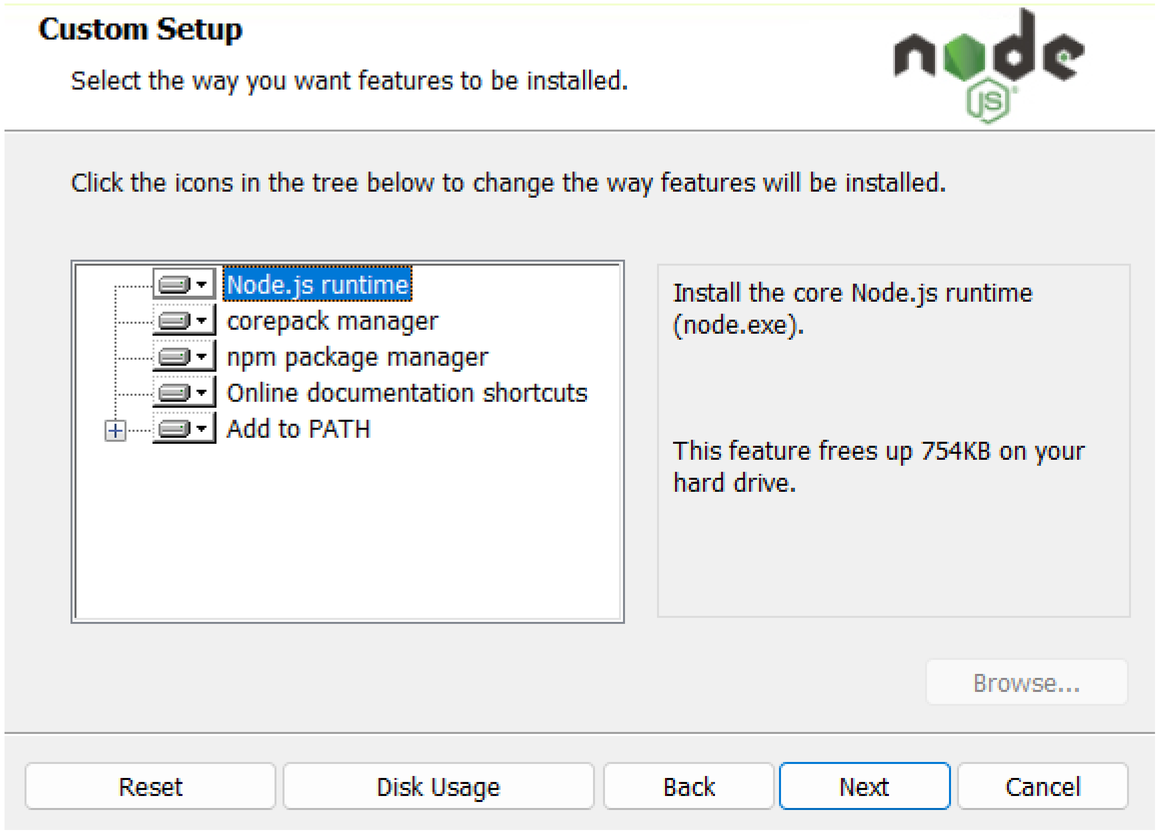
Destination Folder

Choose a custom location or click Next to install.

Install Node.js to:

C:\Program Files\nodejs\

Step 7: Again, click on next.



Custom Setup

Select the way you want features to be installed.

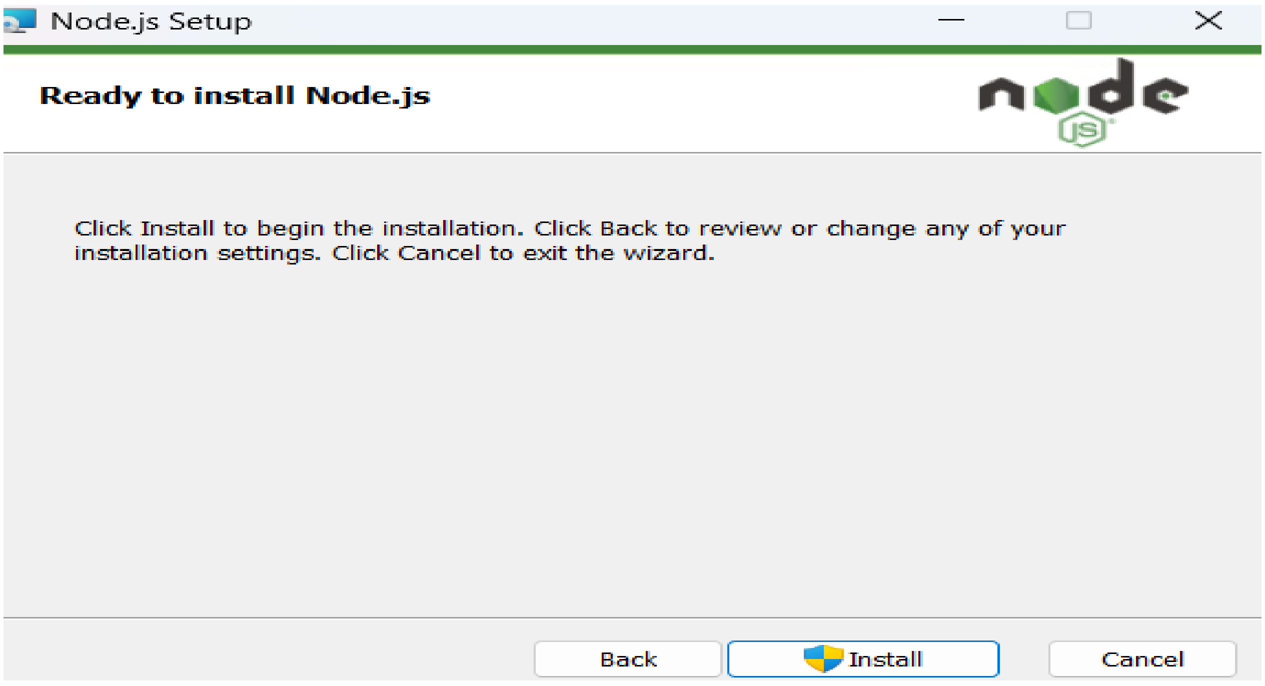
Click the icons in the tree below to change the way features will be installed.

- Node.js runtime
- corepack manager
- npm package manager
- Online documentation shortcuts
- Add to PATH

Install the core Node.js runtime (node.exe).

This feature frees up 754KB on your hard drive.

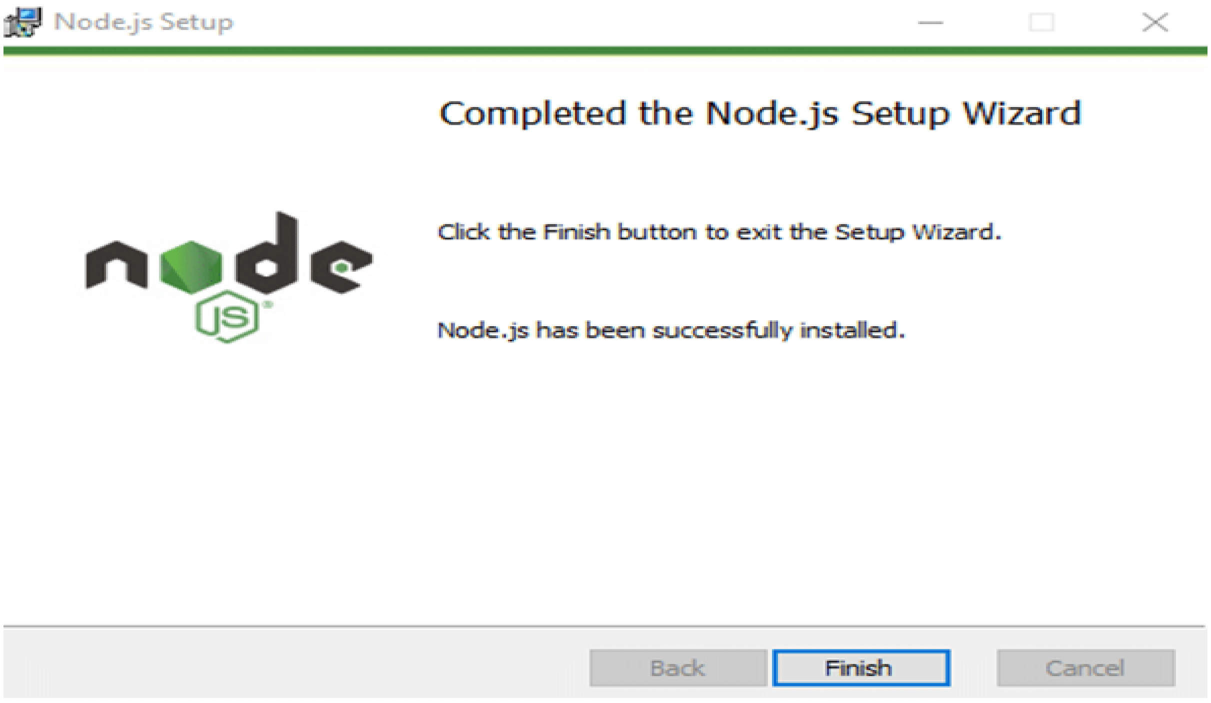
Step 8: Now, click on the 'Install' button to finish up the installation process.


Node.js Setup

Ready to install Node.js

Click Install to begin the installation. Click Back to review or change any of your installation settings. Click Cancel to exit the wizard.

Step 9: Once done with the installation, click on the 'Finish' button to exit the installation wizard.


Node.js Setup

Completed the Node.js Setup Wizard

Click the Finish button to exit the Setup Wizard.

Node.js has been successfully installed.

Step 10: So, the installation process is completed. Verify the installation and correct version, open your PC's command prompt and enter the following command: **'node -v** or **node –version' and 'npm-v',** these commands will display the installed version of Node.js and npm.

Q7. What is REPL. Explain it details through Example.

In node.js REPL environment stands for READ, EVAL, PRINT, LOOP which is similar to command prompt or shell. It is an interactive shell that processes Node.js expressions.

- **READ:** The input provided by the user is read. After necessary parsing, the information is saved in the memory.
- **Eval:** The data structure with the parsed information is evaluated, and a result is generated. Print: The result generated after the evaluation is displayed to the user.
- **Print:** The result generated after the evaluation is displayed to the user.
- **Loop:** All the above three steps are looped until the user presses ctrl+c twice to get out of the loop.

Features of REPL:

- **For start node.js in REPL -**

  Open your terminal and type node.

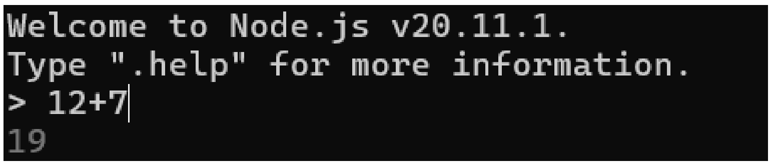
  You will see the REPL prompt (>), it means it's ready for you.

- **Code Execution in REPL-**

  The REPL is a quick way to test JavaScript code without having to create a file.

  We can perform almost every valid node.js expression code executions in node.js without creating a file.

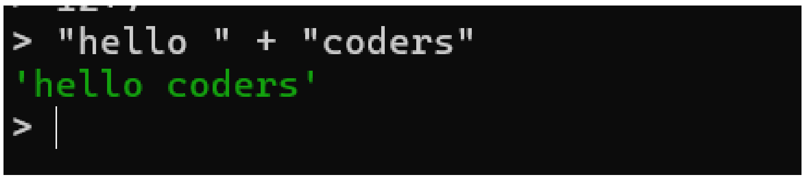
For example: adding two numbers in the REPL type this:



```
Welcome to Node.js v20.11.1.
Type ".help" for more information.
> 12+7
19
```

- **For Concatenate:**

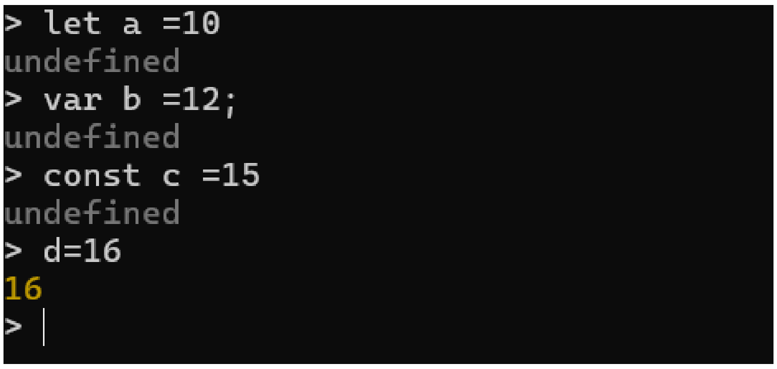
In REPL, we can also perform operation on strings.



```
> "hello " + "coders"
'hello coders'
> 
```

- **Create variables:**

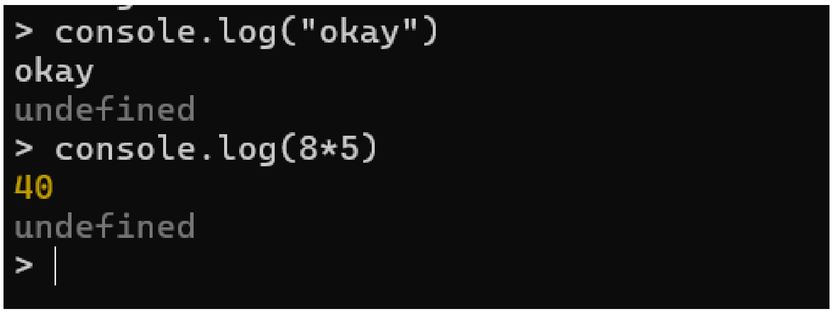
  In this, we can declare variables in 4 ways.



```
> let a =10
undefined
> var b =12;
undefined
> const c =15
undefined
> d=16
16
> |
```

- **Calling Function:**

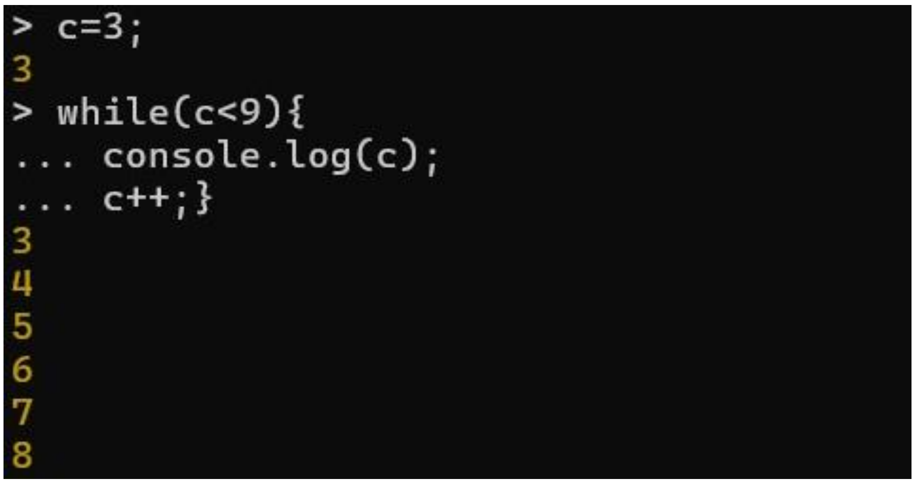
  For print a message, simply we write **console.log( ).**



```
> console.log("okay")
okay
undefined
> console.log(8*5)
40
undefined
> |
```

- **Multiline code:**

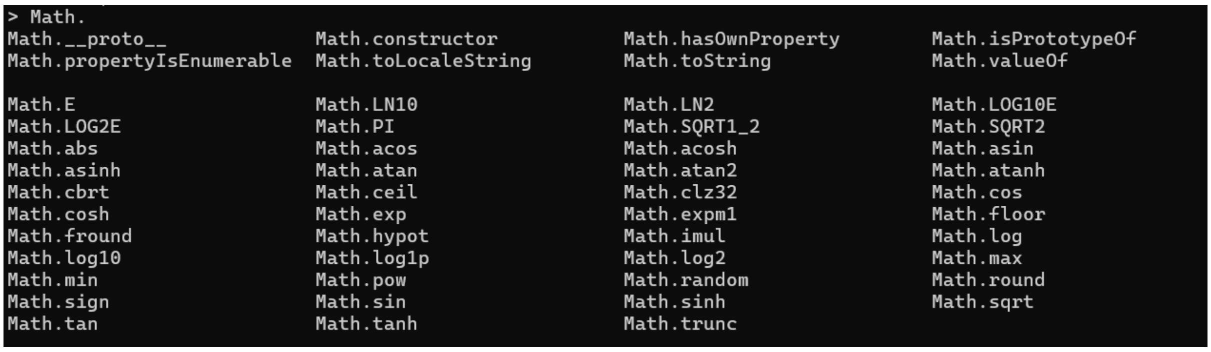
  For longer code, the REPL switches to multiline mode.



```
> c=3;
3
> while(c<9){
... console.log(c);
... c++;}
3
4
5
6
7
8
```

- It also provides some shortcuts to decreasing coding time whenever possible.

For example, if you will enter Math. And press TAB button twice. You get all possibles autocompletions for it.

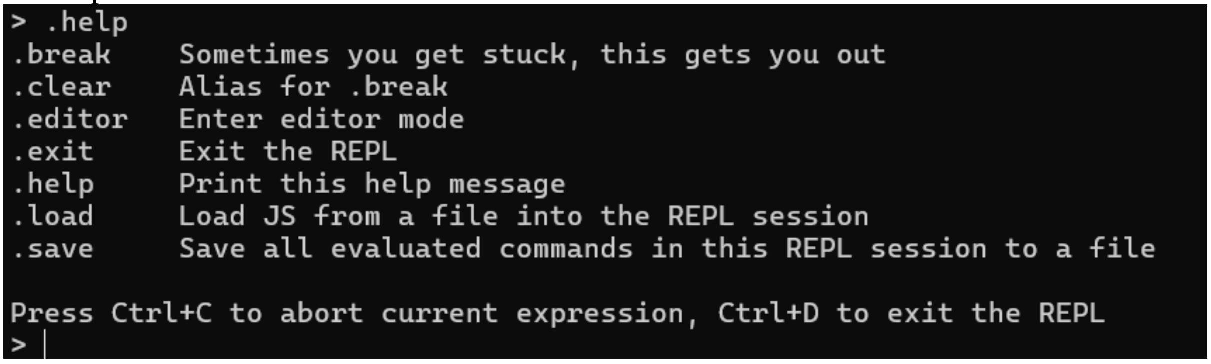


```
> Math.
Math.__proto__              Math.constructor            Math.hasOwnProperty         Math.isPrototypeOf
Math.propertyIsEnumerable   Math.toLocaleString         Math.toString               Math.valueOf

Math.E                      Math.LN10                   Math.LN2                    Math.LOG10E
Math.LOG2E                  Math.PI                     Math.SQRT1_2                Math.SQRT2
Math.abs                    Math.acos                   Math.acosh                  Math.asin
Math.asinh                  Math.atan                   Math.atan2                  Math.atanh
Math.cbrt                   Math.ceil                   Math.clz32                  Math.cos
Math.cosh                   Math.exp                    Math.expm1                  Math.floor
Math.fround                 Math.hypot                  Math.imul                   Math.log
Math.log10                  Math.log1p                  Math.log2                   Math.max
Math.min                    Math.pow                    Math.random                 Math.round
Math.sign                   Math.sin                    Math.sinh                   Math.sqrt
Math.tan                    Math.tanh                   Math.trunc
```

- **REPL commands:**

  The REPL has specific keywords to help control its behavior. Each command begins with a dot.

Example-



```
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the REPL
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file

Press Ctrl+C to abort current expression, Ctrl+D to exit the REPL
> |
```

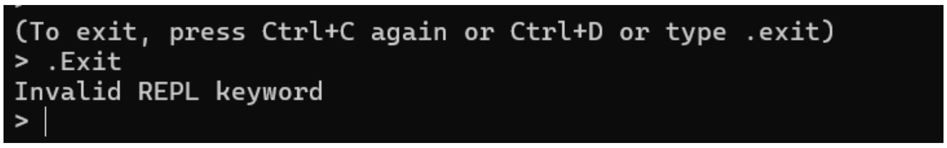
Conclusion:

REPL is an abbreviation for Read Evaluate Print Loop, which is a programming language environment that accepts a single expression as user input, executes it, and returns the result to the console after execution.

Q8. Write a Step for access REPL editor.

REPL stands for Read Evaluate Print Loop. It is a programming language environment (basically a console window/command prompt) that takes single expression as user input and returns the result back to the console after code execution. As we know it is case sensitive, so be careful while write something in it.

For example, we use '.exit' keyword for taking exit from loop but if you enter '.Exit' it throw an error.

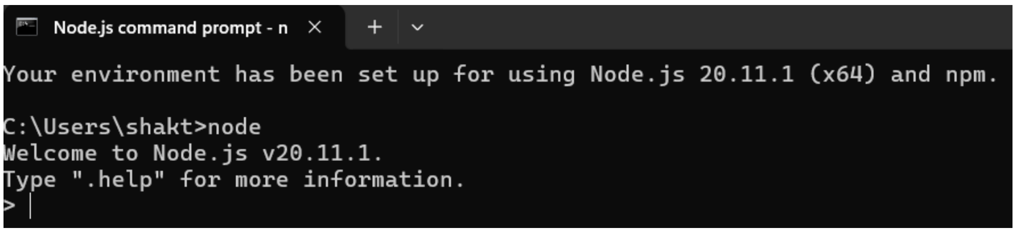


```
(To exit, press Ctrl+C again or Ctrl+D or type .exit)
> .Exit
Invalid REPL keyword
> |
```

Steps for accessing REPL:

Step 1: Open your terminal or command prompt, depending on your operating system.

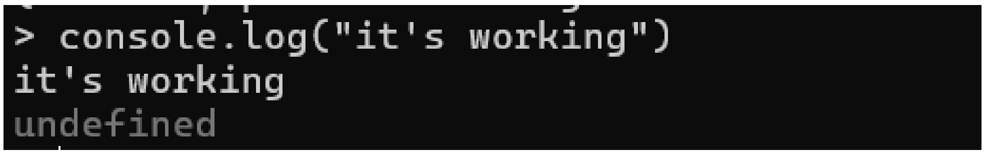
Step 2: To start the REPL type **"node"** and press the enter button. You will see the welcome message and a symbol " > ", which means its ready to take input for evaluation.



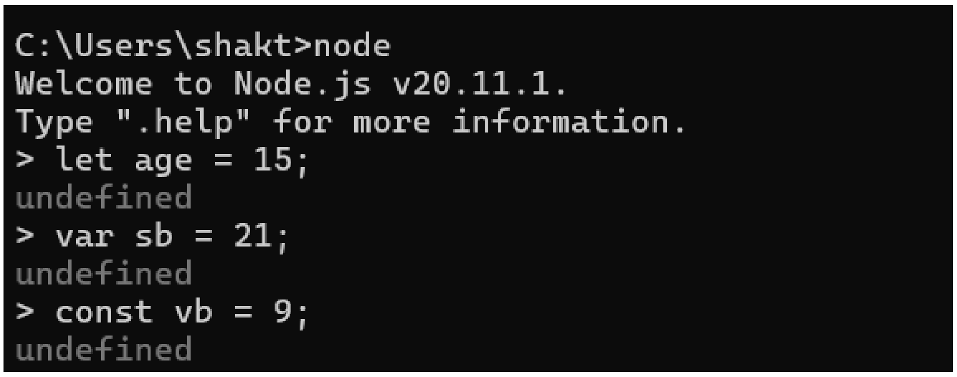
```
Your environment has been set up for using Node.js 20.11.1 (x64) and npm.

C:\Users\shakt>node
Welcome to Node.js v20.11.1.
Type ".help" for more information.
> |
```

Step 3: Now you can write javaScript expressions and press enter for code execution. In this you can also create and declare variables.

For Example,

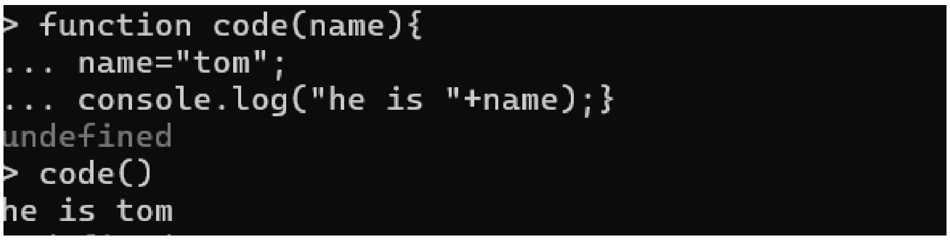


```
> console.log("it's working")
it's working
undefined
```



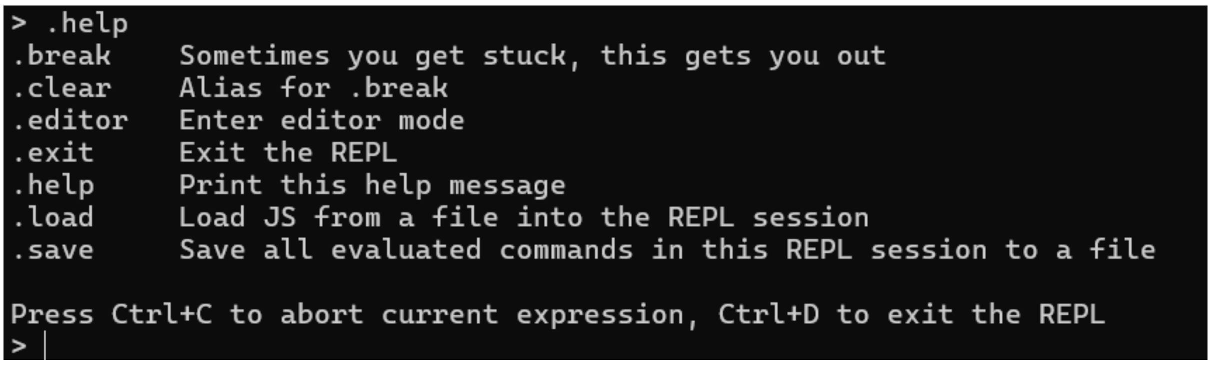
```
C:\Users\shakt>node
Welcome to Node.js v20.11.1.
Type ".help" for more information.
> let age = 15;
undefined
> var sb = 21;
undefined
> const vb = 9;
undefined
```

Step 4: You can also write multiline code in this. Node.js supports it, you can write function as well in REPL.



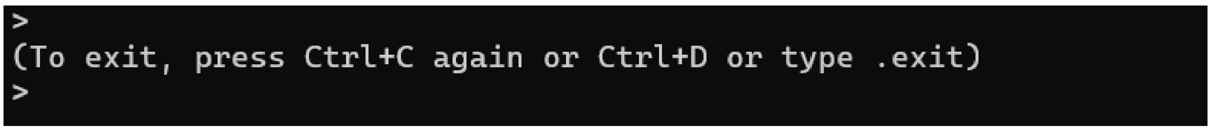
```
> function code(name){
... name="tom";
... console.log("he is "+name);}
undefined
> code()
he is tom
```

Step 5: If you want to access some predefine property / commands then just write **.help** on your terminal and press tab twice it will show you some suggestions or shortcuts for REPL editor.



```
> .help
.break    Sometimes you get stuck, this gets you out
.clear    Alias for .break
.editor   Enter editor mode
.exit     Exit the REPL
.help     Print this help message
.load     Load JS from a file into the REPL session
.save     Save all evaluated commands in this REPL session to a file

Press Ctrl+C to abort current expression, Ctrl+D to exit the REPL
>
```

Step 6: If you want to exit from loop and return to the command prompt then just enter **".exit or ctrl+c twice or ctrl+d "** and you will be out of loop in REPL editor.

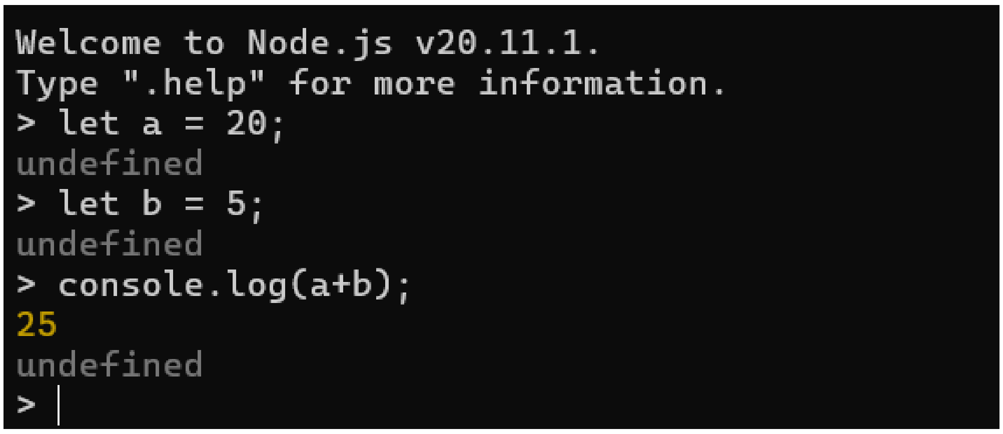


```
>
(To exit, press Ctrl+C again or Ctrl+D or type .exit)
>
```

Q9. **Perform following operation in REPL editor.**

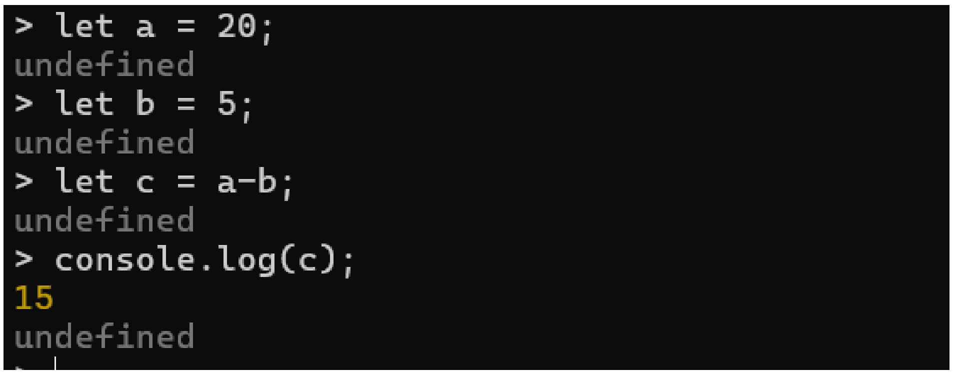
i. **Various arithmetic operations**

a) **Addition Operator:**



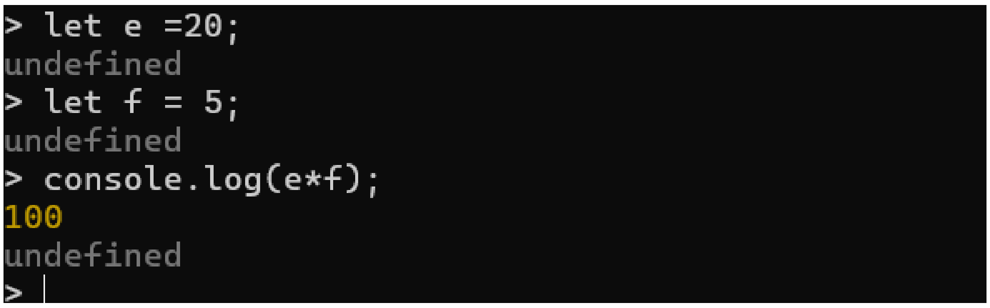
```
Welcome to Node.js v20.11.1.
Type ".help" for more information.
> let a = 20;
undefined
> let b = 5;
undefined
> console.log(a+b);
25
undefined
> 
```

b) **Subtraction Operator:**



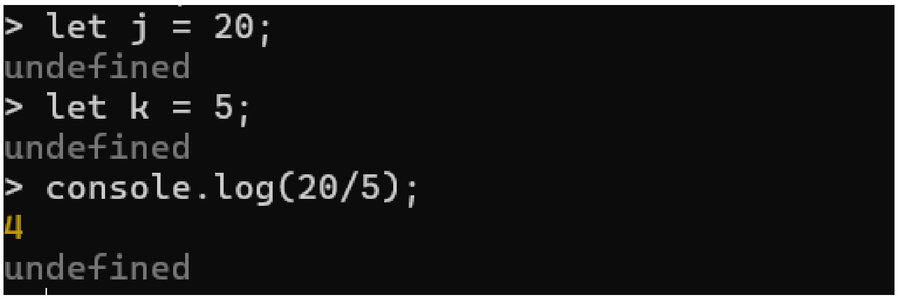
```
> let a = 20;
undefined
> let b = 5;
undefined
> let c = a-b;
undefined
> console.log(c);
15
undefined
```

c) **Multiplication Operator:**



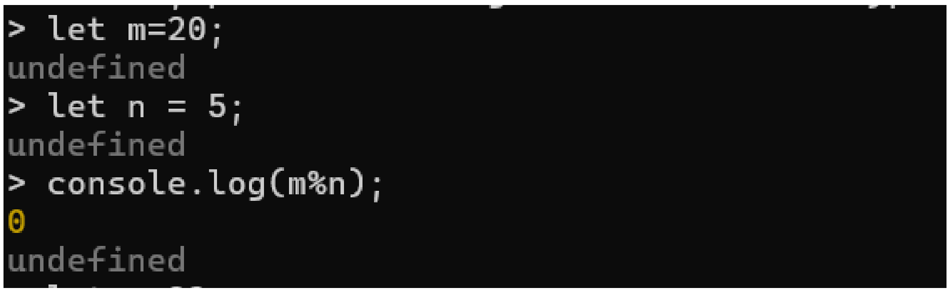
```
> let e =20;
undefined
> let f = 5;
undefined
> console.log(e*f);
100
undefined
> 
```

d) **Division Operator:**



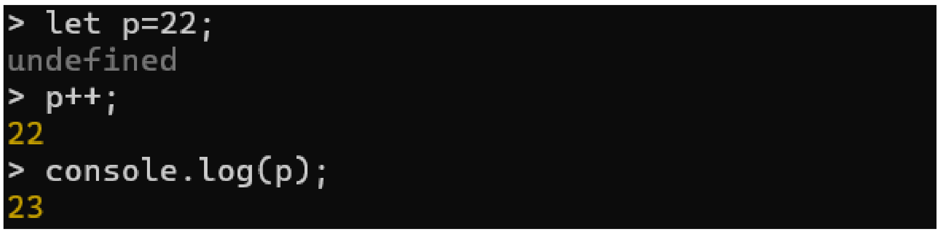
```
> let j = 20;
undefined
> let k = 5;
undefined
> console.log(20/5);
4
undefined
```

e) **Modulus Operator:**



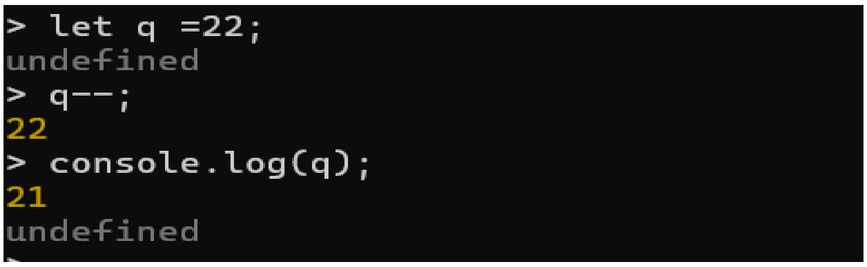
```
> let m=20;
undefined
> let n = 5;
undefined
> console.log(m%n);
0
undefined
```

f) **Increment Operator:**



```
> let p=22;
undefined
> p++;
22
> console.log(p);
23
```

g) **Decrement Operator:**



```
> let q =22;
undefined
> q--;
22
> console.log(q);
21
undefined
```

ii. **Variable Declaration:**

In node.js, there is basically 3 types of variables available-

Const

Var

Let

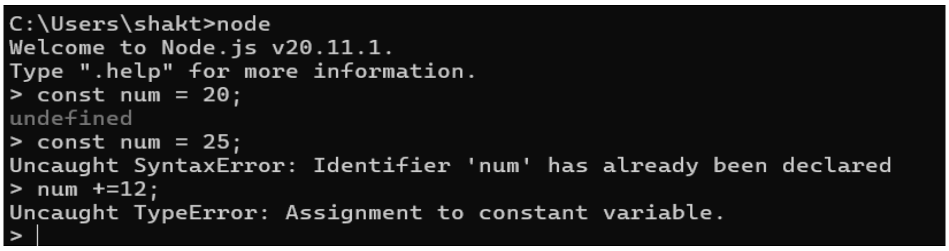
a) **Const-**

The const keyword declares a variable with block scope, block scope is created within specific code blocks, such as conditional statements (if, else, switch) and loops (for, while).Constant variables must be declared and initialized at the same time.

Block-scoped: Variables declared with const also have block scope.

Immutable: Once assigned, the value of a const variable cannot be changed.

No Re-declaration or Update: You cannot re-declare or update const variables.



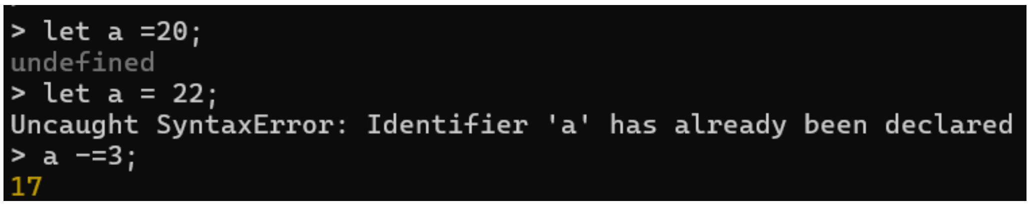
```
C:\Users\shakt>node
Welcome to Node.js v20.11.1.
Type ".help" for more information.
> const num = 20;
undefined
> const num = 25;
Uncaught SyntaxError: Identifier 'num' has already been declared
> num +=12;
Uncaught TypeError: Assignment to constant variable.
> |
```

b) Let-

Variable names are case-sensitive in JavaScript. You cannot declare a duplicate variable using the let keyword with the same name and case. JavaScript will throw a syntax error.

Block-scoped: Variables declared with let have block scope (confined to the block in which they are declared).

Re-declaration and Update: You can re-declare and update var variables within their scope.

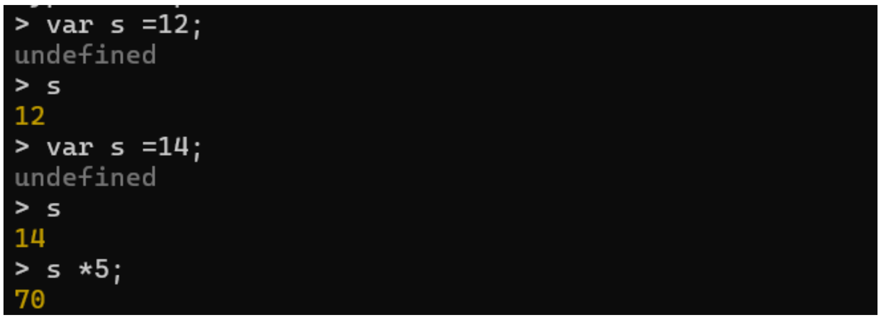


```
> let a =20;
undefined
> let a = 22;
Uncaught SyntaxError: Identifier 'a' has already been declared
> a -=3;
17
```

c) Var:

Variables declared with var are function scoped, which means that they are only accessible within the function in which they are declared. If a variable is declared with var outside of a function, it is considered to be globally scoped and accessible from anywhere in your code. var variables can also be re-assigned new values throughout the code.

If we don't use any identifier for variable, it will be consider as variable.



```
> var s =12;
undefined
> s
12
> var s =14;
undefined
> s
14
> s *5;
70
```

iii. Variable value accession/output a variable value

To access or output a variable's value in a REPL (Read-Eval-Print Loop) editor, you can simply enter the name of the variable into the prompt. The REPL will evaluate the expression and return the value of the variable.

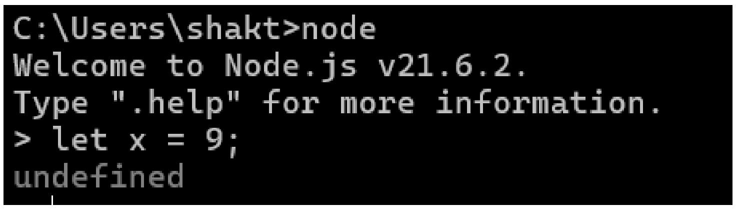
Steps:

- **Start the REPL**:

If you're using Node.js, for example, you can start the REPL by typing node in your command line terminal.

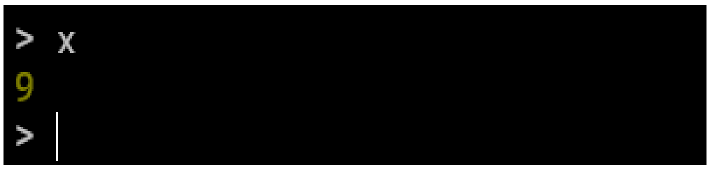
- **Declare a Variable:**

In the REPL, declare a variable and assign it a value. For instance:



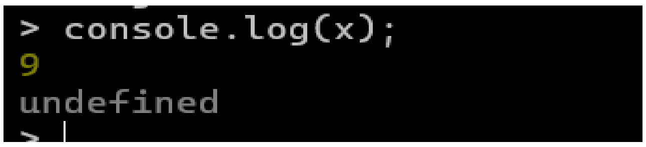
```
C:\Users\shakt>node
Welcome to Node.js v21.6.2.
Type ".help" for more information.
> let x = 9;
undefined
```

- **Access the Variable:** To see the value of the variable x, just type x and press Enter. The REPL will display the value:



```
> x
9
>
```

- **Output the Variable:** If you want to output the variable using a function like console.log, you can do so as follows:



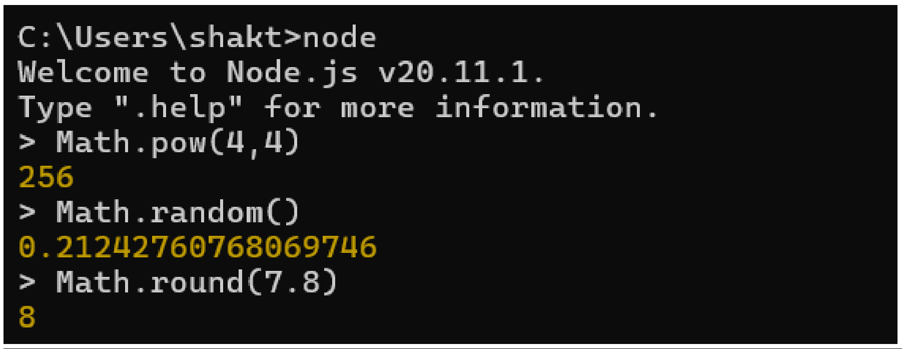
```
> console.log(x);
9
undefined
>
```

This will print the value of x to the console.

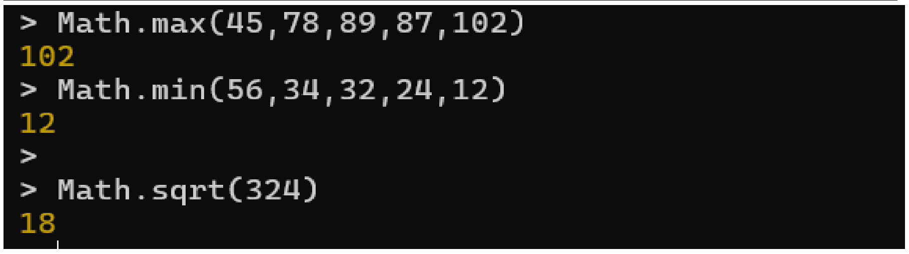
- **Exit the REPL:** To exit the REPL, you can **type .exit, press CTRL+D** once, or press **CTRL+C twice**.

Q10. Perform following operation in REPL editor.

a) Uses Math object related function.



```
C:\Users\shakt>node
Welcome to Node.js v20.11.1.
Type ".help" for more information.
> Math.pow(4,4)
256
> Math.random()
0.21242760768069746
> Math.round(7.8)
8
```

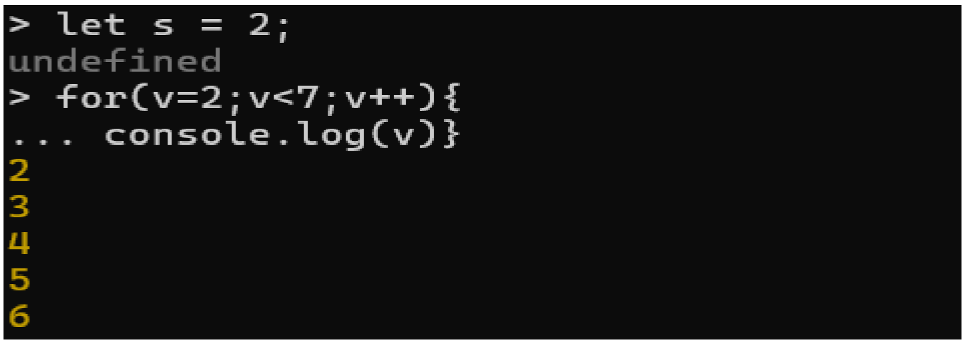


```
> Math.max(45,78,89,87,102)
102
> Math.min(56,34,32,24,12)
12
>
> Math.sqrt(324)
18
```

b) For, do-while, while looping statement:

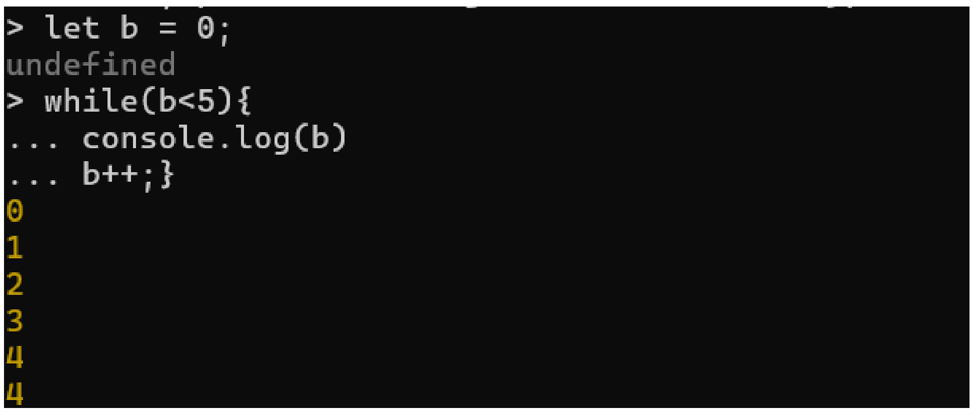
Loops in Node.js are used to iterate through an expression for a particular condition.

For loop:



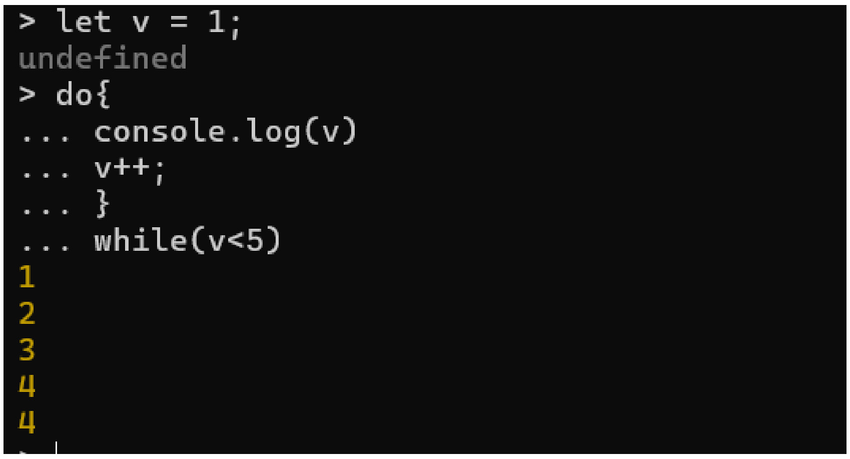
```
> let s = 2;
undefined
> for(v=2;v<7;v++){
... console.log(v)}
2
3
4
5
6
```

While loop:



```
> let b = 0;
undefined
> while(b<5){
... console.log(b)
... b++;}
0
1
2
3
4
4
```

Do-while loop:

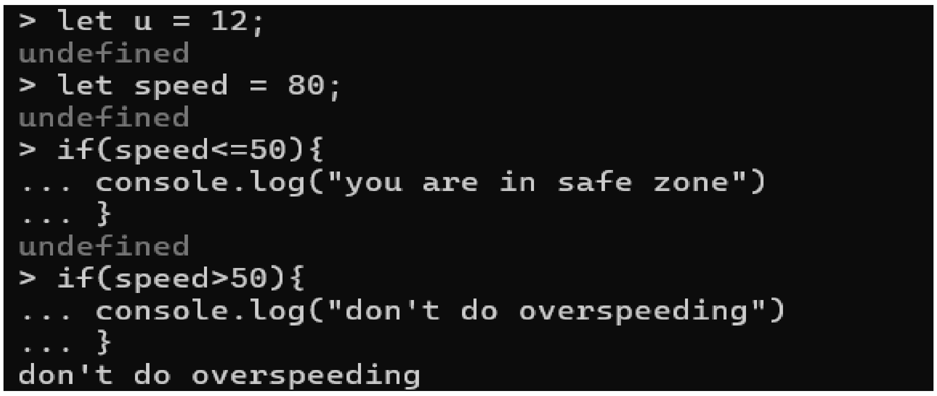


```
> let v = 1;
undefined
> do{
... console.log(v)
... v++;
... }
... while(v<5)
1
2
3
4
4
```

c) if, if-else and elseif branching statement:

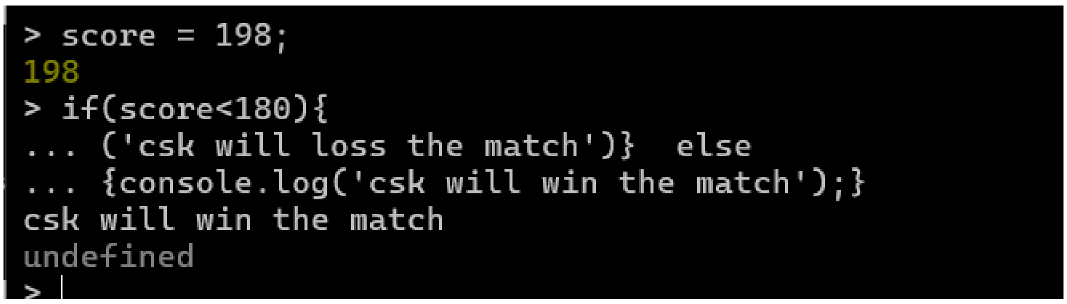
you can use conditional statements to control the flow of your program.

if statement:



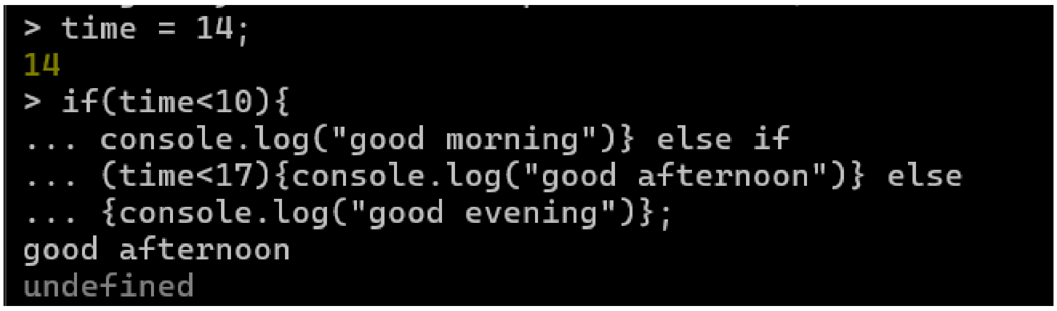
```
> let u = 12;
undefined
> let speed = 80;
undefined
> if(speed<=50){
... console.log("you are in safe zone")
... }
undefined
> if(speed>50){
... console.log("don't do overspeeding")
... }
don't do overspeeding
```

If-else statement:



```
> score = 198;
198
> if(score<180){
... ('csk will loss the match')}  else
... {console.log('csk will win the match');}
csk will win the match
undefined
>
```

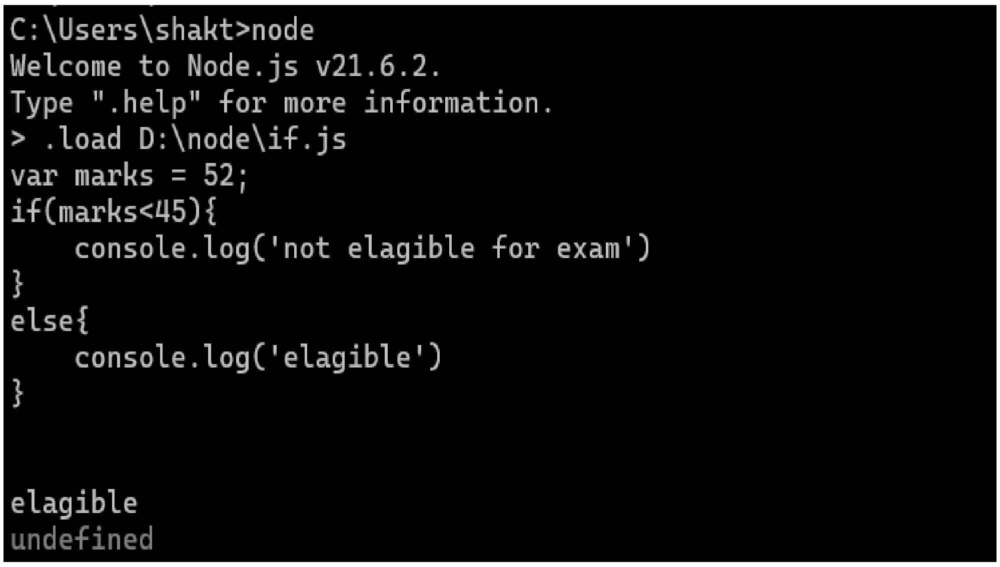
d) Else-if branching:



```
> time = 14;
14
> if(time<10){
... console.log("good morning")} else if
... (time<17){console.log("good afternoon")} else
... {console.log("good evening")};
good afternoon
undefined
```

External JS file execution:

For access external file in REPL editor type **.load** and enter path of that js file.



```
C:\Users\shakt>node
Welcome to Node.js v21.6.2.
Type ".help" for more information.
> .load D:\node\if.js
var marks = 52;
if(marks<45){
    console.log('not elagible for exam')
}
else{
    console.log('elagible')
}


elagible
undefined
```

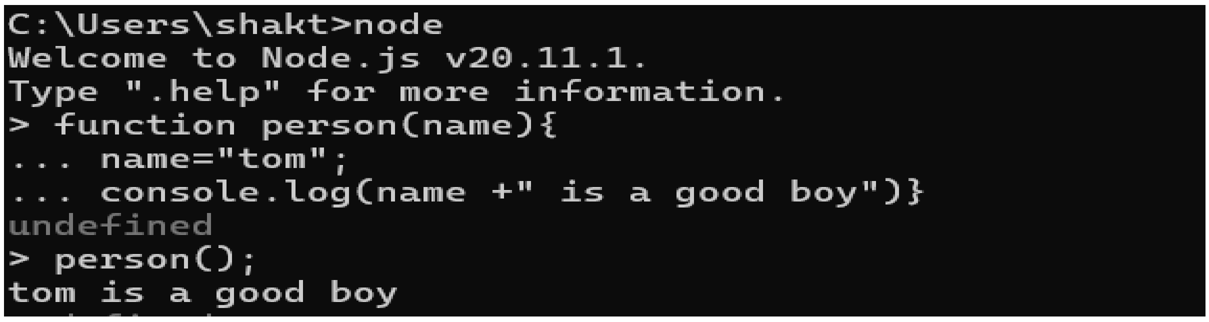
Q11. What is function? Explain the advantages of function through example.

Functions are one of the fundamental concepts in programming. They let us write concise, modular, reusable, and maintainable code. They also help us obey the DRY principle when writing code.

Functions are the main "building blocks" of the program. They allow the code to be called many times without repetition, you can use a function to wrap that code and reuse it.

For declaring a function we use 'function' keyword in our programme. The basic rules of naming a function are similar to naming a variable. It is better to write a descriptive name for your function.

For example-



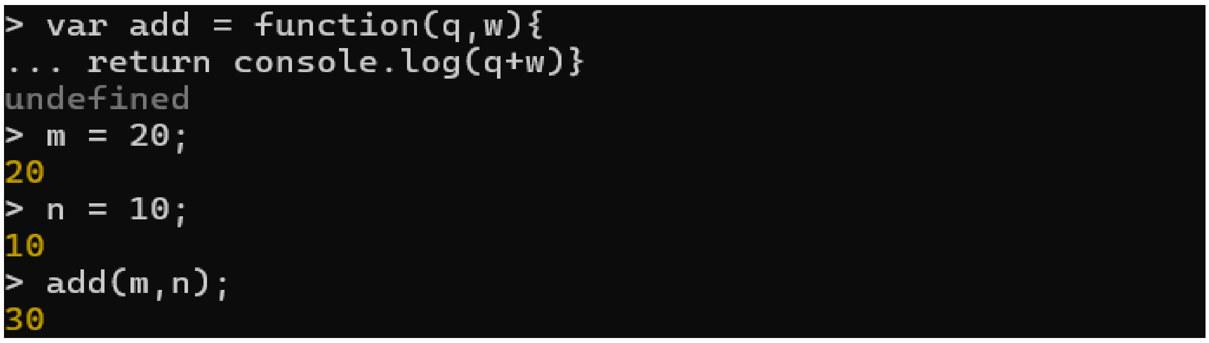
```
C:\Users\shakt>node
Welcome to Node.js v20.11.1.
Type ".help" for more information.
> function person(name){
... name="tom";
... console.log(name +" is a good boy")}
undefined
> person();
tom is a good boy
```

In above example function name is person which declared with a parameter 'name' and function body is written within curly braces{}.

For calling the function we use "person( )".

Function makes the program easier as each small task is divided into a function and increases readability.

Functions can also be defined using expressions. In this case, the function can be anonymous and can be assigned to a variable. This is called a function expression. For example-

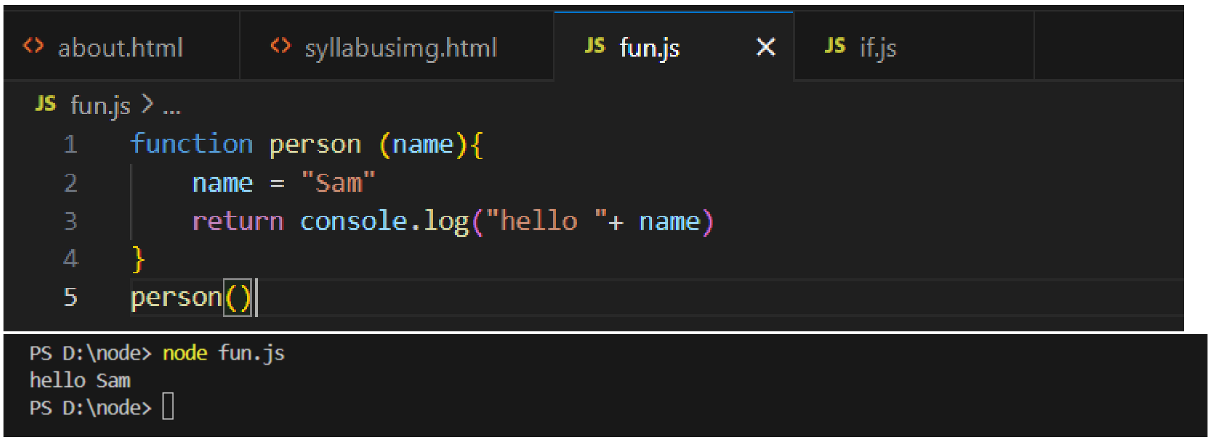


```
> var add = function(q,w){
... return console.log(q+w)}
undefined
> m = 20;
20
> n = 10;
10
> add(m,n);
30
```

Advantages of Function:

Modularity and Code Organization:

- Functions allow you to break down your code into smaller, reusable modules.
- Each function performs a specific task, making your code more organized and easier to maintain.

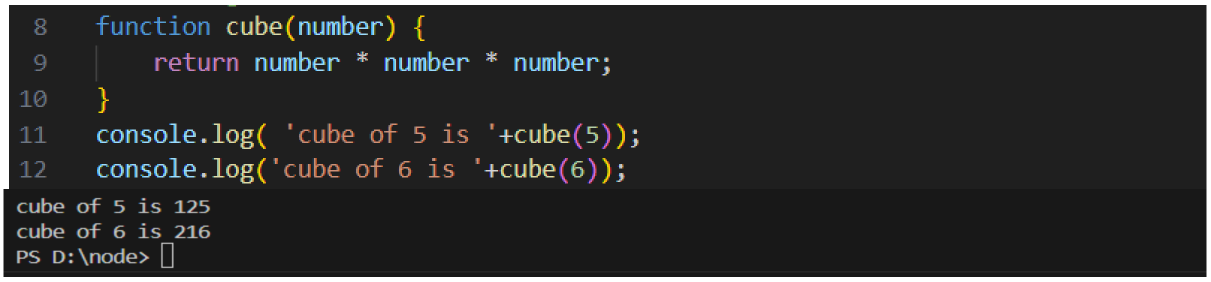


```
function person (name){
    name = "Sam"
    return console.log("hello "+ name)
}
person()
```

```
PS D:\node> node fun.js
hello Sam
PS D:\node>
```

Code Reusability:

- Functions can be defined once and used multiple times throughout your program.
- This reduces redundancy and promotes efficient code maintenance.

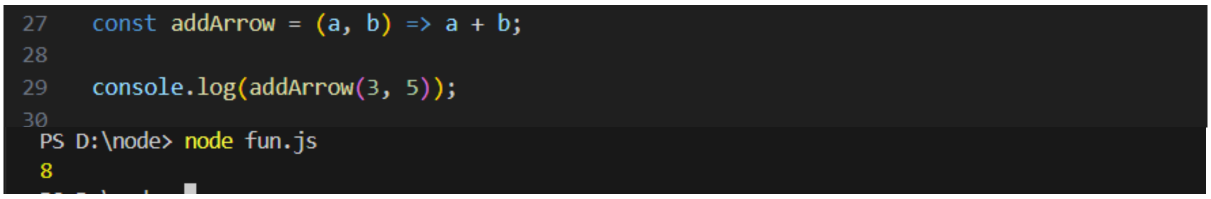


```
function cube(number) {
    return number * number * number;
}
console.log( 'cube of 5 is '+cube(5));
console.log('cube of 6 is '+cube(6));
```

```
cube of 5 is 125
cube of 6 is 216
PS D:\node>
```

Arrow Function:

- An arrow function is defined using the => syntax.
- It can have zero or more parameters, followed by the arrow (=>), and then an expression or a block of code.
- Use arrow functions for concise, single-expression functions.

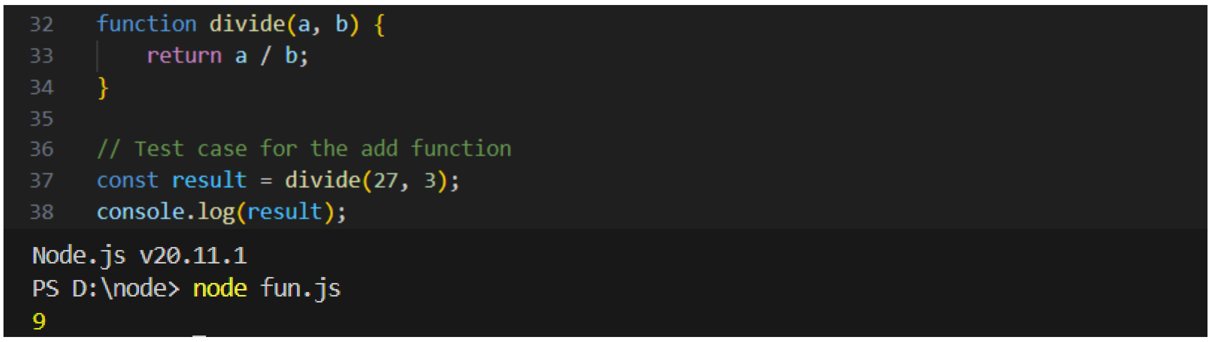


```
const addArrow = (a, b) => a + b;

console.log(addArrow(3, 5));

```

```
PS D:\node> node fun.js
8
```

Testability:

- Functions make code easier to test since you can isolate specific functionalities within functions and test them independently.
- This facilitates unit testing, where each function can be tested in isolation, leading to more reliable and maintainable code.



```
function divide(a, b) {
    return a / b;
}

// Test case for the add function
const result = divide(27, 3);
console.log(result);
```

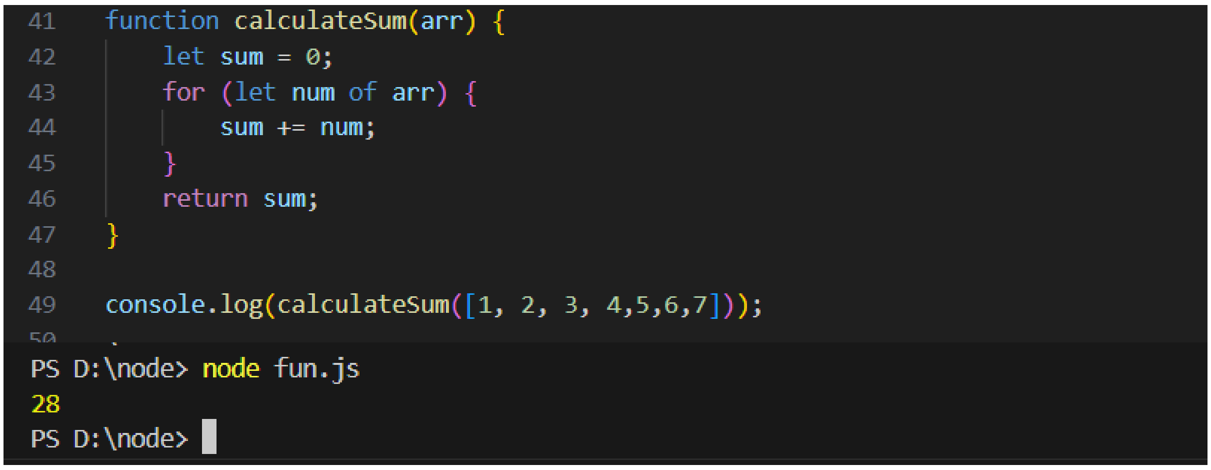
```
Node.js v20.11.1
PS D:\node> node fun.js
9
```

Abstraction and Encapsulation:

- Functions allow you to abstract complex logic behind a simple interface.
- You can hide implementation details and expose only relevant information.

Scoping:

- Functions define their own scope, which helps in preventing variable name clashes and unintended side effects.
- This ensures that variables defined within a function are not accessible outside of it and reducing the risk of bugs.



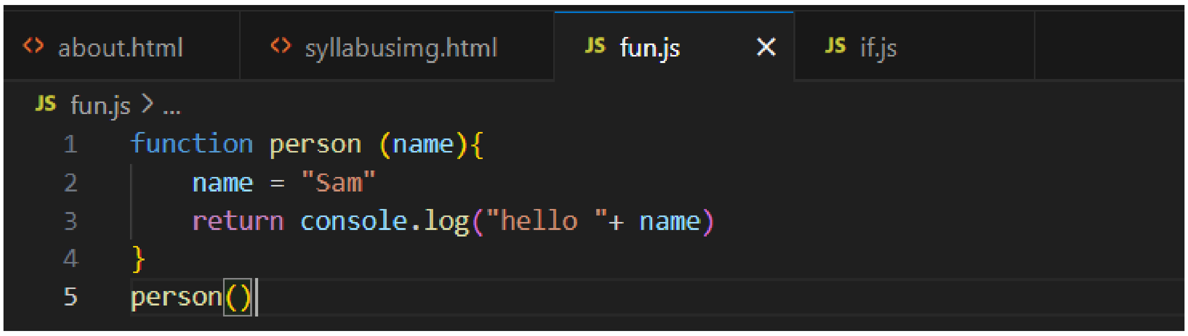
```
function calculateSum(arr) {
    let sum = 0;
    for (let num of arr) {
        sum += num;
    }
    return sum;
}

console.log(calculateSum([1, 2, 3, 4,5,6,7]));
```

```
PS D:\node> node fun.js
28
PS D:\node>
```

Q12. Explain different types of user define function through example?

Regular Named Functions:

- These are the most common type of functions.
- You define them using the function keyword followed by a name.



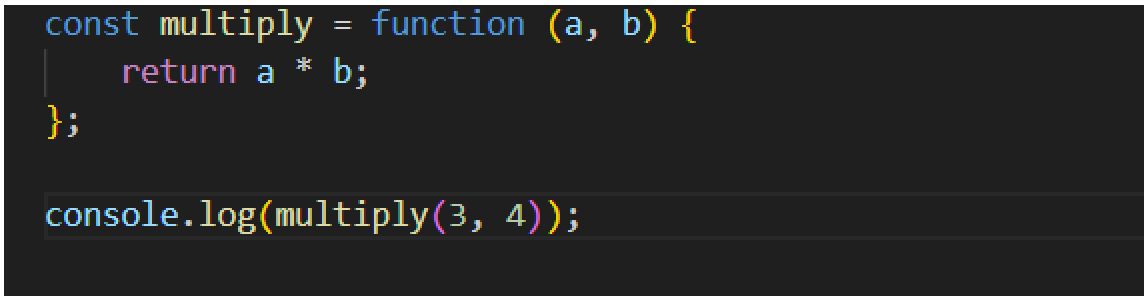
about.html | syllabusimg.html | fun.js | if.js

fun.js > ...

```javascript
function person (name){
    name = "Sam"
    return console.log("hello "+ name)
}
person()
```

Function Expressions:

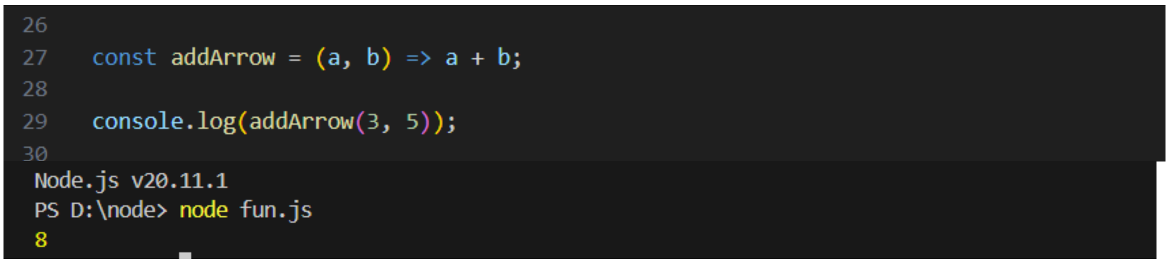
- These functions don't have a name and are assigned to a variable.
- Useful for passing functions as arguments or storing them in variables.



```javascript
const multiply = function (a, b) {
    return a * b;
};

console.log(multiply(3, 4));
```

Arrow Functions :

- A concise way to define functions using the => syntax.
- Especially useful for short, single-expression functions.

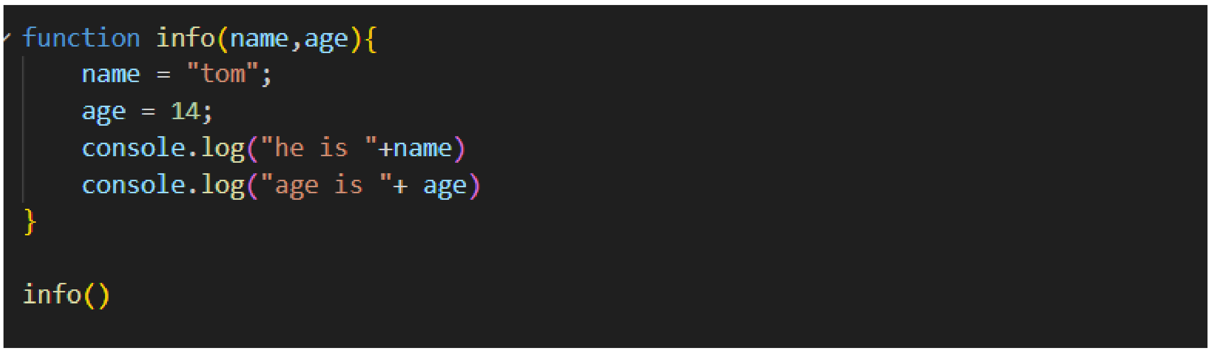


```javascript

const addArrow = (a, b) => a + b;

console.log(addArrow(3, 5));

```

```
Node.js v20.11.1
PS D:\node> node fun.js
8
```

Functions with Default Parameters:

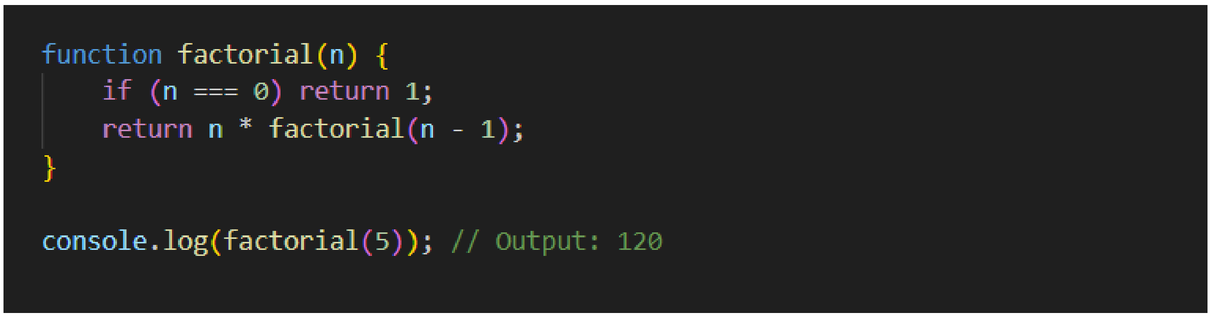
- You can set default values for function parameters.
- If no value is provided, the default value is used.



```
function info(name,age){
    name = "tom";
    age = 14;
    console.log("he is "+name)
    console.log("age is "+ age)
}

info()
```

Recursive Functions:

- A function that calls itself during its execution.
- Useful for solving problems that can be broken down into smaller subproblems.

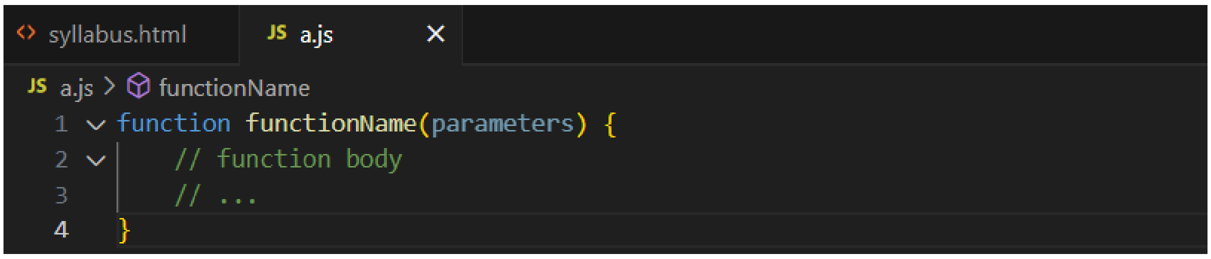


```
function factorial(n) {
    if (n === 0) return 1;
    return n * factorial(n - 1);
}

console.log(factorial(5)); // Output: 120
```

Q13. Explain the component of function implementation?

Function Declaration:

A function is defined using the function keyword followed by the function name and a list of parameters enclosed in parentheses.



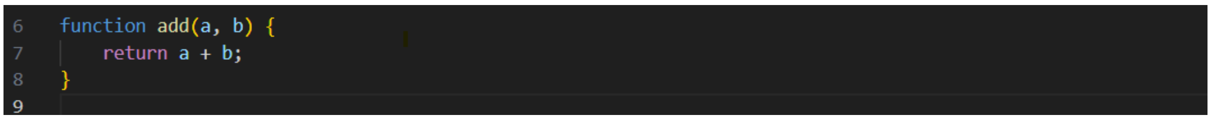
syllabus.html | a.js

a.js > functionName

```javascript
function functionName(parameters) {
    // function body
    // ...
}
```

Function Parameters:

- Parameters are variables that allow you to pass values into a function.
- They are listed within the parentheses after the function name.

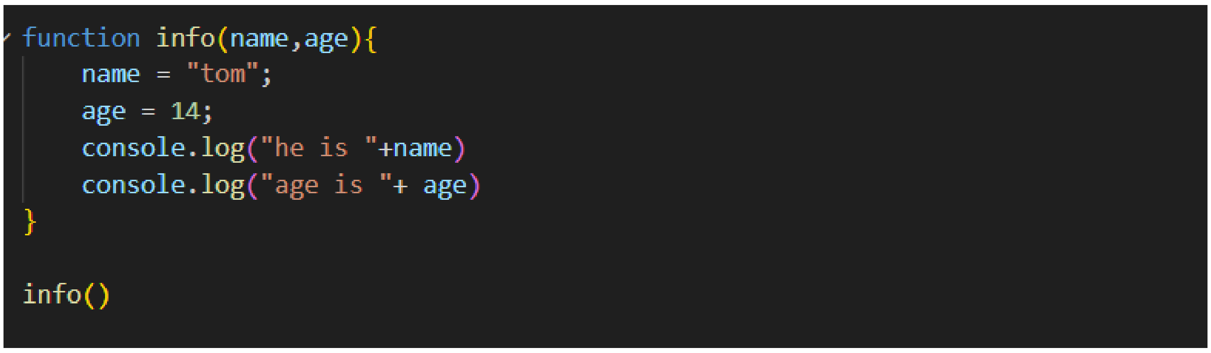


```javascript
function add(a, b) {
    return a + b;
}
```

  - In above example a and b are parameters for the function.

Function Body:

- The function body contains the actual code that executes when the function is called.
- It is enclosed in curly braces {}.

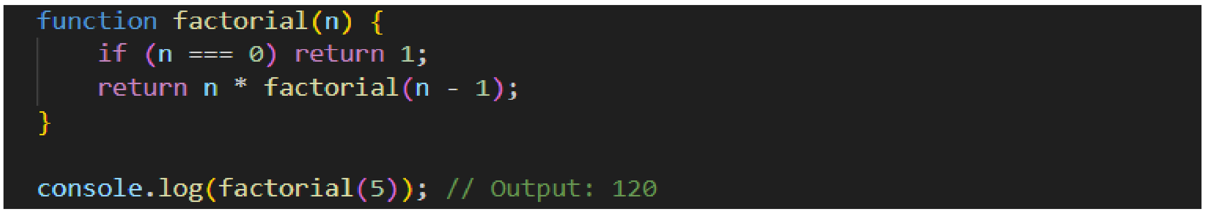


```javascript
function info(name,age){
    name = "tom";
    age = 14;
    console.log("he is "+name)
    console.log("age is "+ age)
}

info()
```

  - The content which is written inside {} is function body for this function.

Return Statement:

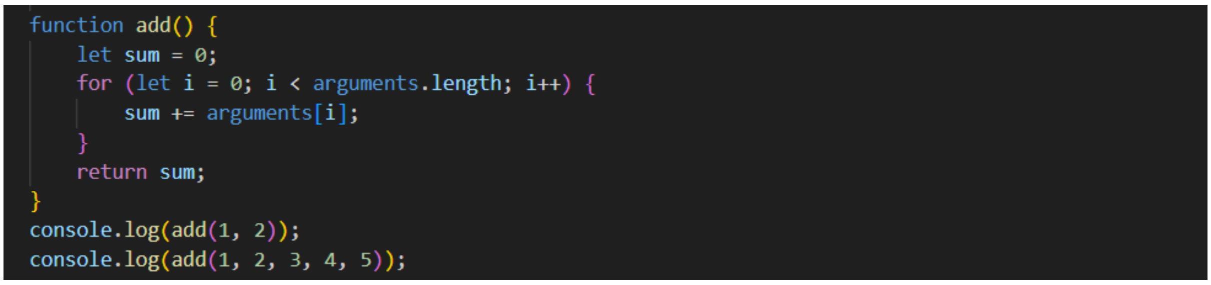
- The return statement specifies the value that the function will produce.
- It ends the execution of the function and sends the result back to the caller.



```
function factorial(n) {
    if (n === 0) return 1;
    return n * factorial(n - 1);
}

console.log(factorial(5)); // Output: 120
```

The Argument Object:

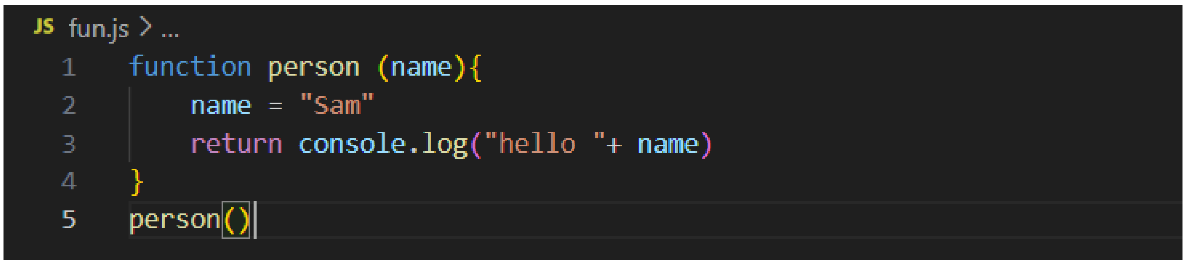
- Inside a function, you can access an object called arguments that represents the named arguments of the function.
- The arguments object behaves like an array though it is not an instance of the Array type.



```
function add() {
    let sum = 0;
    for (let i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}
console.log(add(1, 2));
console.log(add(1, 2, 3, 4, 5));
```

Function Call:

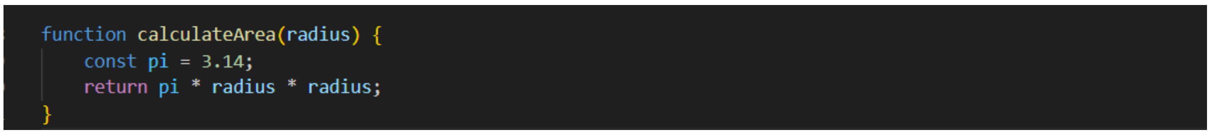
- To execute a function, you call it by its name followed by parentheses.
- Arguments (values) can be passed into the function.



```
function person (name){
    name = "Sam"
    return console.log("hello "+ name)
}
person()
```

Function Scope:

- Variables declared inside a function are scoped to that function.
- They are not accessible outside the function unless explicitly returned.



```
function calculateArea(radius) {
    const pi = 3.14;
    return pi * radius * radius;
}
```

Q14. Explain following functions through advantages, syntax and node JS program.

a) Anonymous Function

An anonymous function is a function without a name. When we create an anonymous function, it is declared without any identifier. They can be beneficial in several programming scenarios, particularly in JavaScript.

Advantages of Anonymous Functions:

- **Concise Syntax:**

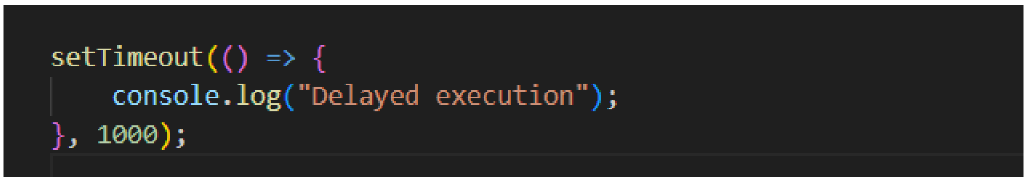
Anonymous functions have a more concise syntax than named functions, making the code more compact and readable in certain scenarios.

- **Immediate Execution:**

Anonymous functions can be immediately executed after their definition, which is useful for creating Immediately Invoked Function Expressions (IIFEs) or closures.

- **Callbacks and Event Handlers:**

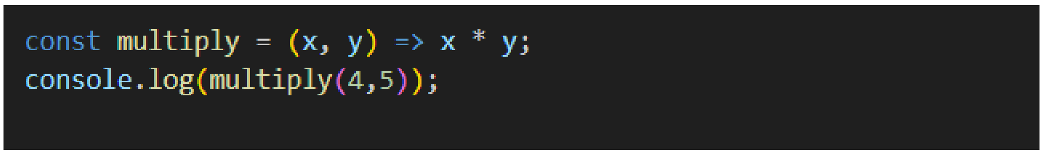
Anonymous functions are commonly used as callback functions or event handlers, where the function itself is not reused elsewhere in the codebase.



```
setTimeout(() => {
    console.log("Delayed execution");
}, 1000);
```

- **Arrow Functions:**

Anonymous functions can be written using the arrow function syntax introduced in ES6, which provides an even more concise way of defining functions.



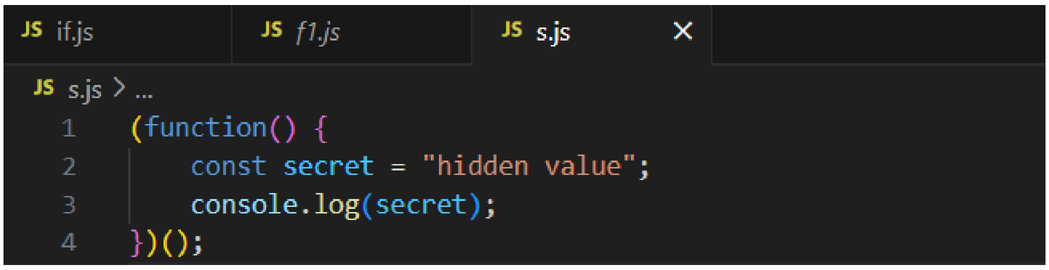
```
const multiply = (x, y) => x * y;
console.log(multiply(4,5));
```

- **Preserving this Context:**

Arrow functions have a lexical this binding, which means they inherit the this value from their enclosing scope, making them useful in certain scenarios where the this context needs to be preserved.

- **Scope Encapsulation:**

They can help encapsulate scope, allowing for variables within the function to be protected from the outside scope.



if.js   f1.js   s.js

s.js > ...

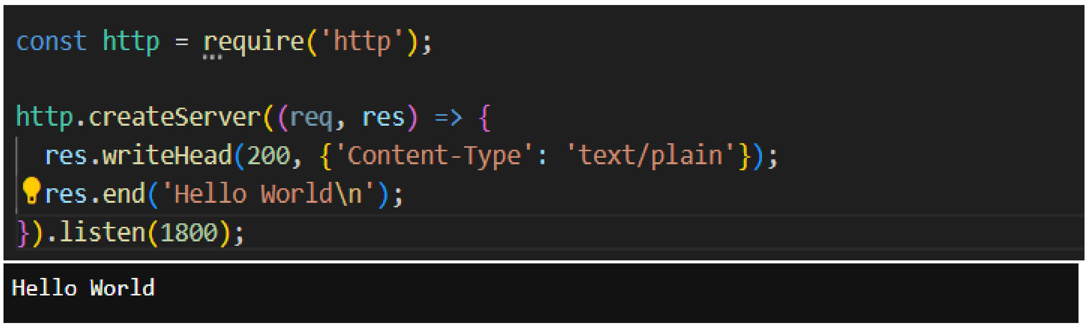
```
(function() {
    const secret = "hidden value";
    console.log(secret);
})();
```

a) Anonymous Function

In Node.js, functions with arguments and no return values are quite common, especially in scenarios involving I/O operations, event handling, or callbacks.

Advantages:

- **Event Handling:** Node.js is event-driven, and many events require handlers that perform an action without needing to return a value. For example, server response handlers often send data to the client without a return statement.



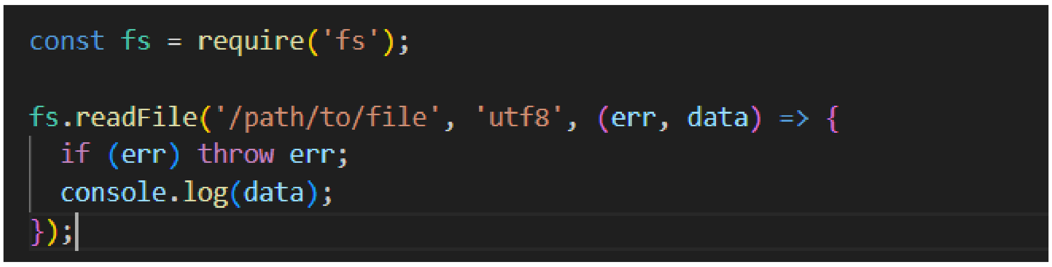
```
const http = require('http');

http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1800);
```

```
Hello World
```

In this example, the function provided to createServer takes request and response objects as arguments and ends the response with "Hello World" without returning any value.

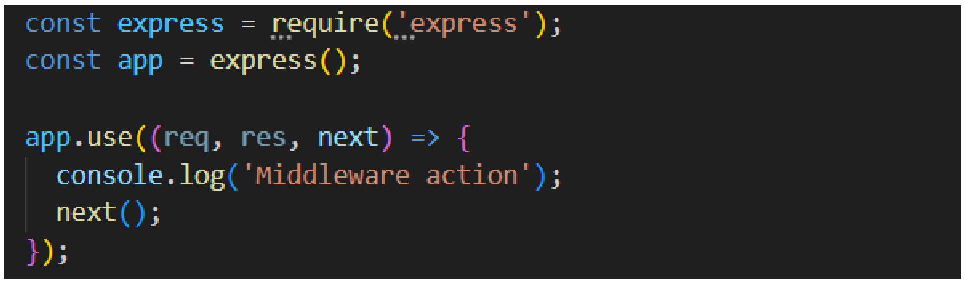
- **Stream Handling:** When working with streams, you often provide functions to handle data events, which process the data chunks as they come in without returning values.

- **Callbacks:** Node.js heavily relies on asynchronous code, and callbacks are used to handle the completion of asynchronous operations. These callback functions often take arguments like error objects and results, and they don't need to return values.



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

fs.readFile('/path/to/file', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});
```

- **Middleware Functions:** In web frameworks like Express, middleware functions are used to modify the request and response objects, perform tasks, or end the response cycle. They typically do not return values but may pass control to the next middleware function.



```
const express = require('express');
const app = express();

app.use((req, res, next) => {
  console.log('Middleware action');
  next();
});
```

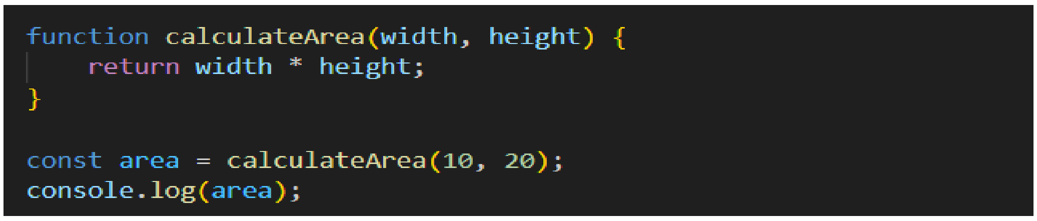
b) Function with argument and return values

Functions with arguments and return values are a core concept in programming, allowing for data to be passed into a function, processed, and then a result to be returned.

Advantages:

- **Modularity:** They break down complex tasks into smaller, manageable **units of code.**
- **Reusability:** Once written, they can be used multiple times, reducing code duplication.
- **Abstraction:** They provide a clear interface, hiding the underlying complexity from the user.
- **Testing:** Easier to test and debug since they have well-defined inputs and outputs.

- **Flexibility:** The internal logic can be changed without affecting other parts of the program as long as the interface remains the same.



```
function calculateArea(width, height) {
    return width * height;
}

const area = calculateArea(10, 20);
console.log(area);
```

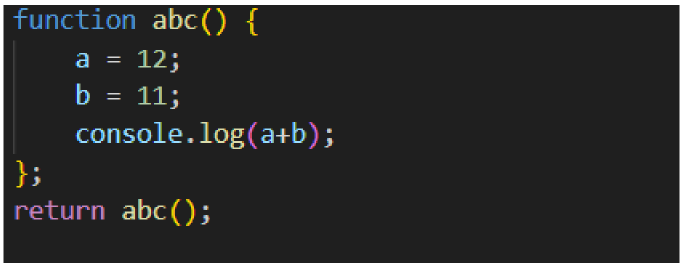
c) Function without argument and return values

Functions without arguments and without return values are often used in programming to execute a block of code that performs an operation but neither takes input nor provides output.

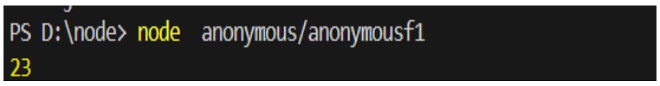
Advantages:

- **Simplicity:** These functions are straightforward to use since they don't require any input and don't need to manage return values.
- **Encapsulation:** They can encapsulate a specific task or behavior that doesn't depend on external data and doesn't need to communicate with other parts of the program.
- **Ease of Invocation**: They can be called without worrying about passing parameters or handling a return value, which simplifies their invocation.
- **Task Automation:** They are ideal for automating repetitive tasks that need to be executed regularly, such as initializing an environment or resetting variables.
- **Event Handling:** In event-driven programming, such as GUI applications, they can be used to handle events where the event itself provides all necessary context.

Example:



```
function abc() {
    a = 12;
    b = 11;
    console.log(a+b);
};
return abc();
```



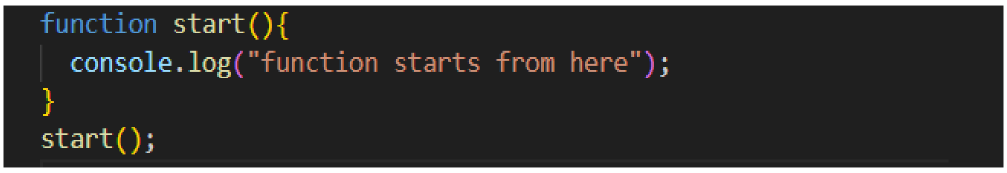
```
PS D:\node> node  anonymous/anonymousf1
23
```

d) Function without argument and no return values

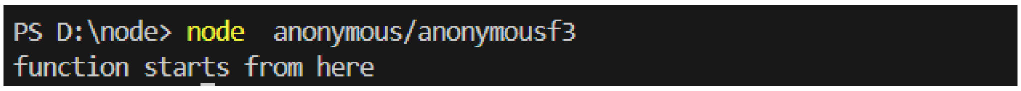
Advantages:

- **Callbacks and Event Handlers:** Node.js is event-driven, and these functions are often used as callbacks for events or asynchronous operations, where they execute a side effect but don't need to return data.
- **Task Automation:** They can automate repetitive tasks like logging, cleaning up resources, or setting up initial states.
- **Statelessness:** Since they don't depend on input arguments or a return value, they can be considered stateless, which is a desirable property in functional programming and can lead to fewer bugs.
- **Testing:** Functions without arguments and no return values are typically easier to test because their behaviour is not dependent on external data.
- **Performance:** They can sometimes offer performance benefits, as there's no overhead of passing arguments or generating a return value.

Example:



```
function start(){
  console.log("function starts from here");
}
start();
```



```
PS D:\node> node  anonymous/anonymousf3
function starts from here
```

Q15. What is module. Explain it advantages through example.

Module: In Node.js, a module is a self-contained unit of code that encapsulates a set of related functions, objects, or variables. Modules can be imported and used in other parts of a Node.js application, promoting code reuse and maintainability.

There are three types of modules in Node.js based on their location to access.

I. **Built-in Modules:**
Node.js has many built-in modules that are part of the platform and come with Node.js installation. These modules can be loaded into the program by using the required function.

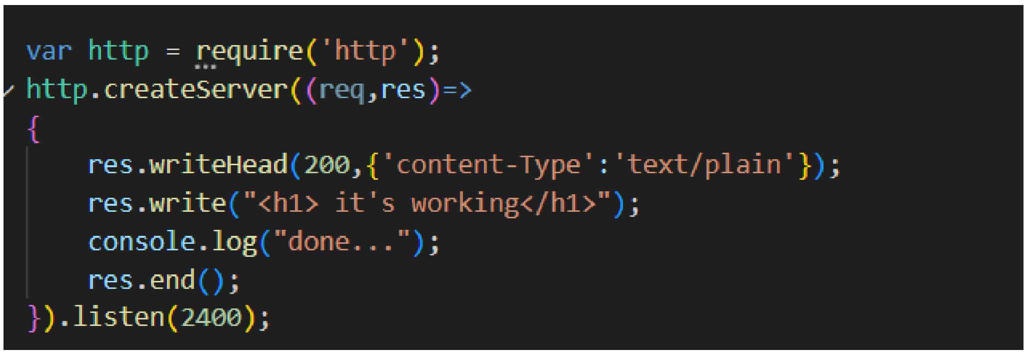
Syntax: var http = require('<module_name>');

II. **User defined Modules:**
Unlike built-in and external modules, local modules are created locally in your Node.js application.

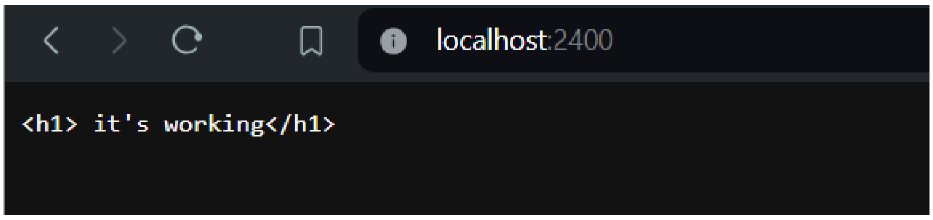
III. **Third Party Module:**
Third-party modules are modules that are available online using the Node Package Manager(NPM). These modules can be installed in the project folder or globally.
Example: Mongoose, express, angular, react etc.

Using Function of a Module:

Now we shall use a function of http module called createServer() to demonstrate on how to use a function of a module.



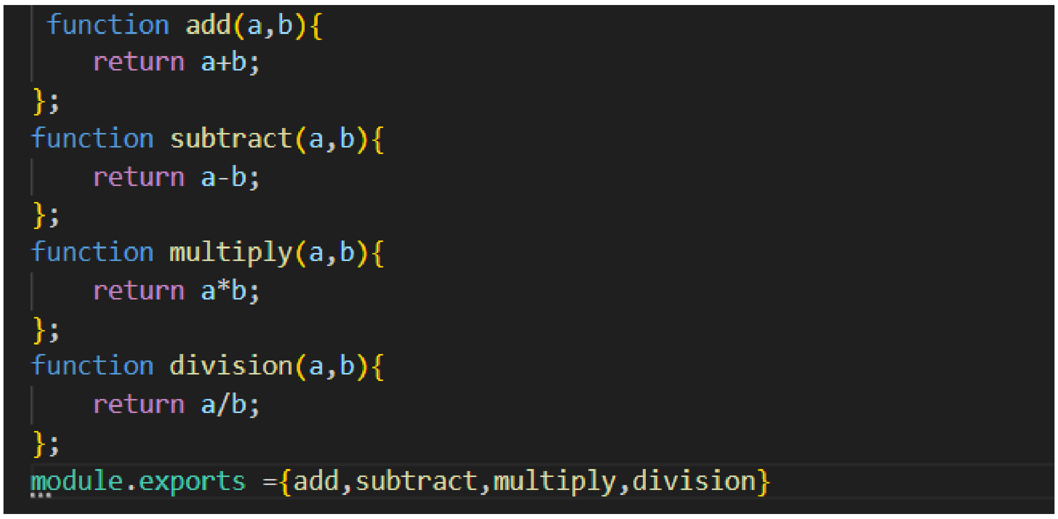
```
var http = require('http');
http.createServer((req,res)=>
{
    res.writeHead(200,{'content-Type':'text/plain'});
    res.write("<h1> it's working</h1>");
    console.log("done...");
    res.end();
}).listen(2400);
```



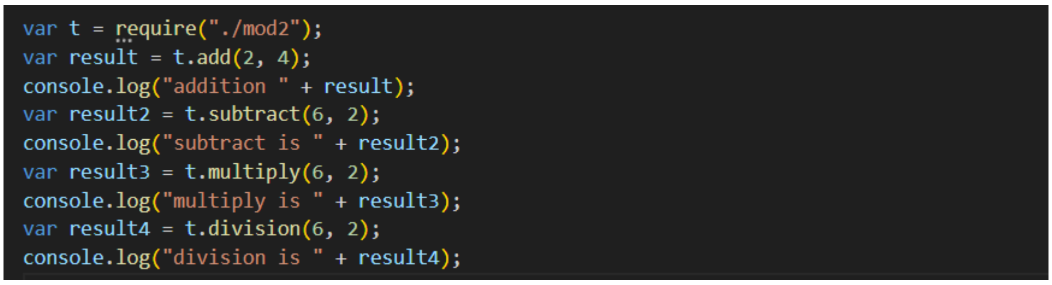
```
<h1> it's working</h1>
```

Advantages of Modules:

- **Modularity:**
  Modules allow you to break down complex code into smaller, manageable pieces, making it easier to understand and maintain.
- **Reusability:**
  Code within a module can be reused across different parts of an application, reducing redundancy and saving development time.
- **Namespace:**
  Modules provide their own scope, which means they help avoid global namespace pollution. Variables and functions defined in a module won't clash with those in other modules or the global scope.
- **Maintainability:**
  With modules, you can update a single part of your application without affecting the rest, as long as the module's interface remains the same.
- **Collaboration:**
  Modules enable teams to work on different features simultaneously without stepping on each other's toes, as each module can be developed independently.



```
 function add(a,b){
    return a+b;
};
function subtract(a,b){
    return a-b;
};
function multiply(a,b){
    return a*b;
};
function division(a,b){
    return a/b;
};
module.exports ={add,subtract,multiply,division}
```



```
var t = require("./mod2");
var result = t.add(2, 4);
console.log("addition " + result);
var result2 = t.subtract(6, 2);
console.log("subtract is " + result2);
var result3 = t.multiply(6, 2);
console.log("multiply is " + result3);
var result4 = t.division(6, 2);
console.log("division is " + result4);
```

Q16. Explain basic component of module. Explain creation and accession steps of module through example.

In Node.js, a module is a separate file containing code-functions, objects, or variables-that can be reused throughout the Node.js application. Here’s a breakdown of the basic components and steps for creating and accessing a module:

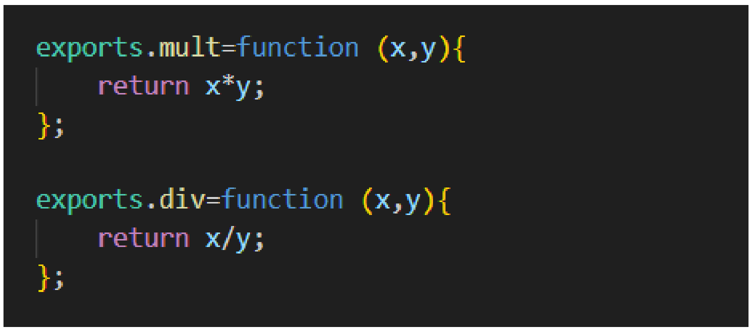
Basic Components of a Node.js Module:

- **Module Scope:** Each module in Node.js has its own scope. Variables, functions, and objects defined in a module are private to that module unless explicitly exported.
- **Exports and module.exports**: These are objects that are used to expose parts of the module to other files. Anything assigned to exports or module.exports becomes accessible to other modules through the require function.
- **Require Function:** This is used to import modules. The require function reads a JavaScript file, executes the file, and then proceeds to return the exports object to the caller.
- **Module Caching:** Once a module is loaded, it is cached. This means that if you require the same module again, Node.js will not load the module from the file system but will return the cached version.
- **Built-in Modules:** Node.js comes with several built-in modules that you can use without installing them separately. Examples include fs for file system operations, http for creating HTTP servers, and path for working with file and directory paths.
- **Local and Third-Party Modules:** Apart from built-in modules, you can create your own local modules or install third-party modules via npm (Node Package Manager).

Create a module: Creating and using modules in Node.js is a fundamental part of the Node.js ecosystem. Here’s how you can create your own module and then access it in another file:

Step 1: Create a Module

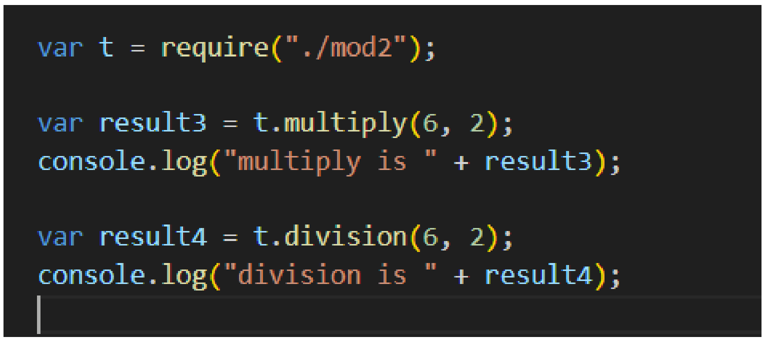
First, you need to create a JavaScript file that will act as a module.



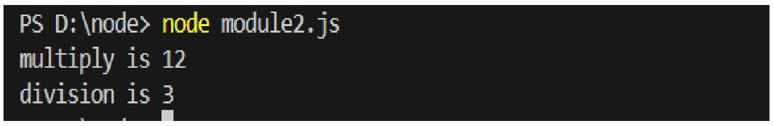
```javascript
exports.mult=function (x,y){
    return x*y;
};

exports.div=function (x,y){
    return x/y;
};
```

Step 2: Access the Module

To use the module we've created, we need to import it using the require function in another JavaScript file.



```javascript
var t = require("./mod2");

var result3 = t.multiply(6, 2);
console.log("multiply is " + result3);

var result4 = t.division(6, 2);
console.log("division is " + result4);
```

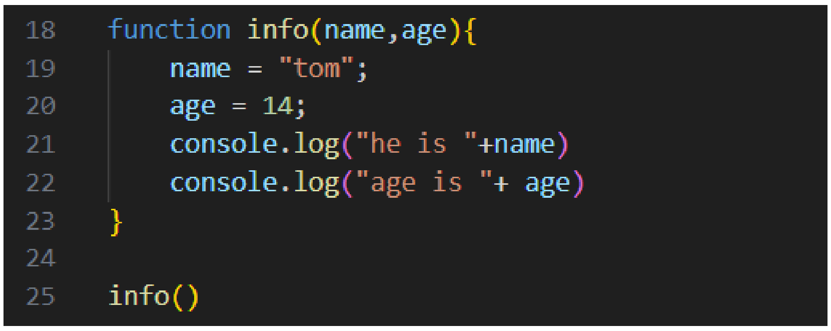


```
PS D:\node> node module2.js
multiply is 12
division is 3
```

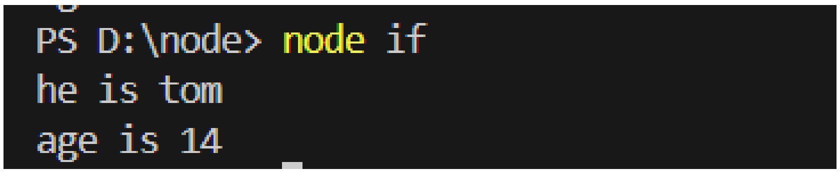
Q17. Write a Node JS program to use anonymous function.

The definition of the term "anonymous" is "unknown or without identification." An anonymous function in JavaScript is a function that has no name, or more precisely, one that lacks a name. An anonymous function has no identification when it is created. The distinction between a regular function and an anonymous function is this. Not just in JavaScript but in various other computer languages as well. An anonymous function serves the same purpose.

Example:



```
function info(name,age){
    name = "tom";
    age = 14;
    console.log("he is "+name)
    console.log("age is "+ age)
}

info()
```

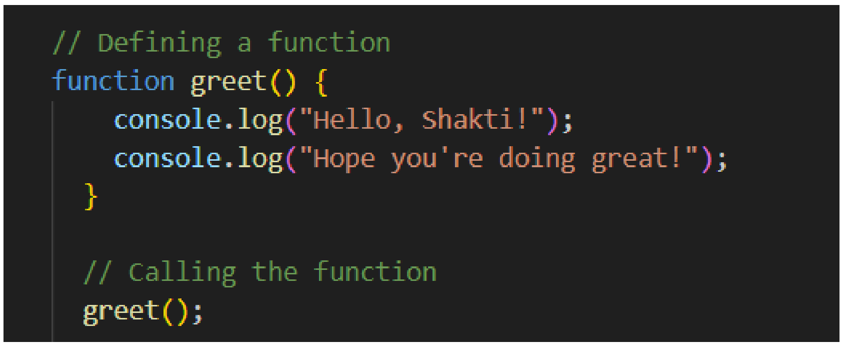


```
PS D:\node> node if
he is tom
age is 14
```

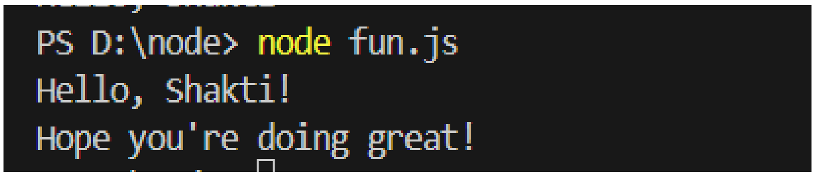
In this example, the anonymous function is stored in the info and we call it by invoking info(); .

Q18. Write a Node JS program to show any user defined function.

- **Simple User-Defined Function:** Below is an example of a simple user-defined function that displays a greeting message

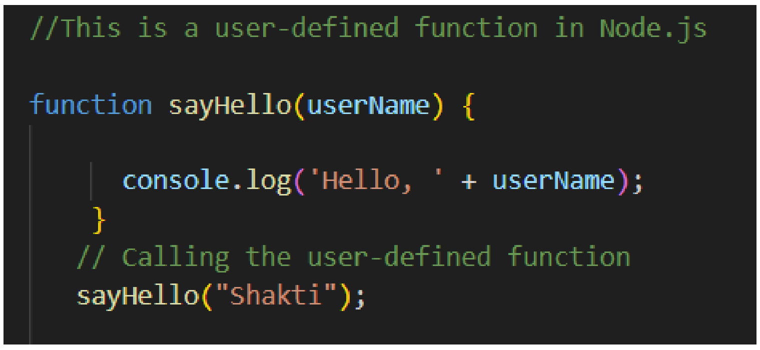


```
// Defining a function
function greet() {
    console.log("Hello, Shakti!");
    console.log("Hope you're doing great!");
  }

  // Calling the function
  greet();
```

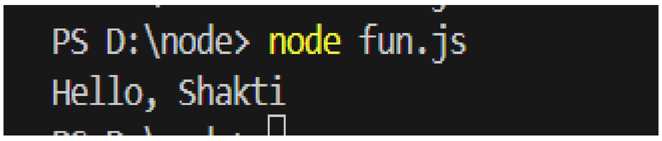


```
PS D:\node> node fun.js
Hello, Shakti!
Hope you're doing great!
```

- **User-Defined Function with Parameters**: You can also create functions that accept parameters. For instance, consider a function that takes a person’s name and prints a customized greeting:

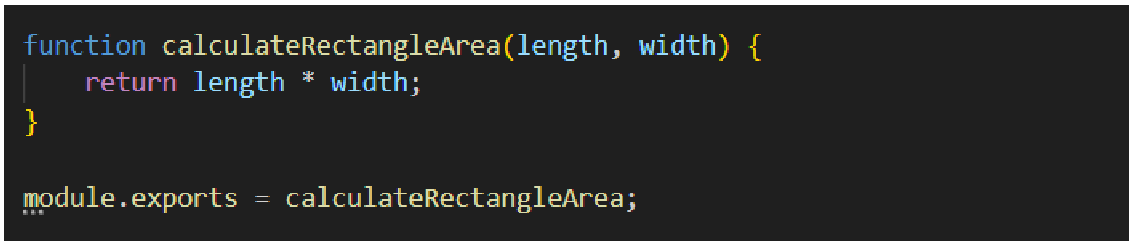


```
//This is a user-defined function in Node.js

function sayHello(userName) {

      console.log('Hello, ' + userName);
    }
   // Calling the user-defined function
   sayHello("Shakti");
```



```
PS D:\node> node fun.js
Hello, Shakti
```

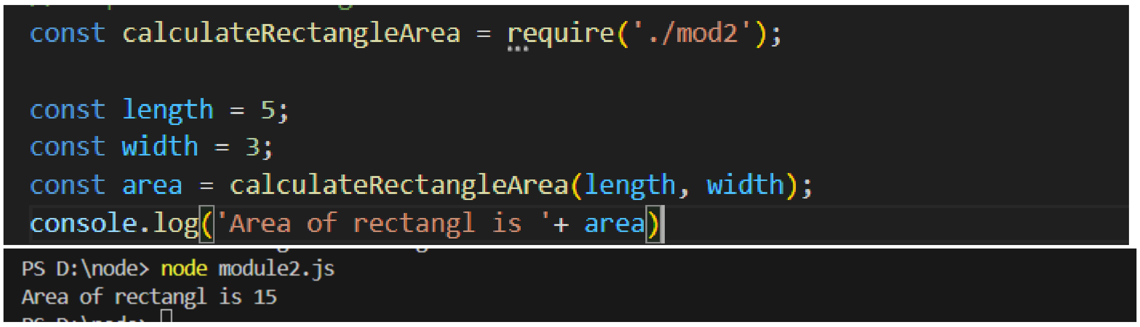
Q19. Write a Node JS program to use module execution.



```
function calculateRectangleArea(length, width) {
    return length * width;
}

module.exports = calculateRectangleArea;
```

In this module (rectangle.js):

- We define a function named calculateRectangleArea that takes two parameters: length and width.
- The function calculates the area of a rectangle by multiplying the length and width.
- We export this function using module.exports.



```
const calculateRectangleArea = require('./mod2');

const length = 5;
const width = 3;
const area = calculateRectangleArea(length, width);
console.log('Area of rectangl is '+ area)
```

```
PS D:\node> node module2.js
Area of rectangl is 15
```

- We use the require function to import the rectangle module.
- We call the calculateRectangleArea function with specific values for length and width.
- Finally, we log the calculated area to the console.

Q20. Explain types of module through example.

Node.js provides us with various modules to make our tasks easier and faster to perform. There are three types of Node.js modules.

Build-in Modules

Local Modules

Third-Party Modules

- **Core modules/Build-in Modules:**

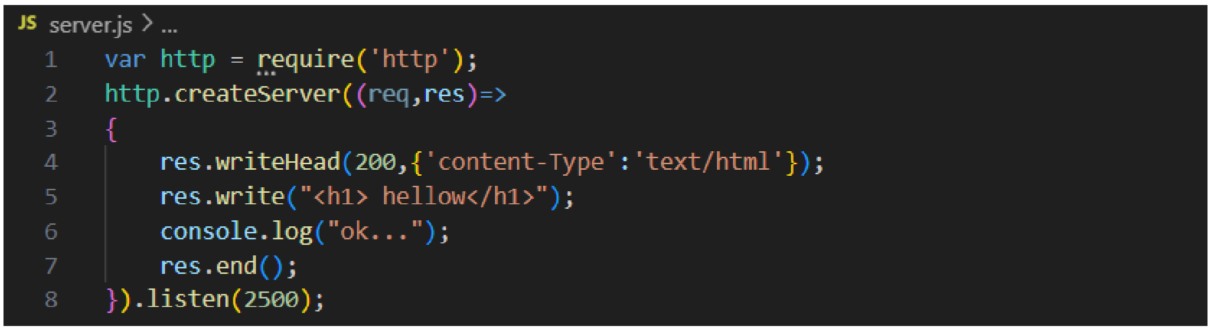
The core modules in Node.js are the built-in modules that are installed with the installation of Node to our system. We just use the required statement to access them.

Syntax: **let coreModule = require(modulename)**

coreModule: The variable in which we store the core module.

moduleName: The core module we want to import.

Example:



server.js > ...

```
var http = require('http');
http.createServer((req,res)=>
{
    res.writeHead(200,{'content-Type':'text/html'});
    res.write("<h1> hellow</h1>");
    console.log("ok...");
    res.end();
}).listen(2500);
```

- **Local modules:**

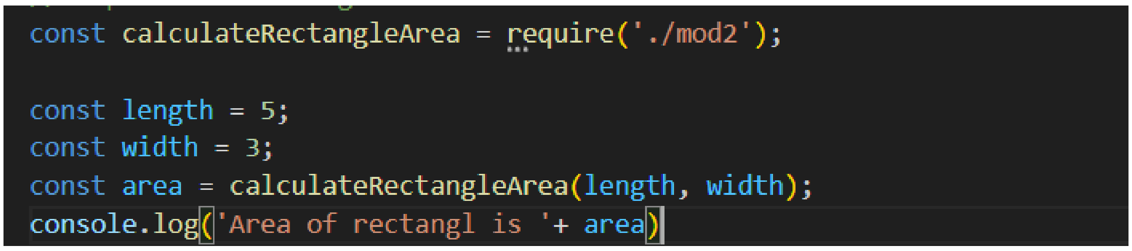
The local modules in Node.js are the modules that we create ourselves and make available to be used in our code. We define the functionality for these modules in a JavaScript file.

Syntax: **let localmodule = require(./local_modulename)**

localmodule: The variable in which we store the local module.

./local_modulename: The relative path to the JavaScript file that contains the local module.

Example:



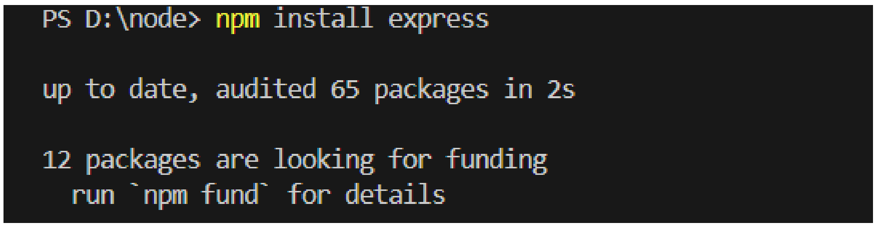
```
const calculateRectangleArea = require('./mod2');

const length = 5;
const width = 3;
const area = calculateRectangleArea(length, width);
console.log('Area of rectangl is '+ area)
```

In above example **('./mod2')** is local module or user define module.

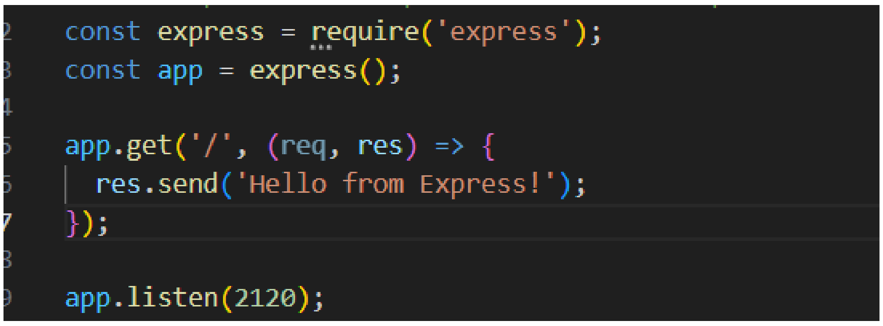
- **Third-party modules:**

The third-party modules in Node.js are those we install globally or on our local machines from external sources, typically via the Node package manager. You can install them in your project using NPM or YARN.

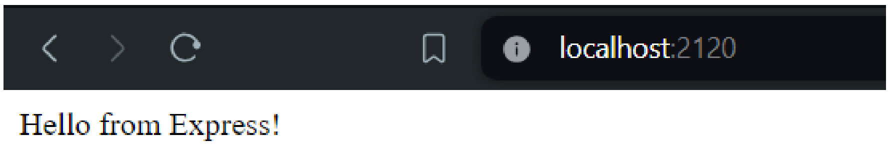
Example:



```
PS D:\node> npm install express

up to date, audited 65 packages in 2s

12 packages are looking for funding
  run `npm fund` for details
```



```
const express = require('express');
const app = express();

app.get('/', (req, res) => {
  res.send('Hello from Express!');
});

app.listen(2120);
```



```
localhost:2120
```

Hello from Express!

Q21. What is buffer. Explain its advantages through example.

Buffer:

Buffer is a temporary storage area in memory that holds data while it is being transferred from one place to another. Buffers are particularly useful when there is a difference in the rate at which data is received and the rate at which it can be processed. They act as an intermediate storage mechanism, allowing components with varying speeds to work together effectively.

JavaScript does not have a byte type data in its core API. To handle binary data Node.js includes a binary buffer implementation with a global module called Buffer.

Advantages of Buffers:

Synchronization:

- Buffers help synchronize data transfer between components that operate at different speeds.
- **For example**, When data is produced faster than it can be consumed, a buffer ensures that no data is lost. It acts as a bridge between the producer and the consumer.

Optimized I/O Operations:

- Buffers reduce the number of read and write operations, improving performance.
- Instead of performing frequent I/O calls, data is accumulated in the buffer and transferred in larger chunks.

Mitigating Speed Mismatches:

- When data generation and processing rates don't match (e.g., fast data production and slower consumption), buffers prevent data loss.
- The producer adds data to the buffer, and the consumer retrieves it at its own pace.

Customization:

- Developers can create custom buffers tailored to their application's needs.

Buffers allow fine-tuning for optimal performance.

Efficient memory allocation:

- Buffers allow for efficient memory allocation and deallocation, which is particularly useful when working with large amounts of binary data.

Flexible data handling:

- Buffers can be used to handle a wide range of data types, including strings, integers, and floating-point numbers, as well as binary data such as images and audio files.

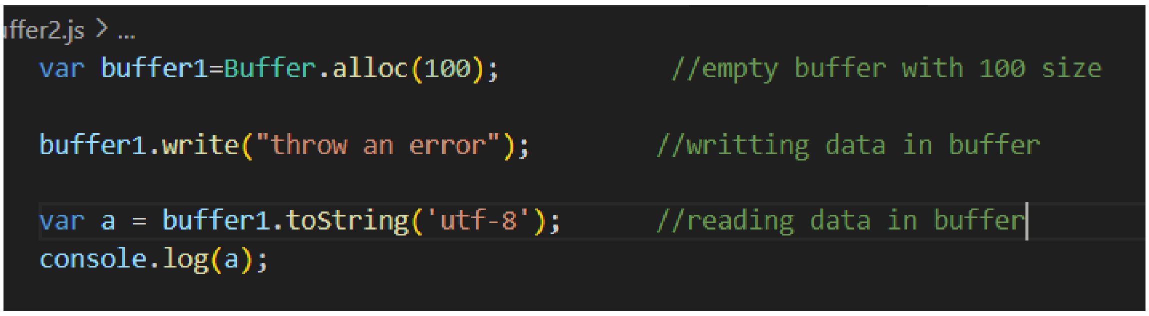
Improved performance:

- Buffers can improve the performance of your Node.js application by reducing the need for unnecessary conversions between different data types.

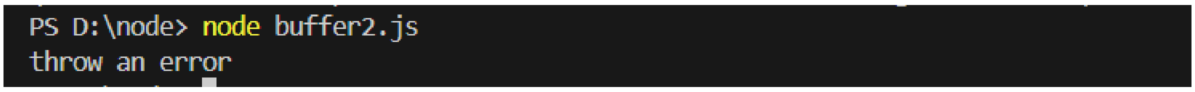
Better error handling:

- Buffers provide a way to handle errors and exceptions in a more robust and controlled manner, which is particularly important when working with binary data.

Example:



```
ffer2.js > ...
  var buffer1=Buffer.alloc(100);           //empty buffer with 100 size

  buffer1.write("throw an error");        //writting data in buffer

  var a = buffer1.toString('utf-8');      //reading data in buffer
  console.log(a);
```



```
PS D:\node> node buffer2.js
throw an error
```

The data is write asynchronously and stored in a buffer.

We convert the buffer to a string using data.toString() before displaying it.

Q22. Explain how to create a buffer and describes various its function through example.

In Node.js, a buffer is a block of memory allocated for storing binary data. To work with streams of binary data, Node.js provides the Buffer class, which is available globally and does not require importing a module.

Creating a Buffer:

A buffer is created using the Buffer.from(), Buffer.alloc(), and Buffer.allocUnsafe() methods.

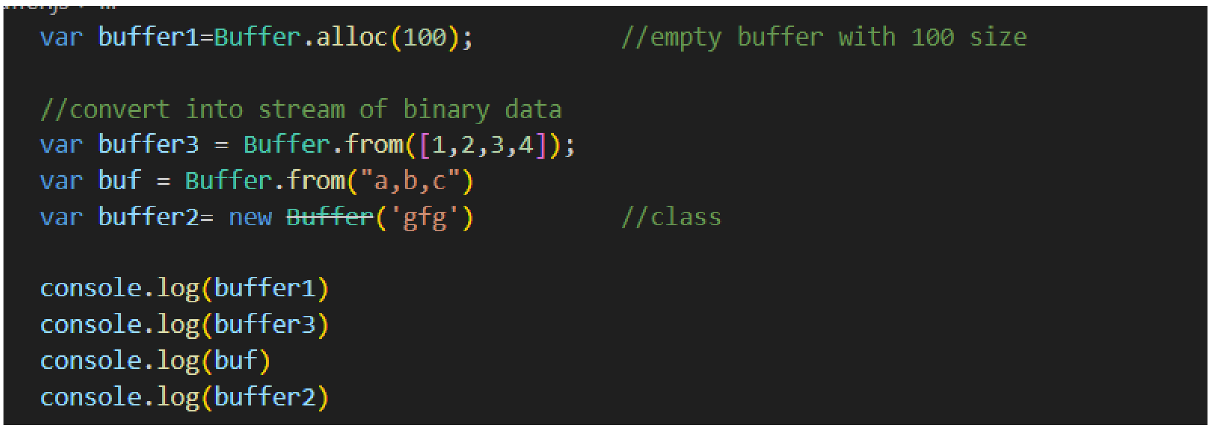
There are several ways to create a buffer in Node.js:

Buffer.alloc(size): Allocates a new buffer of size bytes long.

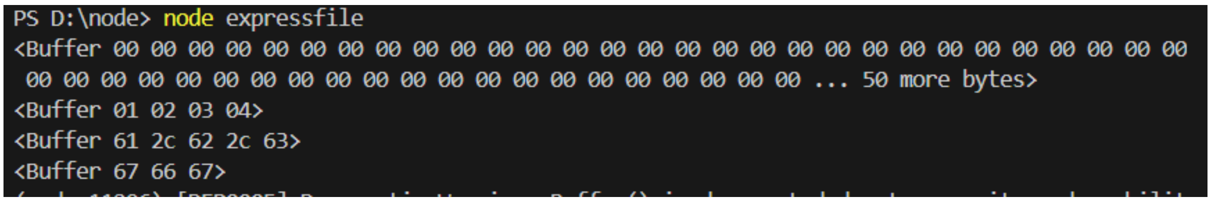
Buffer.from(array): Creates a new buffer containing the provided octets.

Buffer.from(buffer): Copies the passed buffer data into a new buffer instance.

Buffer.from(string[, encoding]): Creates a new buffer containing the provided string.



```
var buffer1=Buffer.alloc(100);          //empty buffer with 100 size

//convert into stream of binary data
var buffer3 = Buffer.from([1,2,3,4]);
var buf = Buffer.from("a,b,c")
var buffer2= new ~~Buffer~~('gfg')          //class

console.log(buffer1)
console.log(buffer3)
console.log(buf)
console.log(buffer2)
```



```
PS D:\node> node expressfile
<Buffer 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ... 50 more bytes>
<Buffer 01 02 03 04>
<Buffer 61 2c 62 2c 63>
<Buffer 67 66 67>
```

Function of Buffer:

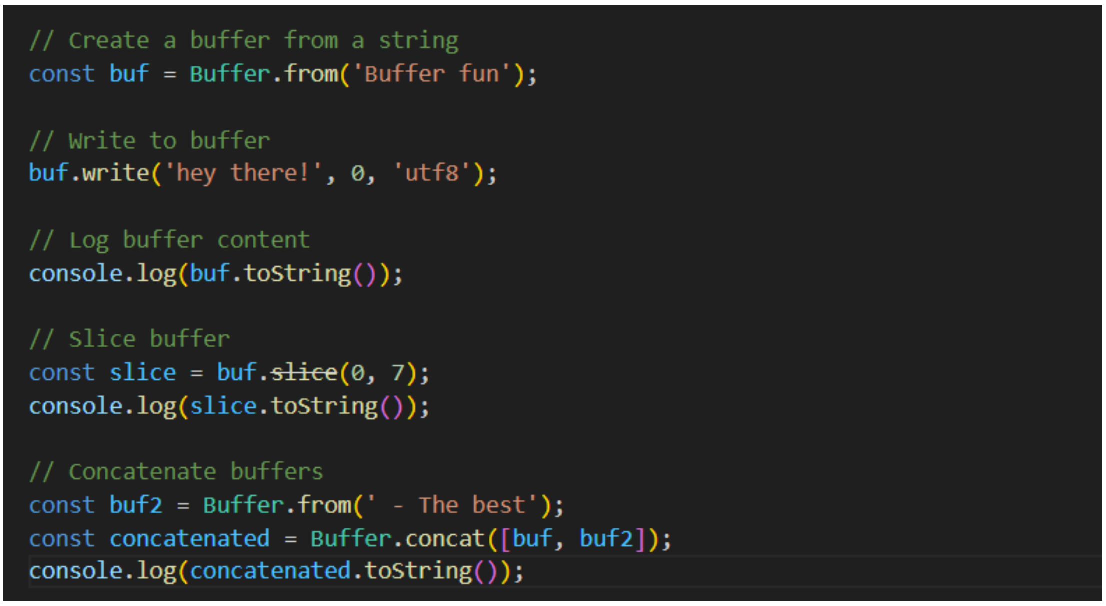
- **buf.write(string[, offset[, length]][, encoding]):** Writes string to the buffer at offset using the given encoding.

- **buf.toString([encoding[, start[, end]]]):** Decodes and returns a string from buffer data between start and end using the specified encoding.

- **buf.length:** Provides the size of the buffer in bytes.

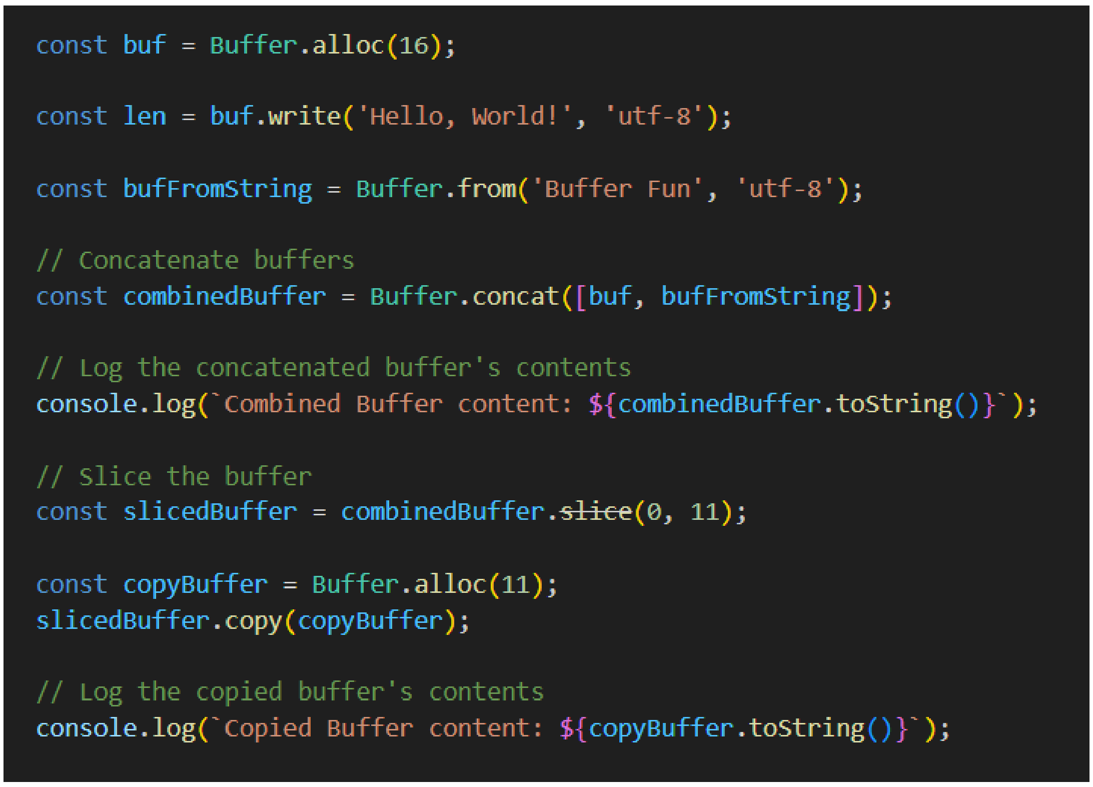
- **buf.slice([start[, end]]):** Returns a new buffer that references the same memory as the original but offset and cropped by the start and end indices.

- **Buffer.concat(list[, totalLength]):** Returns a new buffer which is the result of concatenating all the buffers in the list together.

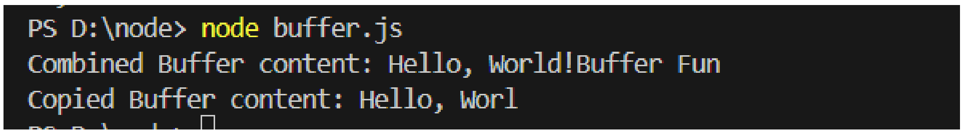


```
// Create a buffer from a string
const buf = Buffer.from('Buffer fun');

// Write to buffer
buf.write('hey there!', 0, 'utf8');

// Log buffer content
console.log(buf.toString());

// Slice buffer
const slice = buf.~~slice~~(0, 7);
console.log(slice.toString());

// Concatenate buffers
const buf2 = Buffer.from(' - The best');
const concatenated = Buffer.concat([buf, buf2]);
console.log(concatenated.toString());
```

Q23. Write a Node JS program to use buffer and their function.



```
const buf = Buffer.alloc(16);

const len = buf.write('Hello, World!', 'utf-8');

const bufFromString = Buffer.from('Buffer Fun', 'utf-8');

// Concatenate buffers
const combinedBuffer = Buffer.concat([buf, bufFromString]);

// Log the concatenated buffer's contents
console.log(`Combined Buffer content: ${combinedBuffer.toString()}`);

// Slice the buffer
const slicedBuffer = combinedBuffer.~~slice~~(0, 11);

const copyBuffer = Buffer.alloc(11);
slicedBuffer.copy(copyBuffer);

// Log the copied buffer's contents
console.log(`Copied Buffer content: ${copyBuffer.toString()}`);
```



```
PS D:\node> node buffer.js
Combined Buffer content: Hello, World!Buffer Fun
Copied Buffer content: Hello, Worl
```

In this program:

- We create a buffer buf with a size of 16 bytes using Buffer.alloc.
- We write the string ‘Hello, World!’ to buf using the write method.
- We create another buffer bufFromString from the string ‘Buffer Fun’.
- We concatenate buf and bufFromString using Buffer.concat and log the result.
- We slice the combined buffer to get the first 11 characters using slice and log the result.
- We copy the sliced buffer into copyBuffer using copy and log the result.

Q24. Explain createserver function in Node JS terms of definition, syntax and example.

The createServer function is a method provided by the http module to create an HTTP server.

The createServer method turns your computer into an HTTP server. It creates an HTTP Server object that can listen to ports on your computer and execute a function, known as a requestListener, each time a request is made to the server.

HTTP, it is an in-build module which is pre-installed along with NodeJS. It is used to create server and set up connections. Using this connection, data sending and receiving can be done as long as connections use a hypertext transfer protocol.

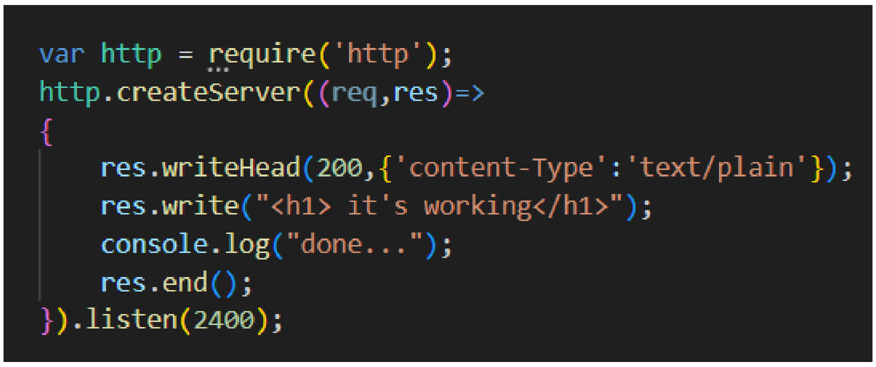
Syntax: **const http = require('http');**

http.createServer([requestListener])

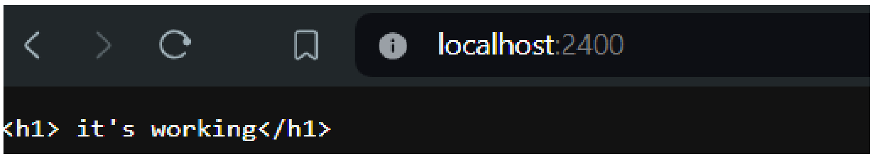
.listen(port_no, callback);

- **requestListener** is an optional parameter. It specifies a function to be executed every time the server gets a request. This function handles requests from the user, as well as responses back to the user.
- **PORT:** A port number that our server will listen on.
- **callback():** After createserver() creates a server and our function listens on the specified port, the callback will be executed.

Example:



```
var http = require('http');
http.createServer((req,res)=>
{
    res.writeHead(200,{'content-Type':'text/plain'});
    res.write("<h1> it's working</h1>");
    console.log("done...");
    res.end();
}).listen(2400);
```



localhost:2400

```
<h1> it's working</h1>
```

Q25. Write a step of creating a server in Node JS and justify each step to example.

Step1: Import the http Module:

Justification: We need the http module to create an HTTP server.

Example:

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

Step2: Create an HTTP Server:

Justification: We create an instance of an HTTP server using the createServer method.

Example:

```
const server = http.createServer((req, res) => { });
```

Step3: Define the Request Listener (Request Handling Logic):

Justification: The request listener function is executed every time a request is made to the server. It handles incoming requests and sends appropriate responses.

Example:

```
server.on('request', (req, res) => {
res.writeHead(200, { 'Content-Type': 'text/plain' });
res.write('NodeJS server program execution');
res.end( );  });
```

Step4: Set the Server to Listen on a Port:

Justification: We specify the port number on which the server should listen for incoming requests.

Example:

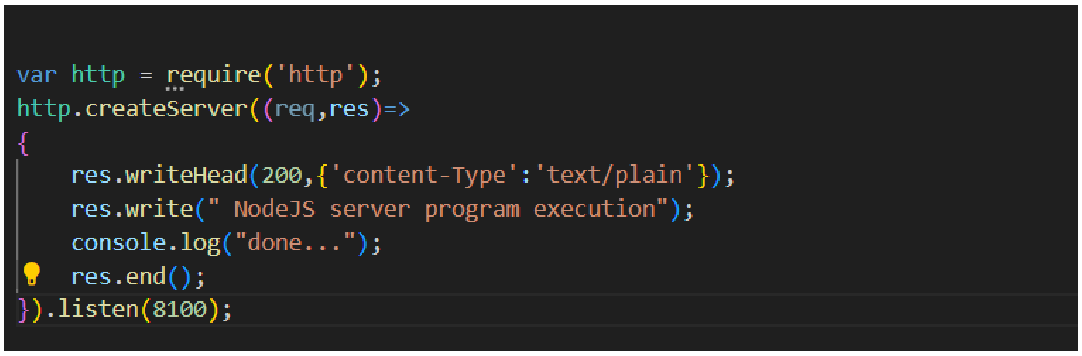
```
server.listen(8100);
```

Justification: The server starts listening for incoming requests once we call the listen method.

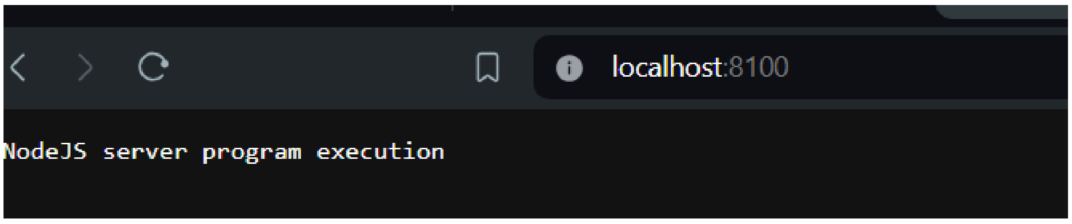
Step5: Access the Server in a Web Browser:

Justification: Open a web browser and navigate to the specified URL (e.g., http://localhost:8100/) to see the server response.

Access http://localhost:8100/ in your browser to view the **"NodeJS server program execution**" message.

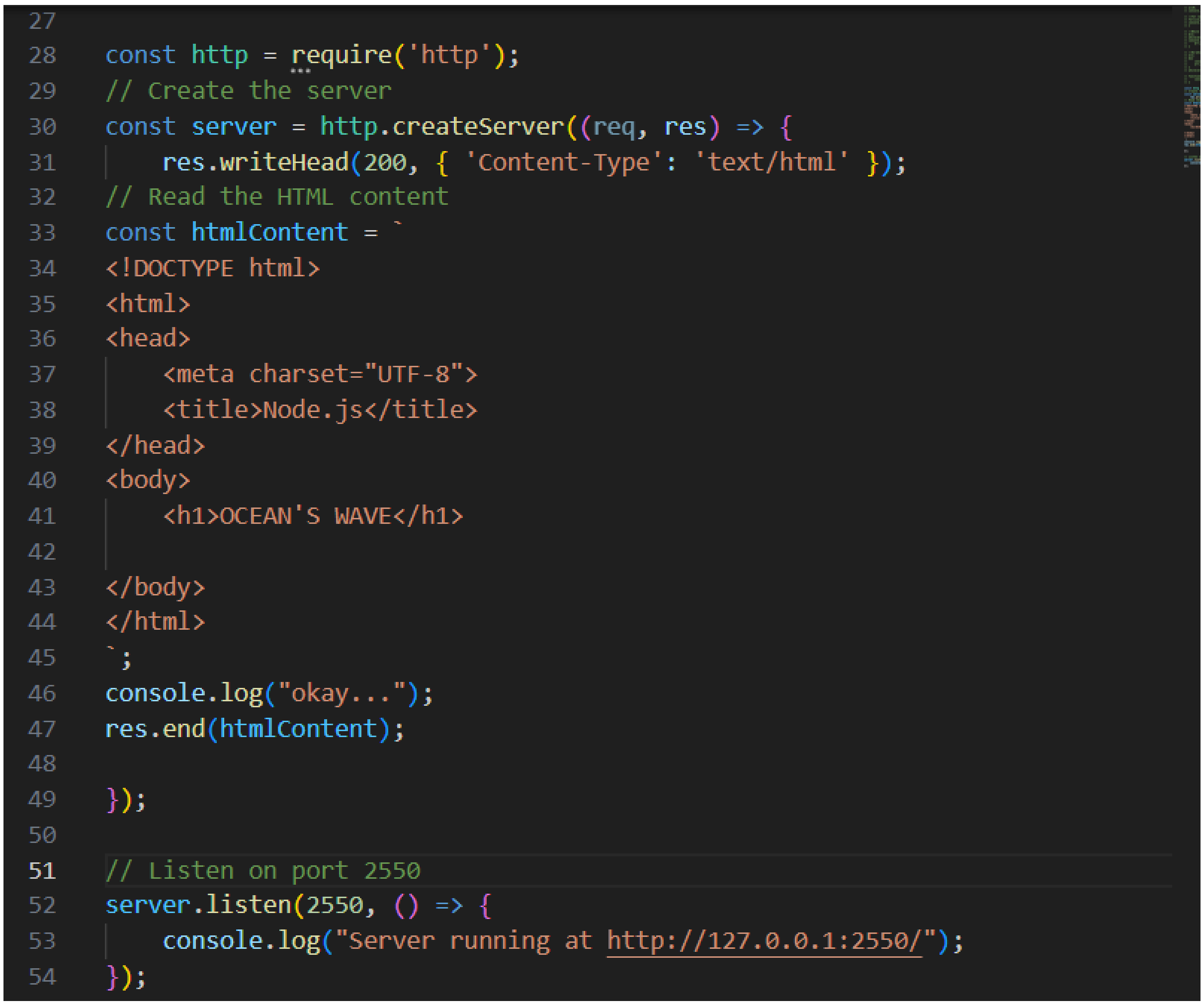


```
var http = require('http');
http.createServer((req,res)=>
{
    res.writeHead(200,{'content-Type':'text/plain'});
    res.write(" NodeJS server program execution");
    console.log("done...");
    res.end();
}).listen(8100);
```

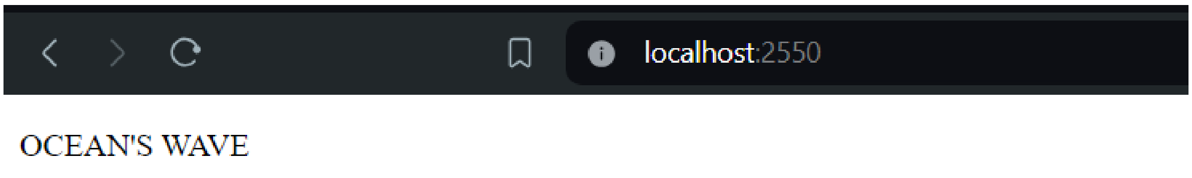


```
NodeJS server program execution
```

Q26. Write a Node JS program to handle response as html code inside a server construction.



```
const http = require('http');
// Create the server
const server = http.createServer((req, res) => {
    res.writeHead(200, { 'Content-Type': 'text/html' });
// Read the HTML content
const htmlContent = `
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Node.js</title>
</head>
<body>
    <h1>OCEAN'S WAVE</h1>

</body>
</html>
`;
console.log("okay...");
res.end(htmlContent);

});

// Listen on port 2550
server.listen(2550, () => {
    console.log("Server running at http://127.0.0.1:2550/");
});
```



localhost:2550

OCEAN'S WAVE

In this Example:

- We define an HTML content string (htmlContent) that will be sent as the response.
- The server responds with this HTML content when accessed.
- The Content-Type header is set to 'text/html' to inform the browser how to interpret the response.

Q27. Write a Node JS program to handle response as server massage inside a server construction.

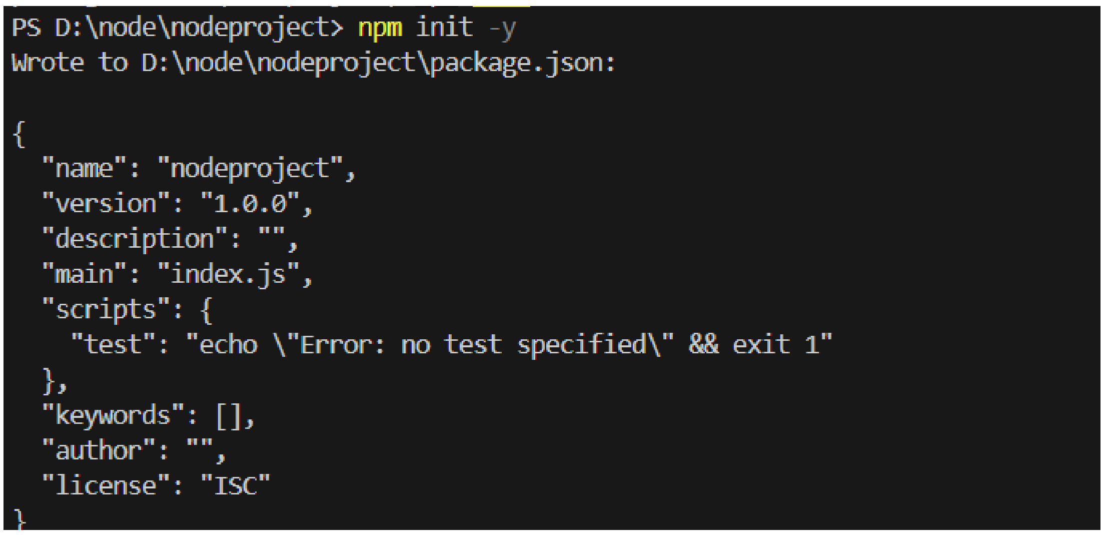
create a simple Node.js program that handles server responses within a server construction. We'll use the popular Express.js framework to set up our server and demonstrate how to handle responses.

Setting Up the Project:

- First, create a new folder for your project (nodeproject).
- Open a terminal and navigate to the project folder.
- Run the following command to initialize a new Node.js project:

npm init -y

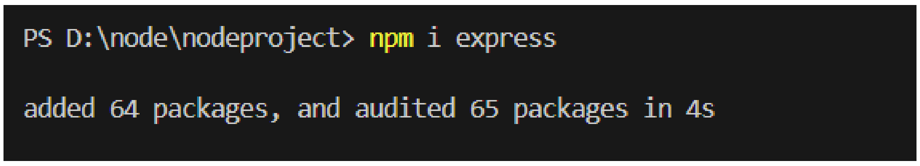
- This will create a package.json file with default settings.



```
PS D:\node\nodeproject> npm init -y
Wrote to D:\node\nodeproject\package.json:

{
  "name": "nodeproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Installing Dependencies:

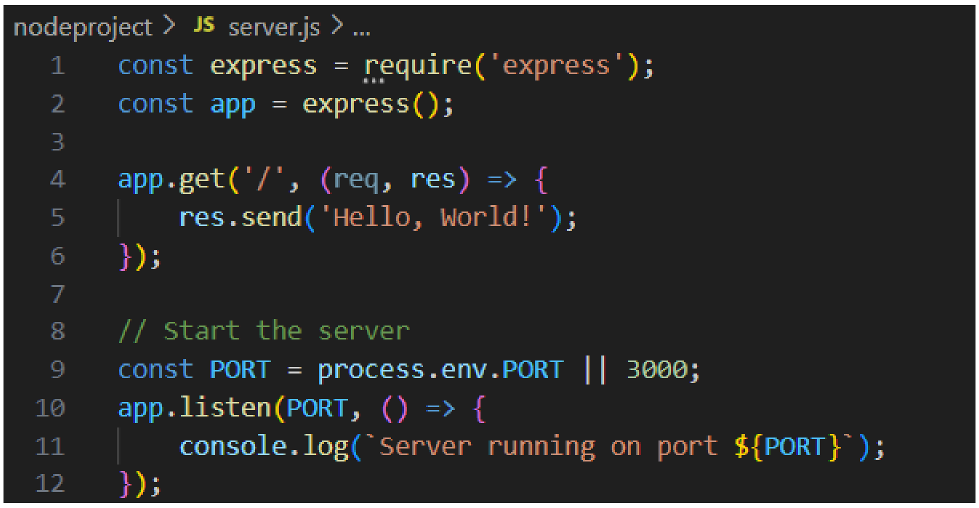
- Install the express package, which we'll use to create our server:



```
PS D:\node\nodeproject> npm i express

added 64 packages, and audited 65 packages in 4s
```

Creating the Express Server:

- Create a file named server.js (or any other name you prefer) in your project folder.
- Add the following code to set up a basic Express server:

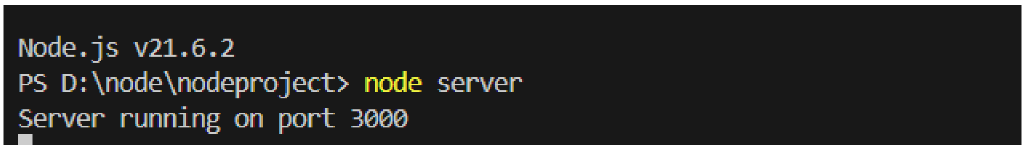


nodeproject > JS server.js > ...

```javascript
const express = require('express');
const app = express();

app.get('/', (req, res) => {
    res.send('Hello, World!');
});

// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

Running the Server:

In your terminal, run the following command to start the server:

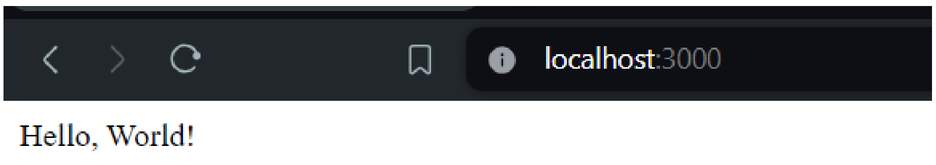


```
Node.js v21.6.2
PS D:\node\nodeproject> node server
Server running on port 3000
```

- You can see the message "Server running on port 3000" indicating that your server is up and running.

Testing the Response:

- Open a web browser or use a tool like curl to make a request to http://localhost:3000.
- You'll receive the response "Hello, World!" as defined in our route handler.



localhost:3000

Hello, World!

Q28. Explain different type of network status code and content type.

Network Status Codes (HTTP Status Codes):

HTTP status codes are standardized responses that indicate the outcome of an HTTP request.

They fall into several categories:

Informational Responses (100–199):

These codes provide information about the request but don’t necessarily indicate success or failure.

Examples:

- 100 Continue: The server acknowledges the request and expects the client to continue sending the rest of the data.
- 101 Switching Protocols: The server is switching to a different protocol (e.g., from HTTP to WebSocket).

Successful Responses (200–299):

These codes indicate that the request was successful.

Examples:

- 200 OK: The request was successful, and the server is returning the requested data.
- 201 Created: A new resource was successfully created (e.g., after a POST request).

Redirection Messages (300–399):

These codes indicate that the client must take additional action to complete the request.

Examples:

- 301 Moved Permanently: The requested resource has moved to a different URL.
- 302 Found (Moved Temporarily): Similar to 301 but indicates a temporary move.

Client Error Responses (400–499):

These codes indicate that there was an error on the client side (e.g., invalid request, authentication failure).

Examples:

- 400 Bad Request: The server cannot process the request due to a client error.
- 401 Unauthorized: Authentication is required, and the client failed to provide valid credentials.

Server Error Responses (500–599):

These codes indicate that there was an error on the server side.

Examples:

- 500 Internal Server Error: A generic error occurred on the server.
- 503 Service Unavailable: The server is temporarily unable to handle requests.

Content Types:

Content types (also known as MIME types) specify the format of the data being sent over the network. They are essential for proper communication between clients and servers. Here are some common content types:

- **text/plain:** Represents plain text data (e.g., .txt files).
- **text/html:** Represents HTML content (e.g., web pages).
- **application/json:** Represents JSON data (commonly used for APIs).
- **application/xml:** Represents XML data (used for structured documents).
- **image/jpeg, image/png:** Represent image files.
- **application/pdf:** Represents PDF documents.
- **application/octet-stream:** Represents binary data (e.g., file downloads).

In Node.js, you can set the content type in the response headers using res.setHeader('Content-Type', '...') before sending the response.

Q29. Explain writeHead, write and end function in Node JS terms of definition, syntax and example.

writeHead function:

- The response.writeHead() method, part of the http module, is used to send a response header to an incoming request.
- It allows you to set the HTTP status code, response headers, and an optional human-readable status message.
- The statusCode represents a 3-digit HTTP status code (e.g., 200, 404).
- The headers parameter contains additional response headers (as an object).
- Optionally, you can provide a statusMessage as the second argument.
- If you haven't used response.setHeader() prior to calling response.writeHead(), it directly writes the supplied headers to the network channel, bypassing internal caching.
- For progressive header population with future retrieval and modification, prefer using response.setHeader().

Syntax: **response.writeHead(statusCode[, statusMessage][, headers]);**

write function:

- The response.write() method sends data (a chunk) to the client.
- You can call response.write() multiple times to send data in chunks.
- The encoding parameter specifies the character encoding (default is 'utf8').
- The optional callback function is called when the data has been fully flushed.

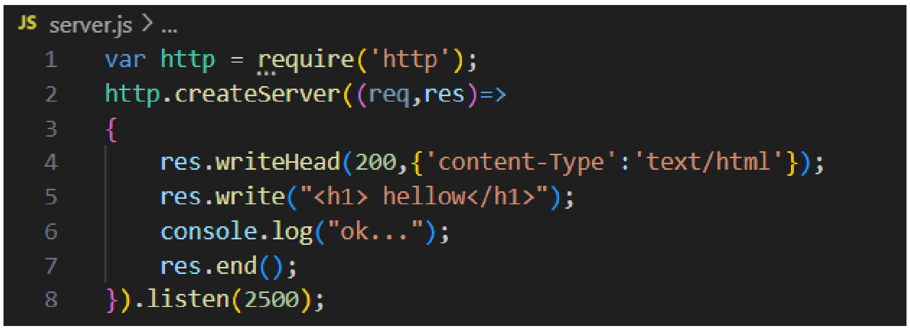
Syntax: **response.write(chunk[, encoding][, callback]);**

end function:

- The response.end() method signals the end of the response.
- You can optionally provide data to be sent as the last chunk.
- The encoding parameter specifies the character encoding (default is 'utf8').
- The optional callback function is called when the response has been fully sent.

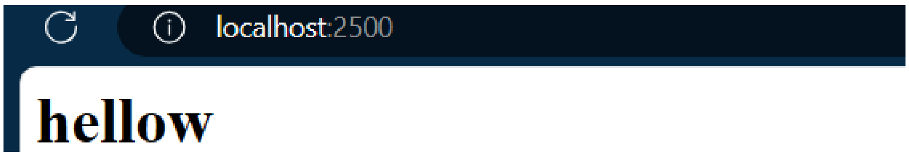
Syntax: **response.end([data][, encoding][, callback]);**

Example:



server.js > ...

```
var http = require('http');
http.createServer((req,res)=>
{
    res.writeHead(200,{'content-Type':'text/html'});
    res.write("<h1> hellow</h1>");
    console.log("ok...");
    res.end();
}).listen(2500);
```



localhost:2500

hellow

Q30. Explain how to handle a client side request and server response through Node JS.

Setting Up a Basic Server:

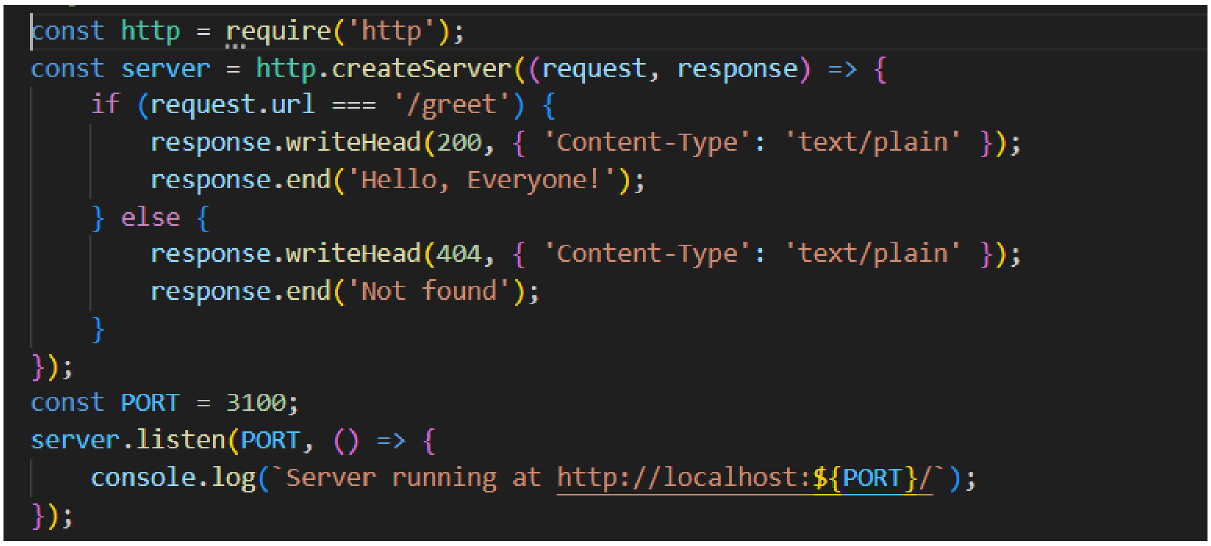
- To create a server in Node.js, you can use the built-in http module.
- First, import the http module and create a server using http.createServer().
- Define a request listener function that handles incoming HTTP requests and returns an HTTP response.

Handling Requests:

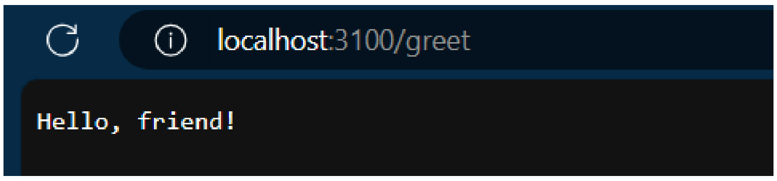
- The request listener function receives two arguments: request and response.
- You can inspect the request object to determine the type of request (e.g., GET, POST) and extract relevant data (e.g., query parameters, headers).
- Based on the request, you can perform different actions (e.g., read files, query a database, process data).

Sending Responses:

- Use the response object to send data back to the client.
- response.writeHead(statusCode, headers).
- Write the response body using response.write(data) (you can call this method multiple times for streaming data).
- Finally, end the response using response.end([data])

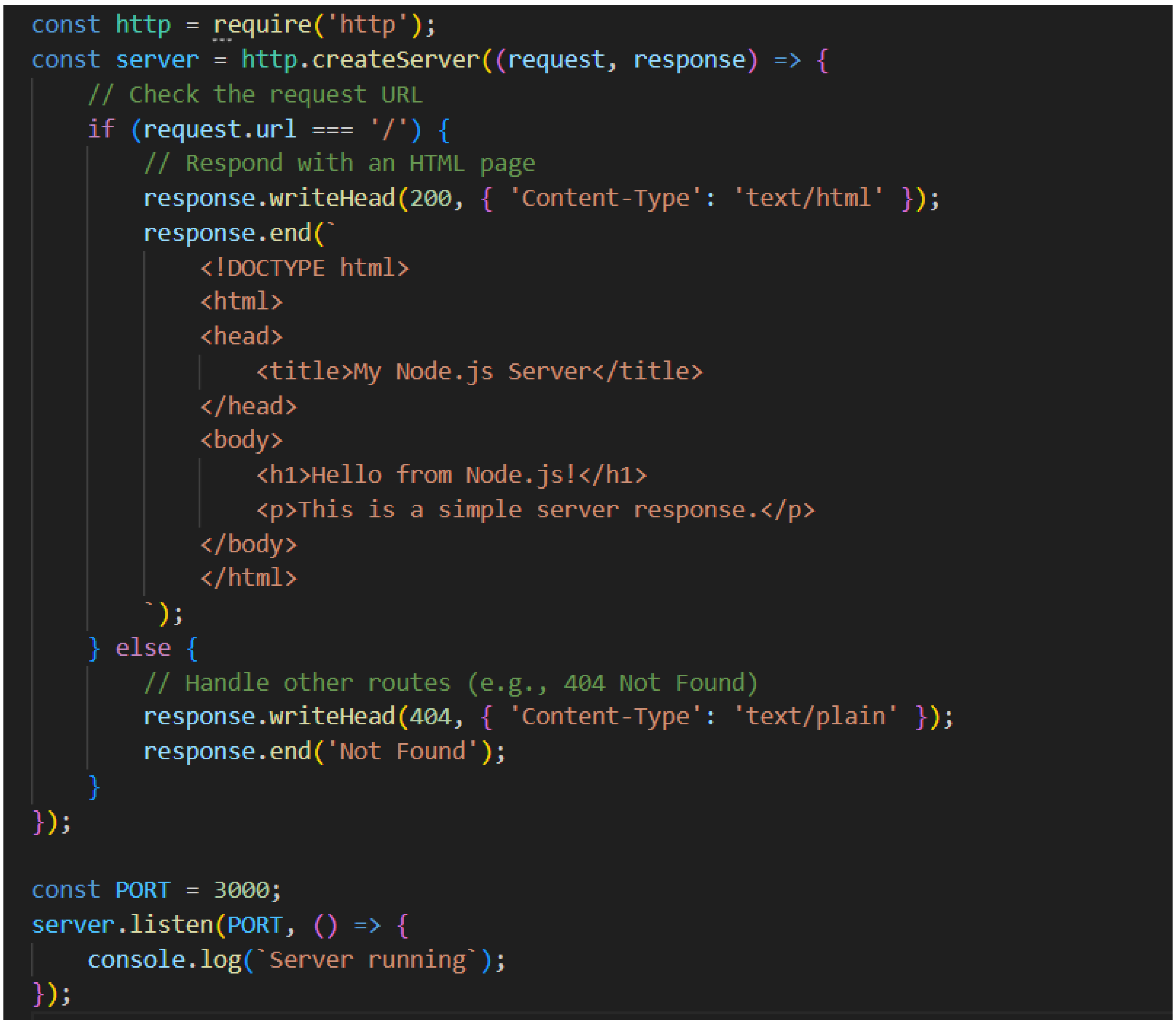


```javascript
const http = require('http');
const server = http.createServer((request, response) => {
    if (request.url === '/greet') {
        response.writeHead(200, { 'Content-Type': 'text/plain' });
        response.end('Hello, Everyone!');
    } else {
        response.writeHead(404, { 'Content-Type': 'text/plain' });
        response.end('Not found');
    }
});
const PORT = 3100;
server.listen(PORT, () => {
    console.log(`Server running at http://localhost:${PORT}/`);
});
```

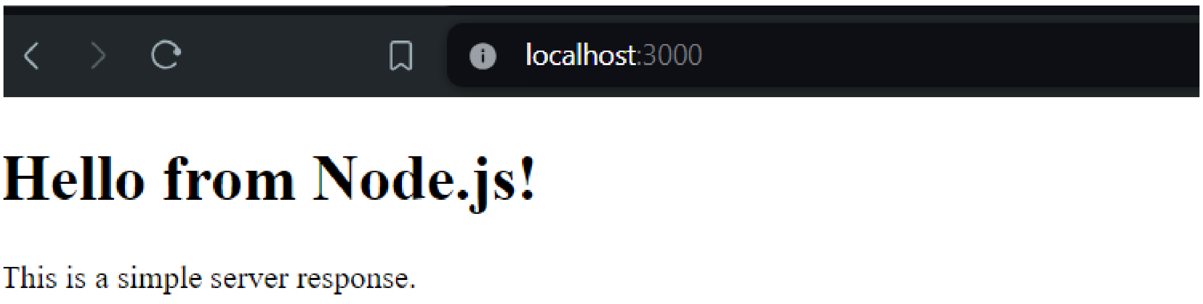


```
Hello, friend!
```

Q31. Write a Node JS code to handle client request as page directive and server response as html code or server massage.



```
const http = require('http');
const server = http.createServer((request, response) => {
    // Check the request URL
    if (request.url === '/') {
        // Respond with an HTML page
        response.writeHead(200, { 'Content-Type': 'text/html' });
        response.end(`
            <!DOCTYPE html>
            <html>
            <head>
                <title>My Node.js Server</title>
            </head>
            <body>
                <h1>Hello from Node.js!</h1>
                <p>This is a simple server response.</p>
            </body>
            </html>
        `);
    } else {
        // Handle other routes (e.g., 404 Not Found)
        response.writeHead(404, { 'Content-Type': 'text/plain' });
        response.end('Not Found');
    }
});

const PORT = 3000;
server.listen(PORT, () => {
    console.log(`Server running`);
});
```

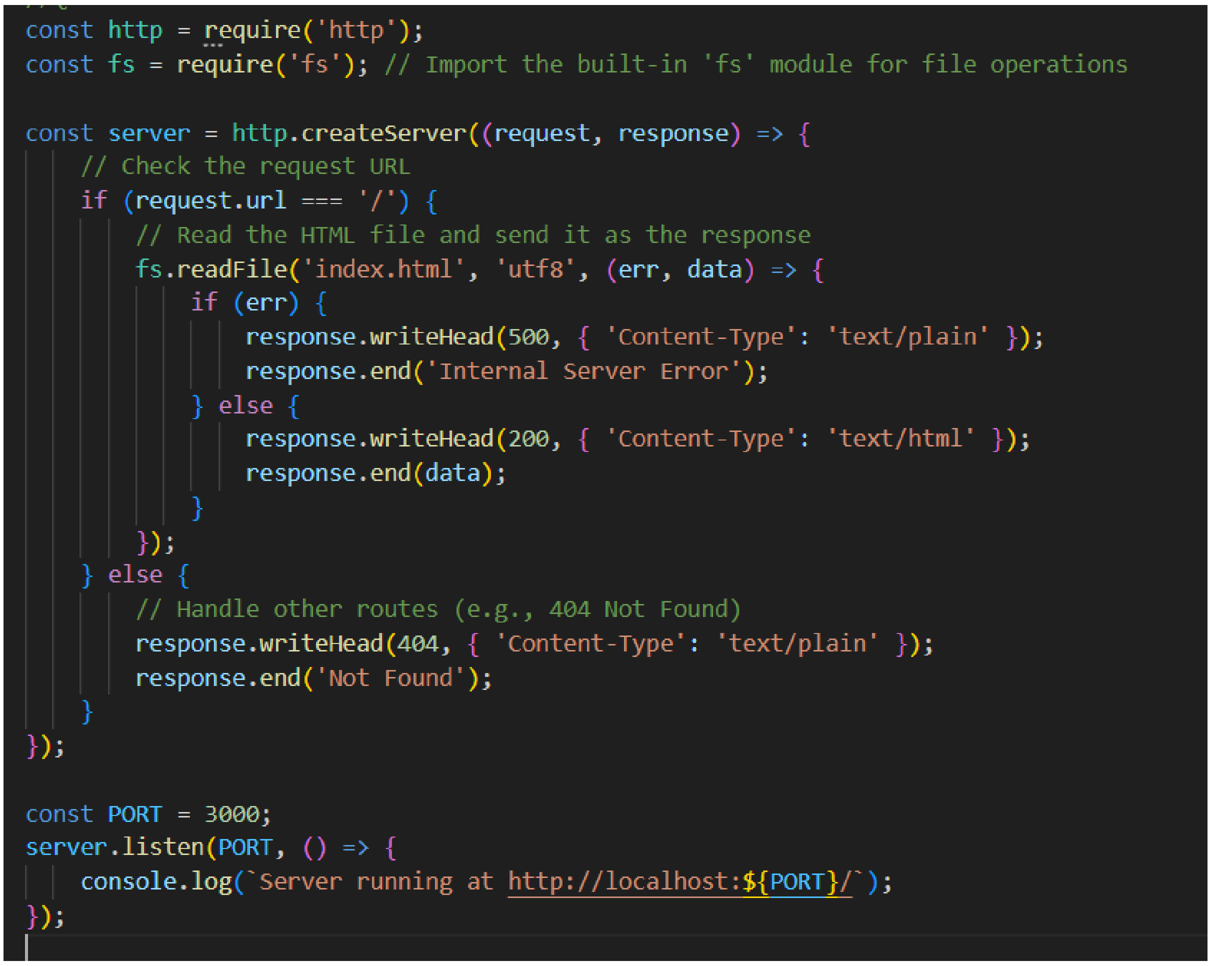


localhost:3000

Hello from Node.js!

This is a simple server response.

Q32. Write a Node JS code to handle client request as page directive and server response as html file.



```
const http = require('http');
const fs = require('fs'); // Import the built-in 'fs' module for file operations

const server = http.createServer((request, response) => {
    // Check the request URL
    if (request.url === '/') {
        // Read the HTML file and send it as the response
        fs.readFile('index.html', 'utf8', (err, data) => {
            if (err) {
                response.writeHead(500, { 'Content-Type': 'text/plain' });
                response.end('Internal Server Error');
            } else {
                response.writeHead(200, { 'Content-Type': 'text/html' });
                response.end(data);
            }
        });
    } else {
        // Handle other routes (e.g., 404 Not Found)
        response.writeHead(404, { 'Content-Type': 'text/plain' });
        response.end('Not Found');
    }
});

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

In this example:

- When the client requests the root URL (/), the server reads the contents of an index.html file and sends it as the response.
- If the file read operation encounters an error, the server responds with a 500 Internal Server Error.
- For any other route, the server responds with a “Not Found” message.

Q33. Explain following function through advantages, syntax and example.

a) writefile b) writefilesyn c) readfile d) readfilesyn

writeFile and writeFileSync:

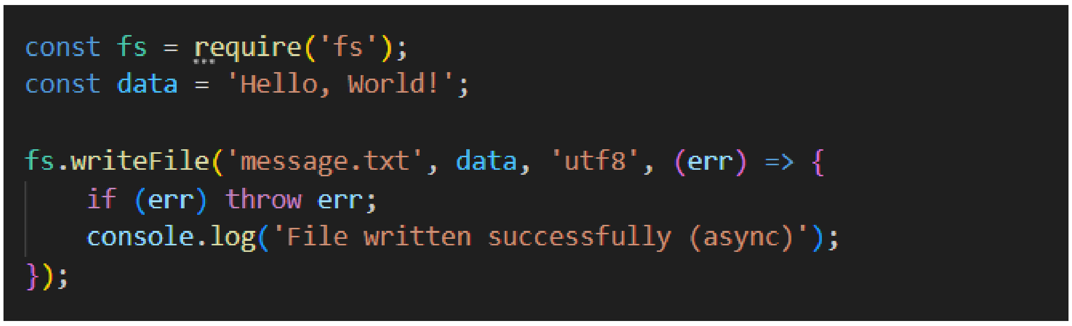
- These functions are used to write data to a file asynchronously (non-blocking) and synchronously (blocking), respectively.
- They create a new file or overwrite an existing file with the specified data.
- Both methods allow you to specify the file path, data to be written, and optional parameters (such as encoding and file mode).

Advantages:

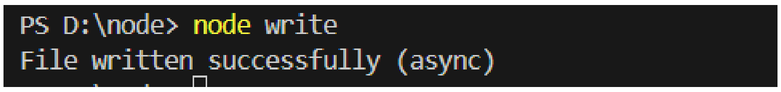
Asynchronous (writeFile):

- Non-blocking: Does not block the main thread, allowing other operations to continue.
- Ideal for handling multiple requests concurrently.

Syntax: **fs.writeFile(file, data, [options], callback);**



```
const fs = require('fs');
const data = 'Hello, World!';

fs.writeFile('message.txt', data, 'utf8', (err) => {
    if (err) throw err;
    console.log('File written successfully (async)');
});
```

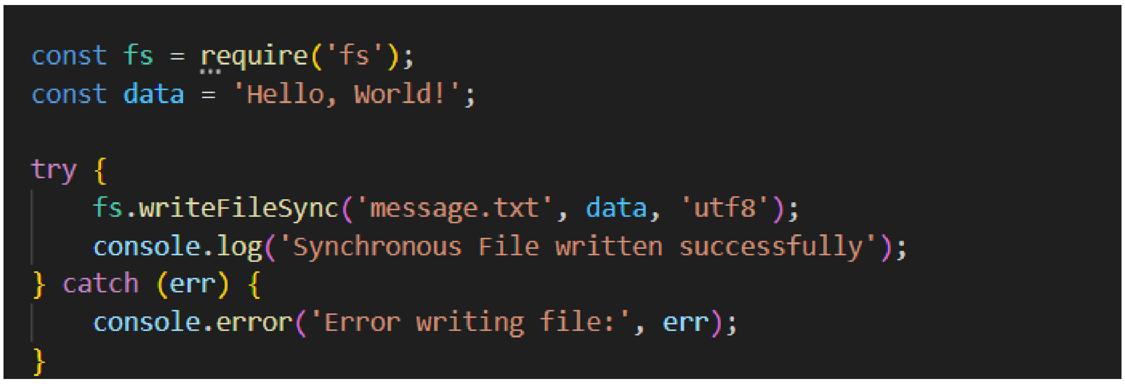


```
PS D:\node> node write
File written successfully (async)
```

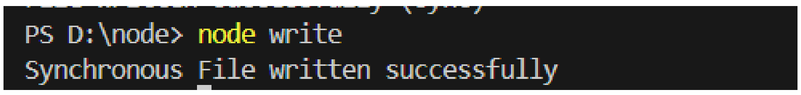
Synchronous (writeFileSync):

- Simplicity: Easier to use for simple scripts or debugging.
- Blocking: Should be used cautiously to avoid blocking the event loop.

Syntax: **fs.writeFileSync(file, data, [option]);**



```
const fs = require('fs');
const data = 'Hello, World!';

try {
    fs.writeFileSync('message.txt', data, 'utf8');
    console.log('Synchronous File written successfully');
} catch (err) {
    console.error('Error writing file:', err);
}
```



```
PS D:\node> node write
Synchronous File written successfully
```

readFile and readFileSync:

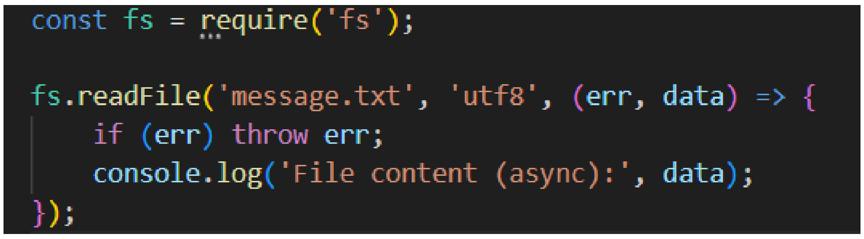
- These functions read data from a file asynchronously (non-blocking) and synchronously (blocking), respectively.
- They allow you to retrieve the contents of an existing file.
- You can specify the file path and optional parameters (such as encoding).

Advantages:

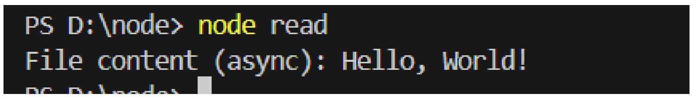
Asynchronous (readFile):

- Non-blocking: Does not block the event loop.
- Suitable for handling multiple read requests concurrently.

Syntax: **fs.readFile(file, [options], callback);**



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

fs.readFile('message.txt', 'utf8', (err, data) => {
    if (err) throw err;
    console.log('File content (async):', data);
});
```

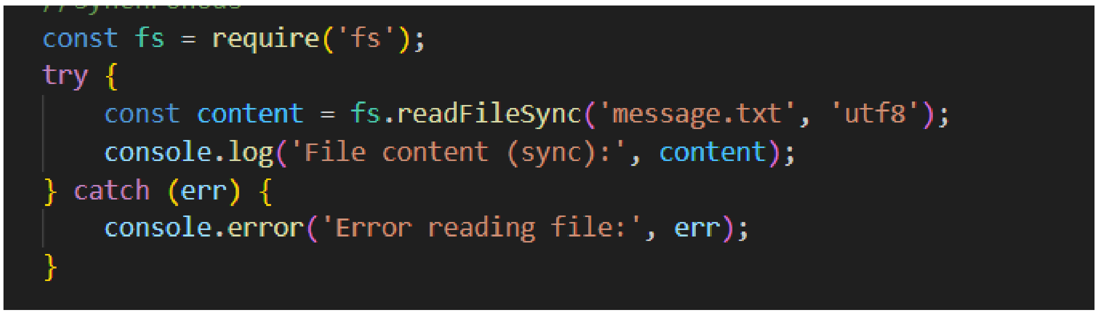


```
PS D:\node> node read
File content (async): Hello, World!
```

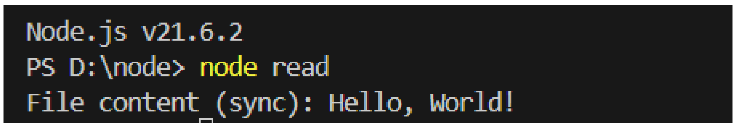
Synchronous (readFileSync):

- Simplicity: Useful for simple scripts or small-scale applications.
-

Syntax: **fs.readFileSync(file, [options]);**



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



```
Node.js v21.6.2
PS D:\node> node read
File content (sync): Hello, World!
```

Q34. Write any five difference between synchronous and asynchronous function.

i. **Execution Flow:**

Synchronous:

- Executes sequentially, blocking the program until the current operation completes.
- Each operation waits for the previous one to finish.

Asynchronous:

- Executes non-sequentially, allowing other operations to proceed while waiting for a task to complete.
- Does not block the program; continues executing subsequent code.

ii. **Blocking vs. Non-Blocking:**

Synchronous:

- Blocks the event loop until the operation finishes.
- Can lead to poor performance if used extensively.

Asynchronous:

- Non-blocking; allows other tasks to run concurrently.
- Ideal for I/O-bound operations (e.g., reading files, making network requests).

iii. **Callback Mechanism:**

Synchronous:

- No explicit callbacks needed; functions return results directly.

Asynchronous:

- Requires callbacks or promises to handle results.
- Callbacks are executed when the operation completes.

iv. **Error Handling:**

Synchronous:

- Errors can be caught using try-catch blocks.
- Errors immediately propagate up the call stack.

Asynchronous:

- Errors are typically handled in callbacks or promise .catch() blocks.
- Errors do not block subsequent code execution.

v. **Use Cases:**

Synchronous:

- Useful for CPU-bound tasks (e.g., mathematical computations).
- Simple to reason about and debug.

Asynchronous:

- Ideal for I/O-bound tasks (e.g., reading/writing files, making API calls).
- Allows efficient handling of multiple requests concurrently.

Q35. Write a steps to synchronously reading and writing file through example.

Import the fs module:

- Ensure you have Node.js installed on your system.
- Import the built-in fs (file system) module, which provides file-related functions.

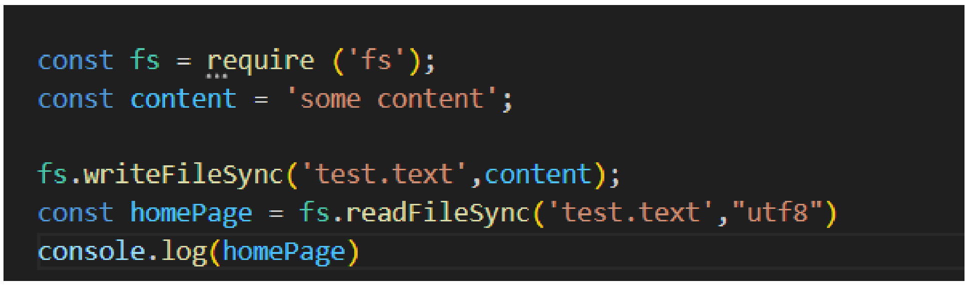
Reading a File Synchronously:

- Use the fs.readFileSync() function to read the contents of a file synchronously.
- Provide the file path and an optional encoding (e.g., 'utf8' for text files).

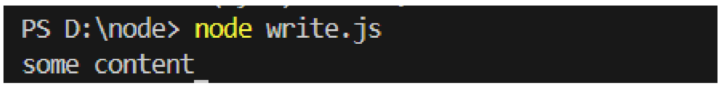
Writing to a File Synchronously:

- Use the fs.writeFileSync() function to write data to a file synchronously.
- Provide the file path, data to be written, and an optional encoding.

Example:



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

fs.writeFileSync('test.text',content);
const homePage = fs.readFileSync('test.text',"utf8")
console.log(homePage)
```



```
PS D:\node> node write.js
some content
```

Q36. Write a steps to asynchronously reading and writing file through example.

Import the fs Module:

- First, ensure you have Node.js installed on your system.
- Import the built-in fs (file system) module, which provides file-related functions.

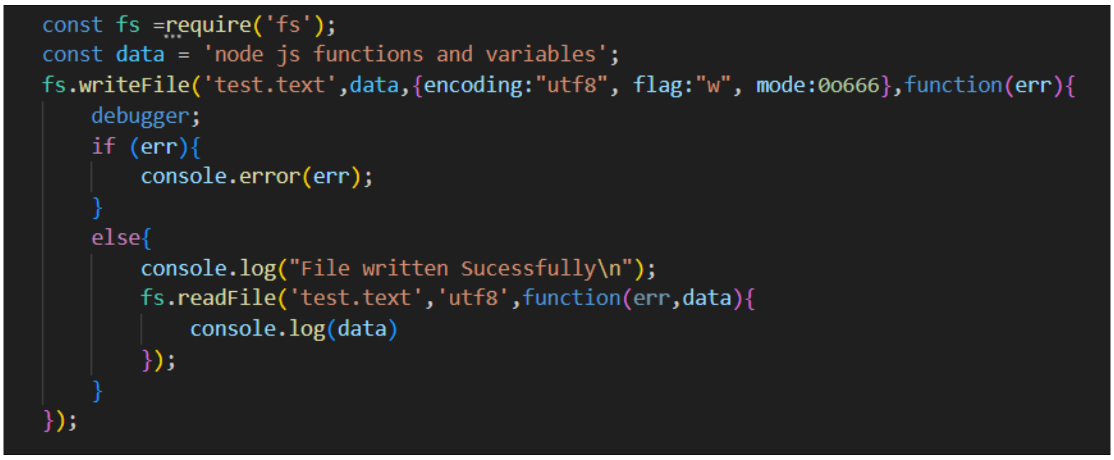
Reading a File Asynchronously:

- Use the fs.readFile() function to read the contents of a file asynchronously.
- Provide the file path and an optional encoding (e.g., 'utf8' for text files).
- Pass a callback function that will be executed when the file read operation completes.

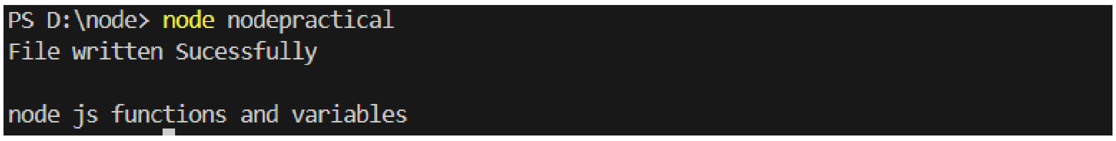
Writing to a File Asynchronously:

- Use the fs.writeFile() function to write data to a file asynchronously.
- Provide the file path, data to be written, and an optional encoding.
- Pass a callback function that will be executed when the file write operation completes.

Example:

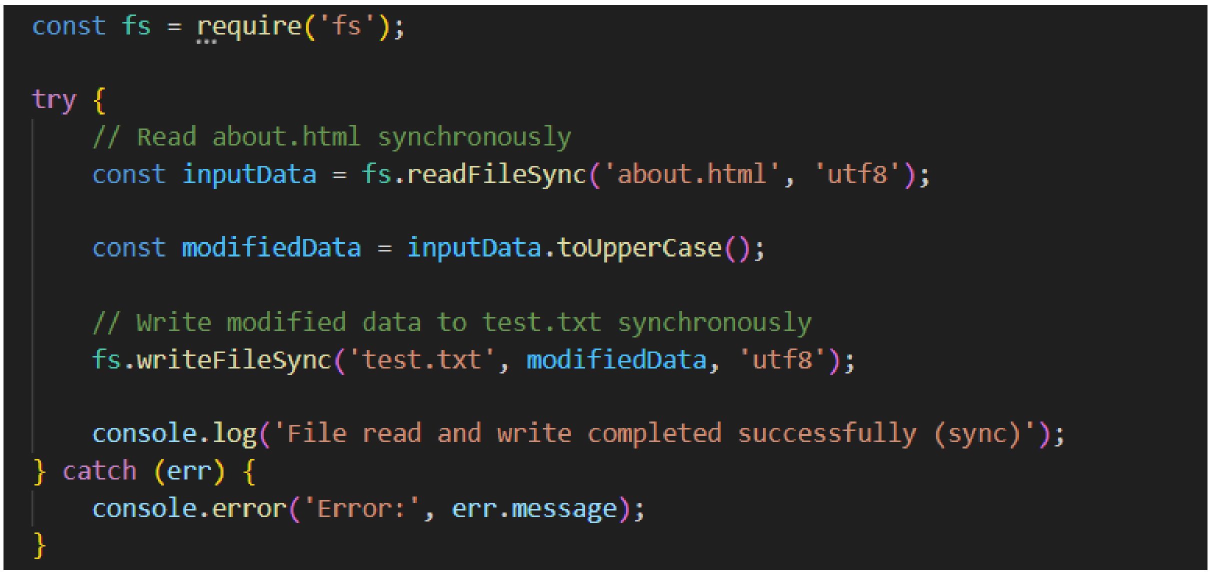


```
const fs =require('fs');
const data = 'node js functions and variables';
fs.writeFile('test.text',data,{encoding:"utf8", flag:"w", mode:0o666},function(err){
    debugger;
    if (err){
        console.error(err);
    }
    else{
        console.log("File written Sucessfully\n");
        fs.readFile('test.text','utf8',function(err,data){
            console.log(data)
        });
    }
});
```

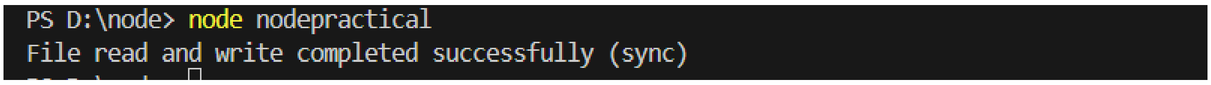


```
PS D:\node> node nodepractical
File written Sucessfully

node js functions and variables
```

Q37. Write a Node JS program to demonstrated reading and writing file through synchronous function.

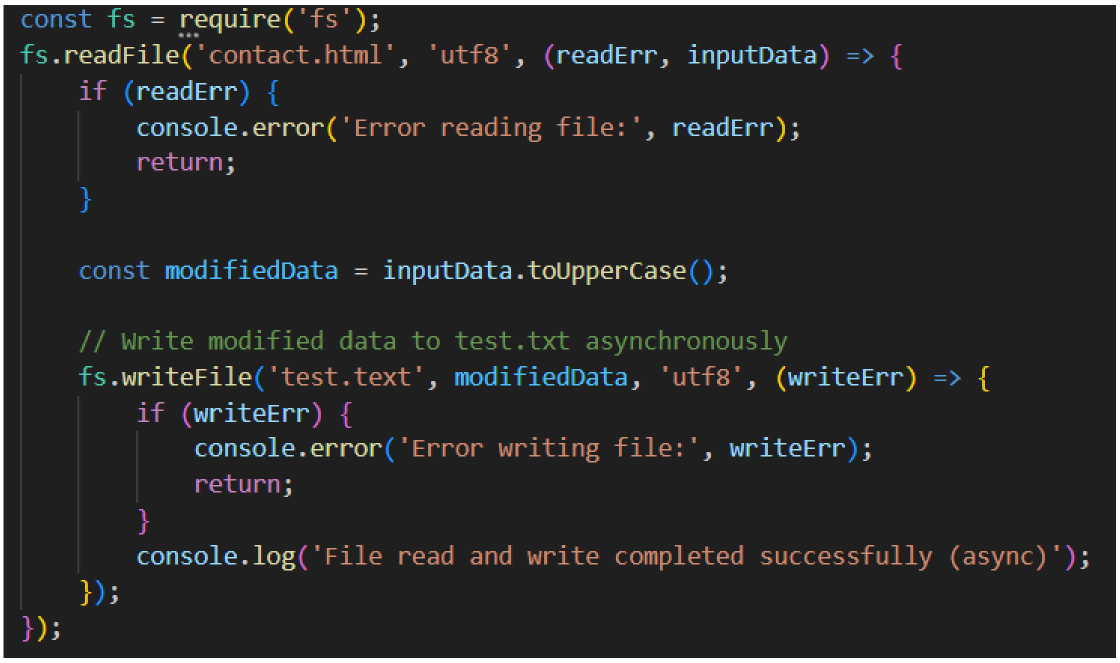


```
const fs = require('fs');

try {
    // Read about.html synchronously
    const inputData = fs.readFileSync('about.html', 'utf8');

    const modifiedData = inputData.toUpperCase();

    // Write modified data to test.txt synchronously
    fs.writeFileSync('test.txt', modifiedData, 'utf8');

    console.log('File read and write completed successfully (sync)');
} catch (err) {
    console.error('Error:', err.message);
}
```

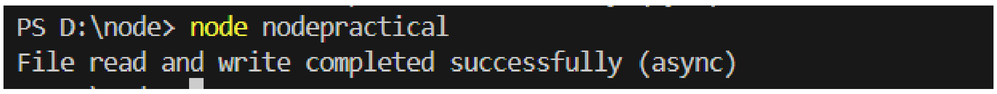


```
PS D:\node> node nodepractical
File read and write completed successfully (sync)
```

Q38. Write a Node JS program to demonstrated reading and writing file asynchronous function.

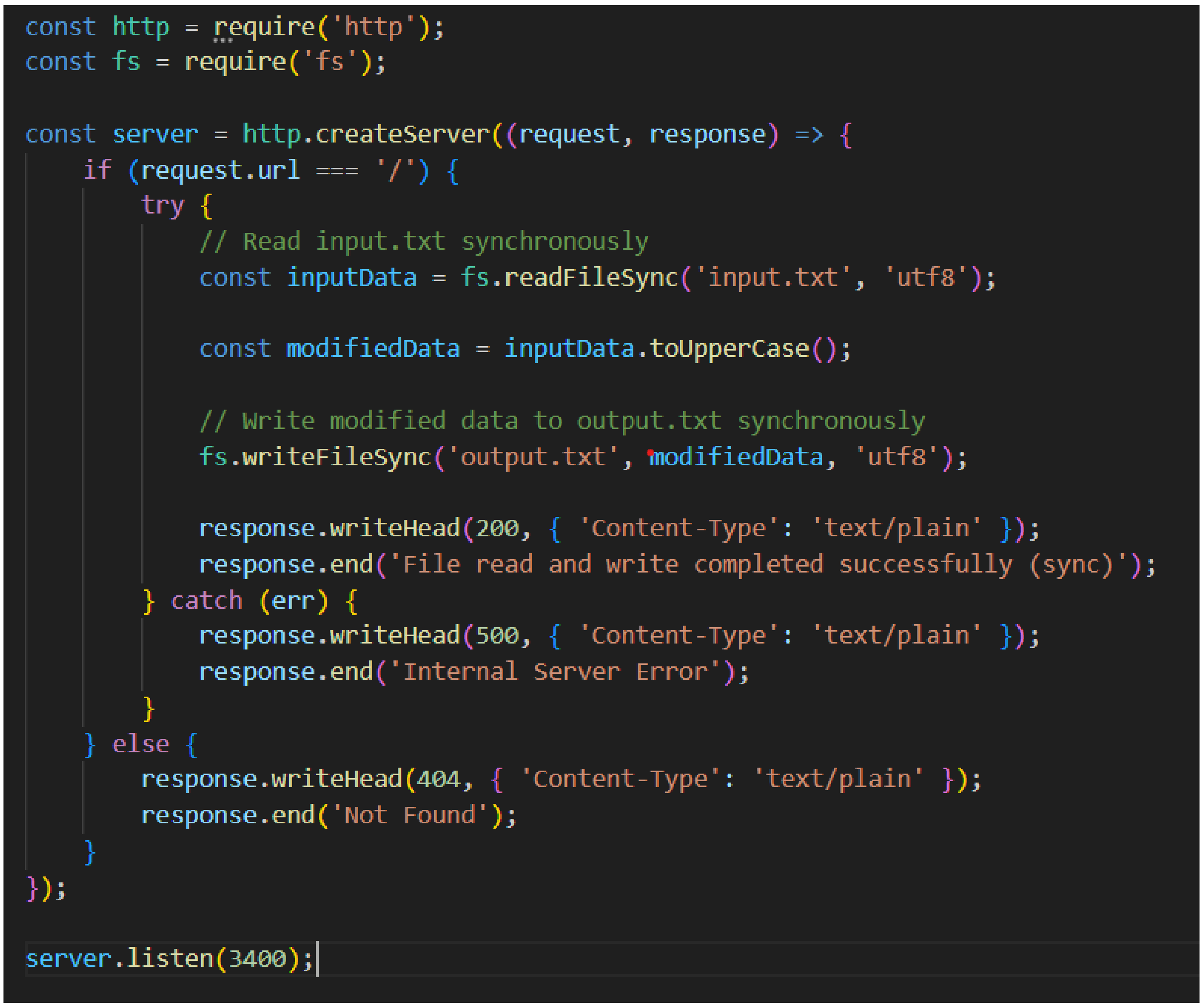


```
const fs = require('fs');
fs.readFile('contact.html', 'utf8', (readErr, inputData) => {
    if (readErr) {
        console.error('Error reading file:', readErr);
        return;
    }

    const modifiedData = inputData.toUpperCase();

    // Write modified data to test.txt asynchronously
    fs.writeFile('test.text', modifiedData, 'utf8', (writeErr) => {
        if (writeErr) {
            console.error('Error writing file:', writeErr);
            return;
        }
        console.log('File read and write completed successfully (async)');
    });
});
```

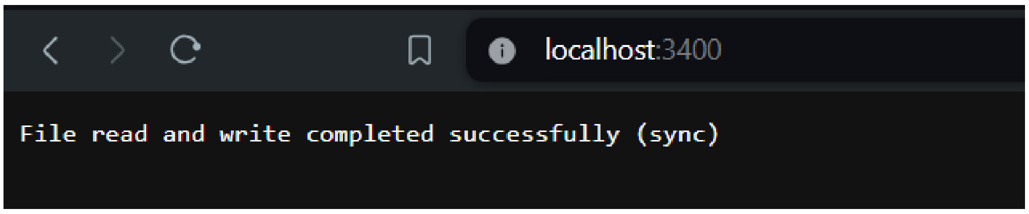


```
PS D:\node> node nodepractical
File read and write completed successfully (async)
```

Q39. Write a Node JS program to demonstrated reading and writing file through synchronous function inside a server.

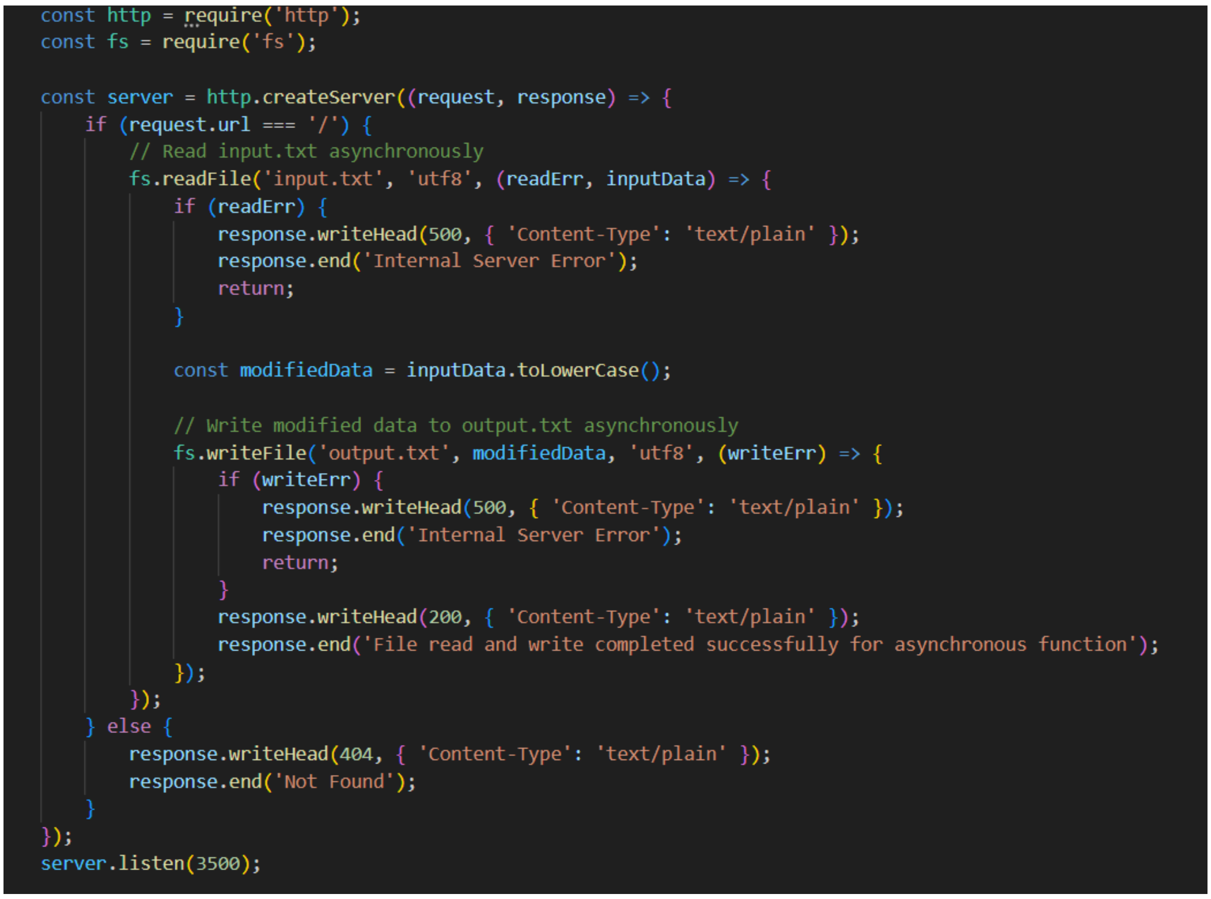


```
const http = require('http');
const fs = require('fs');

const server = http.createServer((request, response) => {
    if (request.url === '/') {
        try {
            // Read input.txt synchronously
            const inputData = fs.readFileSync('input.txt', 'utf8');

            const modifiedData = inputData.toUpperCase();

            // Write modified data to output.txt synchronously
            fs.writeFileSync('output.txt', modifiedData, 'utf8');

            response.writeHead(200, { 'Content-Type': 'text/plain' });
            response.end('File read and write completed successfully (sync)');
        } catch (err) {
            response.writeHead(500, { 'Content-Type': 'text/plain' });
            response.end('Internal Server Error');
        }
    } else {
        response.writeHead(404, { 'Content-Type': 'text/plain' });
        response.end('Not Found');
    }
});

server.listen(3400);
```

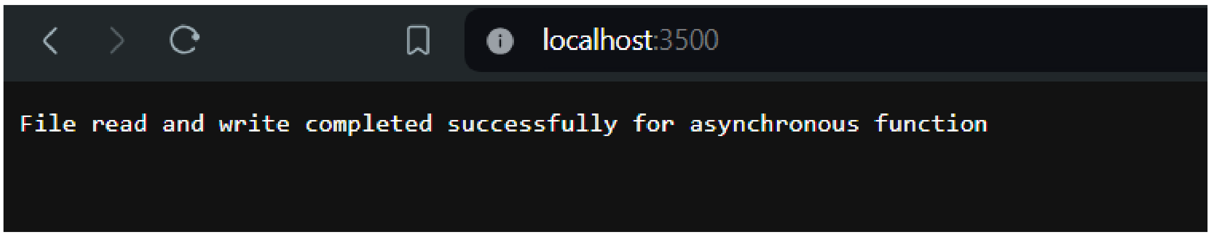


```
File read and write completed successfully (sync)
```

Q40. Write a Node JS program to demonstrated reading and writing file through asynchronous function inside a server.



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

const server = http.createServer((request, response) => {
    if (request.url === '/') {
        // Read input.txt asynchronously
        fs.readFile('input.txt', 'utf8', (readErr, inputData) => {
            if (readErr) {
                response.writeHead(500, { 'Content-Type': 'text/plain' });
                response.end('Internal Server Error');
                return;
            }

            const modifiedData = inputData.toLowerCase();

            // Write modified data to output.txt asynchronously
            fs.writeFile('output.txt', modifiedData, 'utf8', (writeErr) => {
                if (writeErr) {
                    response.writeHead(500, { 'Content-Type': 'text/plain' });
                    response.end('Internal Server Error');
                    return;
                }
                response.writeHead(200, { 'Content-Type': 'text/plain' });
                response.end('File read and write completed successfully for asynchronous function');
            });
        });
    } else {
        response.writeHead(404, { 'Content-Type': 'text/plain' });
        response.end('Not Found');
    }
});
server.listen(3500);
```



```
File read and write completed successfully for asynchronous function
```

In this example:

- When the client requests the root URL (/), the server reads the contents of input.txt asynchronously.
- Converts the data to uppercase (you can perform any other processing).
- Writes the modified data to output.txt.
- The server responds with appropriate messages based on success or failure.

Q41. Explain following function through advantages, syntax and example.

Open(), close(), stat(), openSync(), readfilesync(), fstatSync().

Open( ):

Advantages:

- Asynchronous function that allows non-blocking file opening.
- Useful for handling multiple file operations concurrently.
- The fs.open() function returns a file descriptor (fd), which is an abstract reference to the opened file.
- it provides error handling mrthod through callback function.

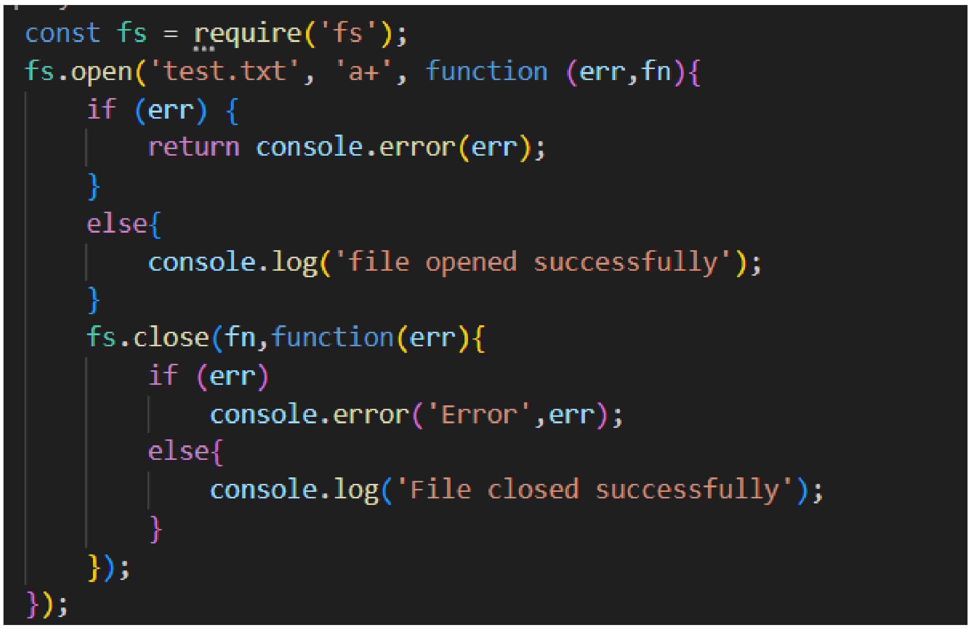
Syntax: **fs.open(path,flags[, mode], callback)**

Close( ):

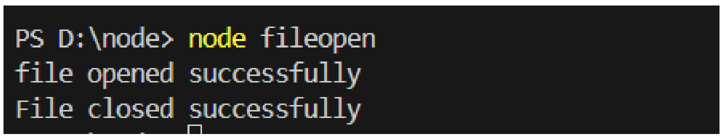
Advantages:

- Closes an opened file, releasing resources.
- Ensure proper cleanup after file operations.
- Prevent resources by leaking

Syntax: **fs.close(fd, callback)**



```
const fs = require('fs');
fs.open('test.txt', 'a+', function (err,fn){
    if (err) {
        return console.error(err);
    }
    else{
        console.log('file opened successfully');
    }
    fs.close(fn,function(err){
        if (err)
            console.error('Error',err);
        else{
            console.log('File closed successfully');
        }
    });
});
```



```
PS D:\node> node fileopen
file opened successfully
File closed successfully
```

Stat( ):

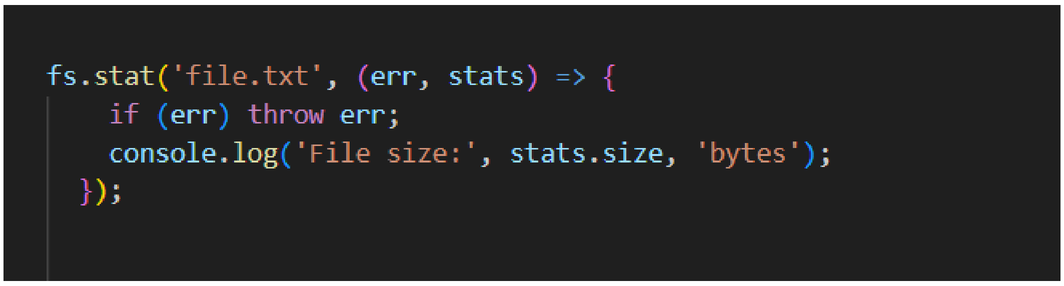
dvantages:

Retrieves information about a file (e.g., size, permissions).

Useful for checking file existence and properties.

Syntax: **fs.stat(path, callback)**

- **path:** Path to the file.
- **callback:** Callback function to handle the result**.**



```
fs.stat('file.txt', (err, stats) => {
    if (err) throw err;
    console.log('File size:', stats.size, 'bytes');
  });
```

openSync():

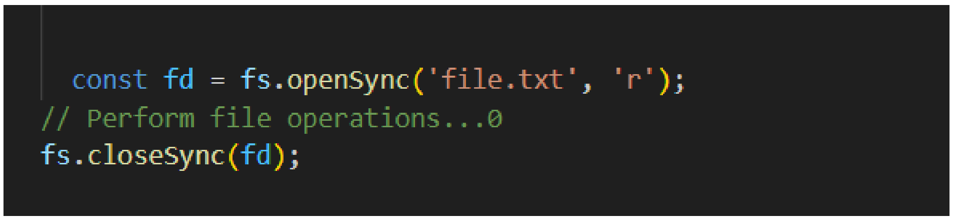
Advantages:

Synchronously opens a file (blocking operation).

Easier to use when sequential execution is acceptable.

Syntax: **const fd = fs.openSync(path, flags[, mode])**

- **path:** Path to the file.
- **flags:** Specifies the file access mode (e.g., 'r' for read, 'w' for write).
- **mode (optional**): Permissions (e.g., 0o666 for read/write permissions).
- **callback:** Callback function to handle the result.



```
  const fd = fs.openSync('file.txt', 'r');
// Perform file operations...0
fs.closeSync(fd);
```

readFileSync():

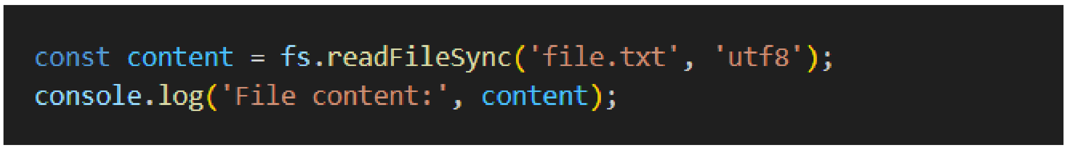
Advantages:

Synchronously reads file content (blocking).

Returns the content as a buffer or string.

Syntax: **const data = fs.readFileSync(path[, options])**

- **path:** Path to the file.
- **options (optional):** Encoding (e.g., 'utf8').



```
const content = fs.readFileSync('file.txt', 'utf8');
console.log('File content:', content);
```

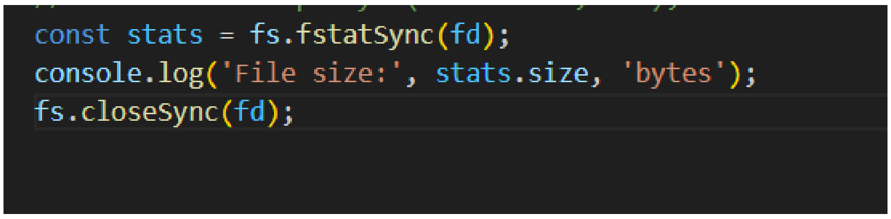
fstatSync():

Advantages:

Synchronously retrieves file information using a file descriptor.

Useful when you already have an open file descriptor.

Syntax: **const stats = fs.fstatSync(fd)**



```
const stats = fs.fstatSync(fd);
console.log('File size:', stats.size, 'bytes');
fs.closeSync(fd);
```

Q42. Write a steps to synchronously file opening and closing through example.

Steps:

1) **Import the fs module:** First, you need to require the fs module in your Node.js script.

   const fs = require('fs');

2) **Open a File Synchronously:** To open a file synchronously, you can use the fs.openSync() method. This method allows you to create, read, or write a file. Here's how you can open a file in read-write mode:

   const filePath = 'input.txt';
   const fileDescriptor = fs.openSync(filePath, 'r+');

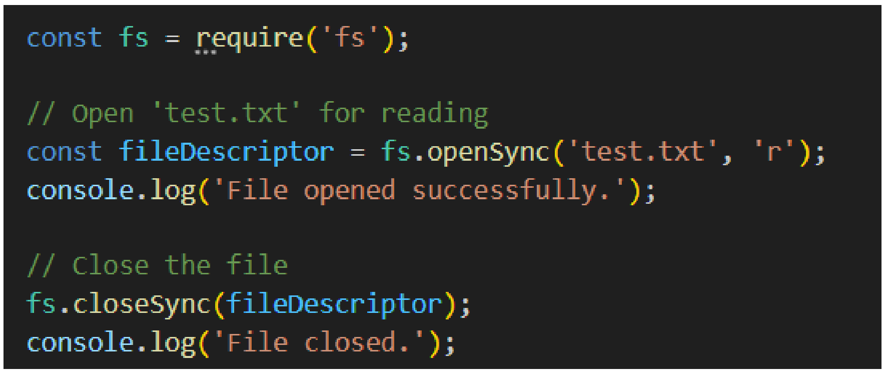
3) **Perform Operations on the File:** Once the file is open, you can perform various operations such as reading, writing, or appending data.

   fs.readFileSync()

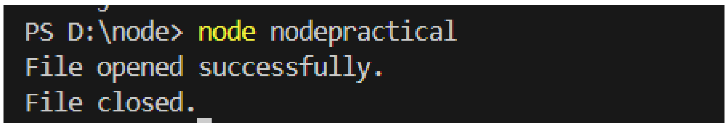
4) **Close the File Synchronously:** After you've finished working with the file, make sure to close it using

   fs.closeSync()

Example:



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

// Open 'test.txt' for reading
const fileDescriptor = fs.openSync('test.txt', 'r');
console.log('File opened successfully.');

// Close the file
fs.closeSync(fileDescriptor);
console.log('File closed.');
```



```
PS D:\node> node nodepractical
File opened successfully.
File closed.
```

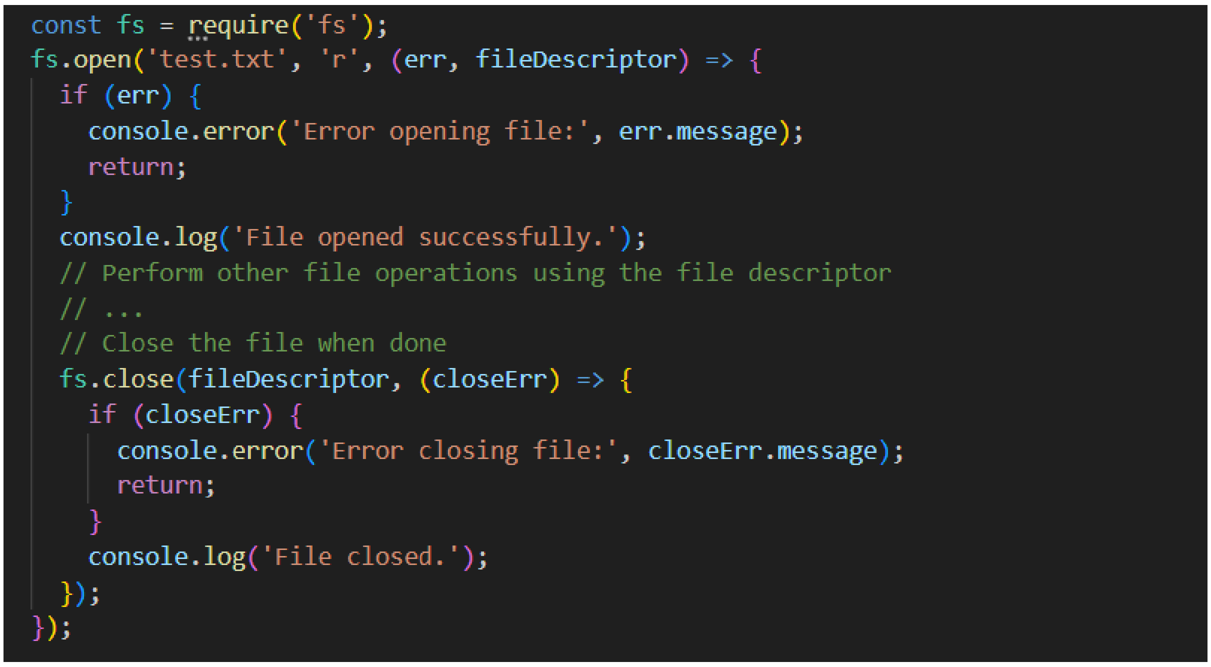
Q43. Write a steps to asynchronously file opening and closing through example.

Steps:

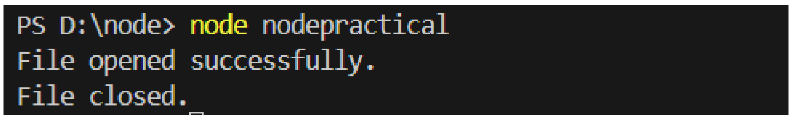
1) **Import the fs module:** First, you need to require the fs module in your Node.js script.
   const fs = require('fs');
2) **Open a file asynchronously:** To open asynchronously, you can use **fs.open()** method.
   The second argument ('r') specifies the file access mode (read-only in this case).
3) **Perform Operations on the File:** Once the file is open, you can perform various operations such as reading, writing, or appending data.
   fs.readFile(filedescripter)

4) **Close the File Synchronously:** After you've finished working with the file, make sure to close it using
   fs.close()

5) **Error Handling:** Always handle errors appropriately. Both the fs.open() and fs.close() functions can encounter errors, so use proper error handling.

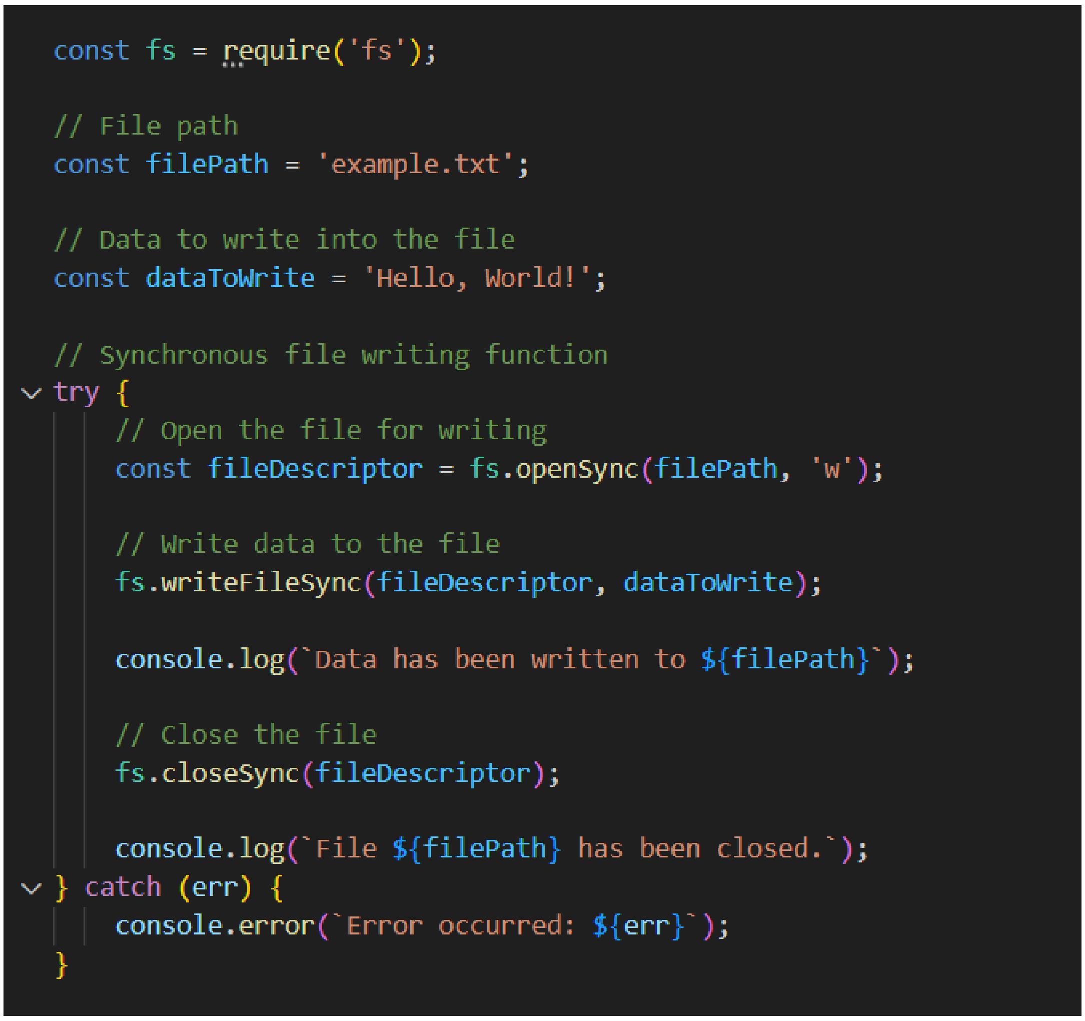


```
const fs = require('fs');
fs.open('test.txt', 'r', (err, fileDescriptor) => {
  if (err) {
    console.error('Error opening file:', err.message);
    return;
  }
  console.log('File opened successfully.');
  // Perform other file operations using the file descriptor
  // ...
  // Close the file when done
  fs.close(fileDescriptor, (closeErr) => {
    if (closeErr) {
      console.error('Error closing file:', closeErr.message);
      return;
    }
    console.log('File closed.');
  });
});
```

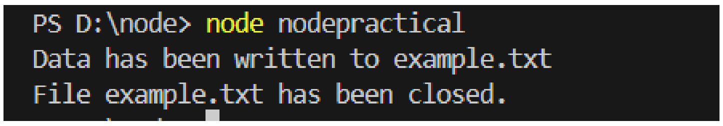


```
PS D:\node> node nodepractical
File opened successfully.
File closed.
```

Q44. Write a Node JS program to demonstrated file opening and closing through synchronous function.

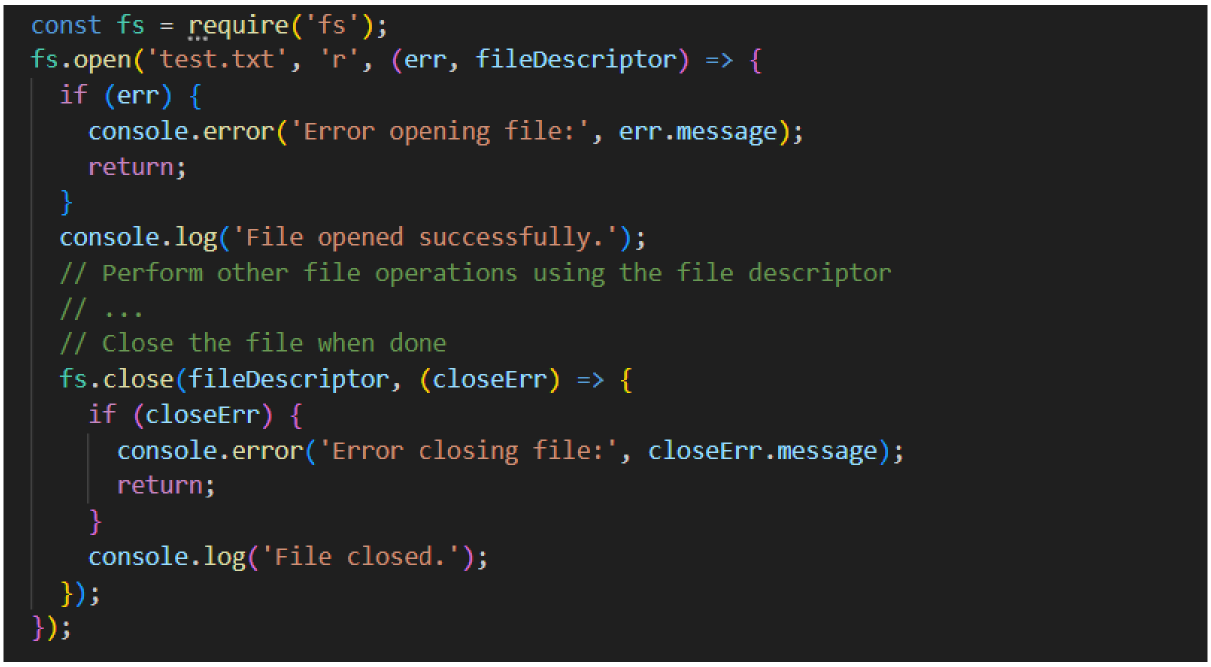


```javascript
const fs = require('fs');

// File path
const filePath = 'example.txt';

// Data to write into the file
const dataToWrite = 'Hello, World!';

// Synchronous file writing function
try {
    // Open the file for writing
    const fileDescriptor = fs.openSync(filePath, 'w');

    // Write data to the file
    fs.writeFileSync(fileDescriptor, dataToWrite);

    console.log(`Data has been written to ${filePath}`);

    // Close the file
    fs.closeSync(fileDescriptor);

    console.log(`File ${filePath} has been closed.`);
} catch (err) {
    console.error(`Error occurred: ${err}`);
}
```

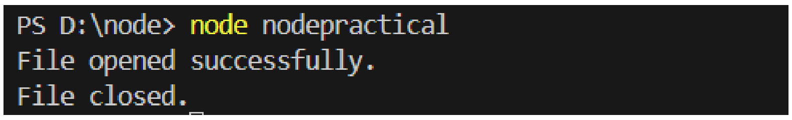


```
PS D:\node> node nodepractical
Data has been written to example.txt
File example.txt has been closed.
```

Q45. Write a Node JS program to demonstrated file opening and closing asynchronous function.

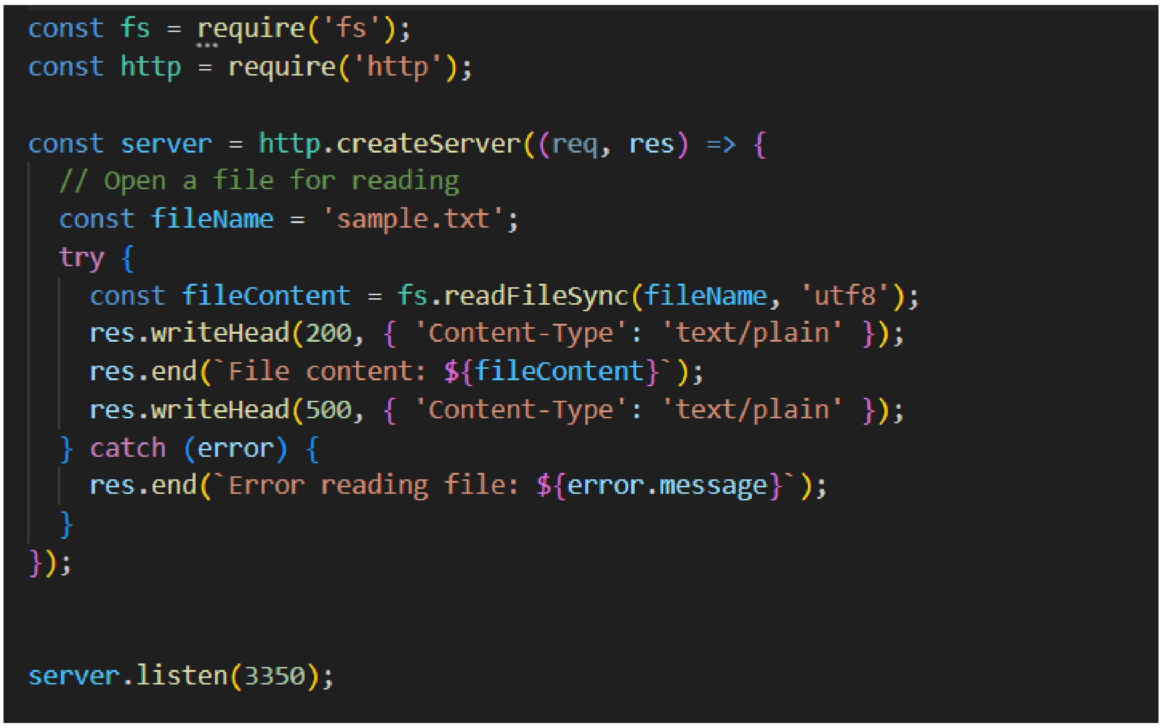


```
const fs = require('fs');
fs.open('test.txt', 'r', (err, fileDescriptor) => {
  if (err) {
    console.error('Error opening file:', err.message);
    return;
  }
  console.log('File opened successfully.');
  // Perform other file operations using the file descriptor
  // ...
  // Close the file when done
  fs.close(fileDescriptor, (closeErr) => {
    if (closeErr) {
      console.error('Error closing file:', closeErr.message);
      return;
    }
    console.log('File closed.');
  });
});
```

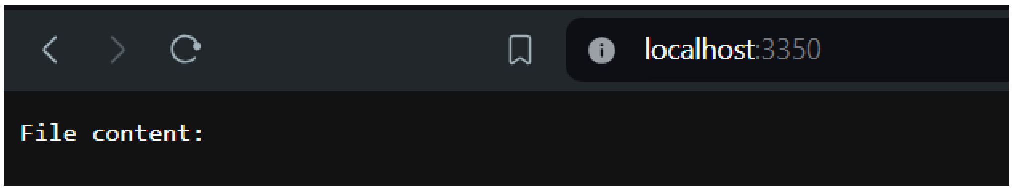


```
PS D:\node> node nodepractical
File opened successfully.
File closed.
```

Q46. Write a Node JS program to demonstrated file opening and closing through synchronous function inside a server.



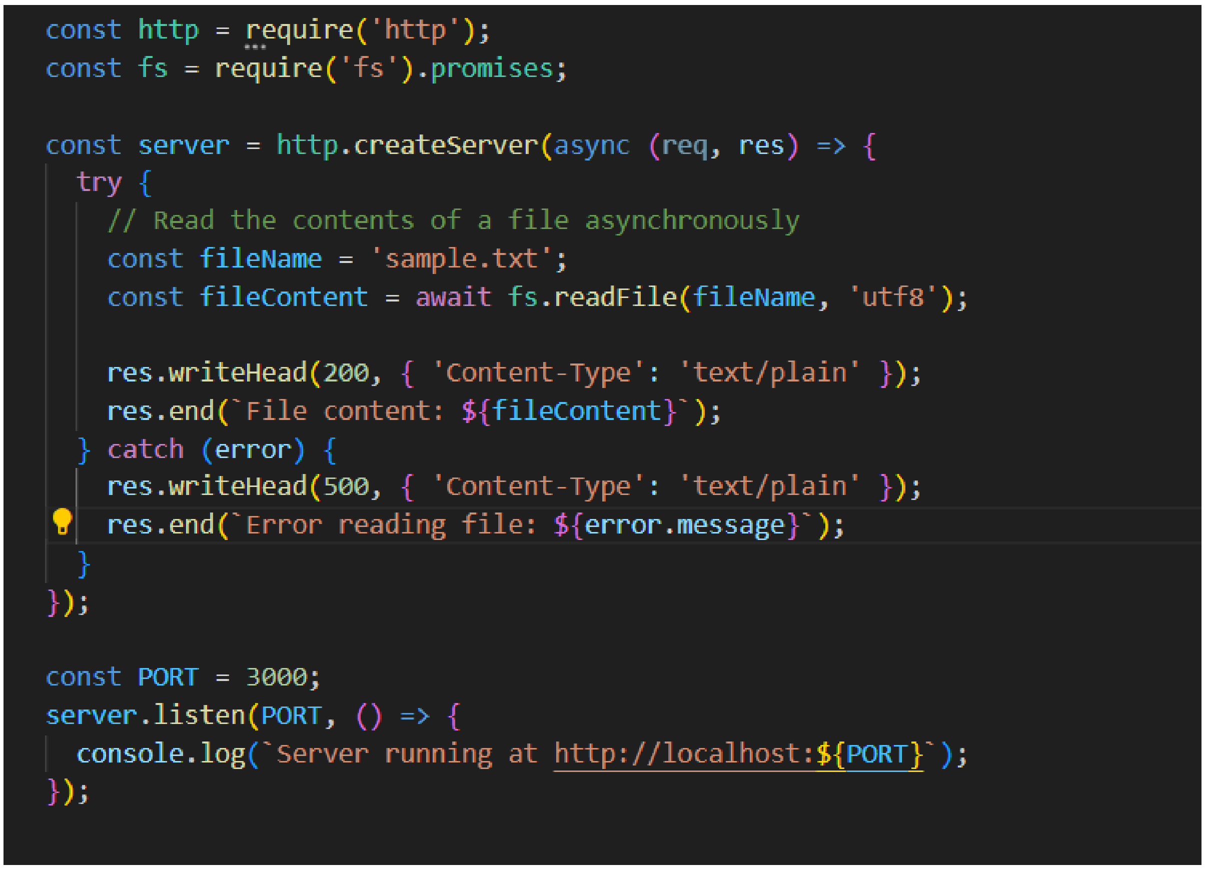
```
const fs = require('fs');
const http = require('http');

const server = http.createServer((req, res) => {
  // Open a file for reading
  const fileName = 'sample.txt';
  try {
    const fileContent = fs.readFileSync(fileName, 'utf8');
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end(`File content: ${fileContent}`);
    res.writeHead(500, { 'Content-Type': 'text/plain' });
  } catch (error) {
    res.end(`Error reading file: ${error.message}`);
  }
});


server.listen(3350);
```



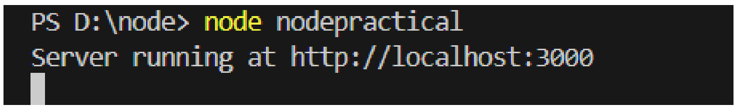
```
File content:
```

- We create an HTTP server using the http module.
- When a request is made to the server, it reads the content from the sample.txt file synchronously.
- If successful, it responds with the file content; otherwise, it sends an error message.

Q47. Write a Node JS program to demonstrated file opening and closing through asynchronous function inside a server.



```
const http = require('http');
const fs = require('fs').promises;

const server = http.createServer(async (req, res) => {
  try {
    // Read the contents of a file asynchronously
    const fileName = 'sample.txt';
    const fileContent = await fs.readFile(fileName, 'utf8');

    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end(`File content: ${fileContent}`);
  } catch (error) {
    res.writeHead(500, { 'Content-Type': 'text/plain' });
    res.end(`Error reading file: ${error.message}`);
  }
});

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

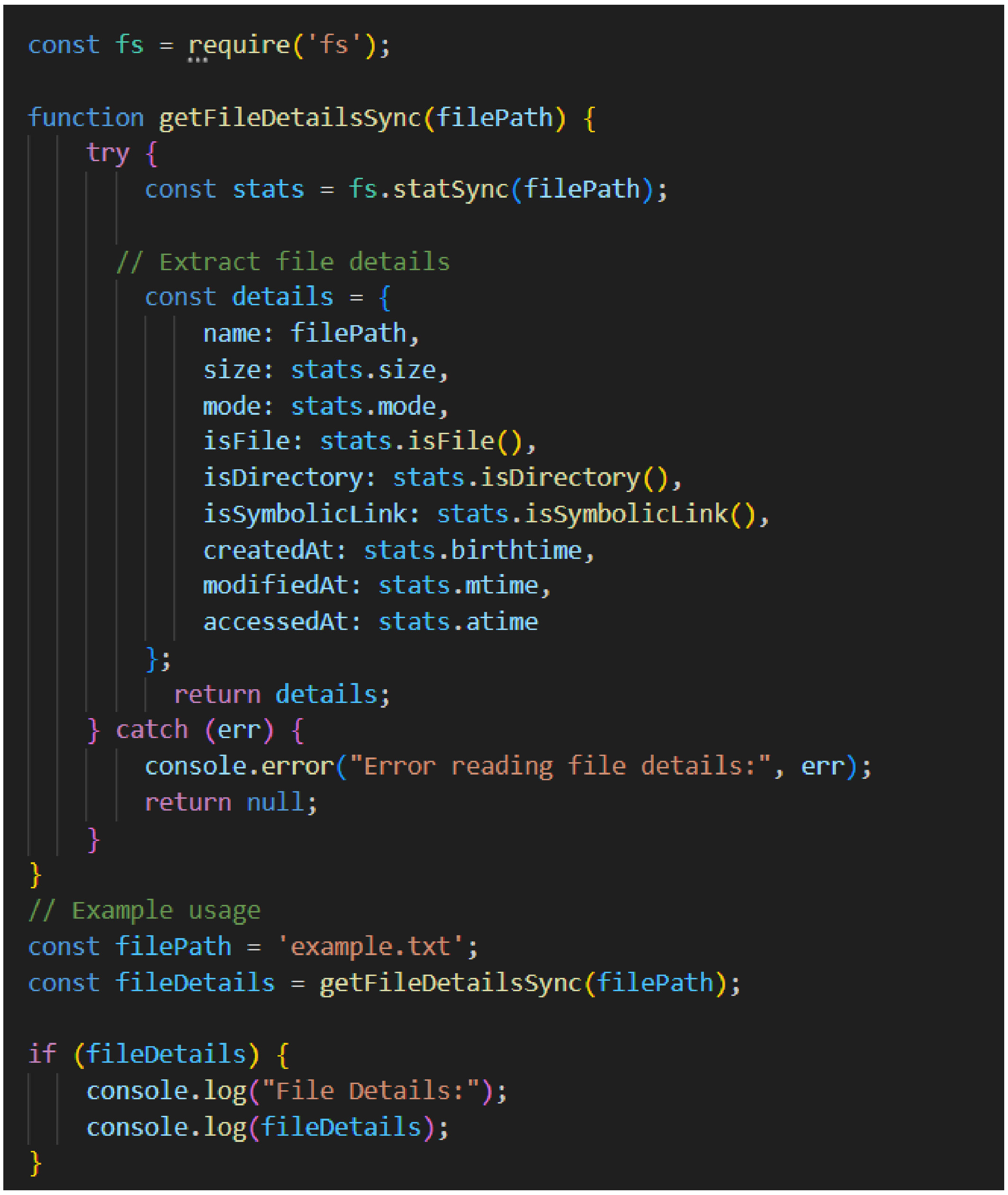


```
PS D:\node> node nodepractical
Server running at http://localhost:3000
```

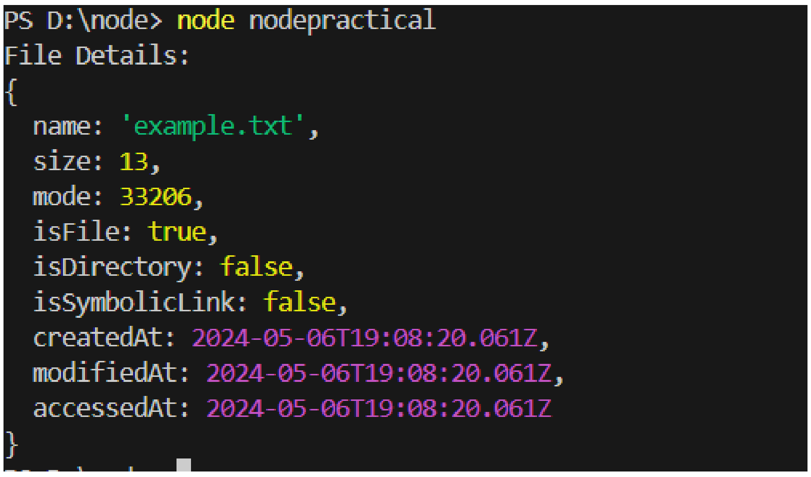
In this program:

- We create an HTTP server using the http module.
- When a request is made to the server, it reads the content from the sample.txt file asynchronously.
- If successful, it responds with the file content; otherwise, it sends an error message.

Q48. Write a Node JS program to demonstrated status/details of file through synchronous function.

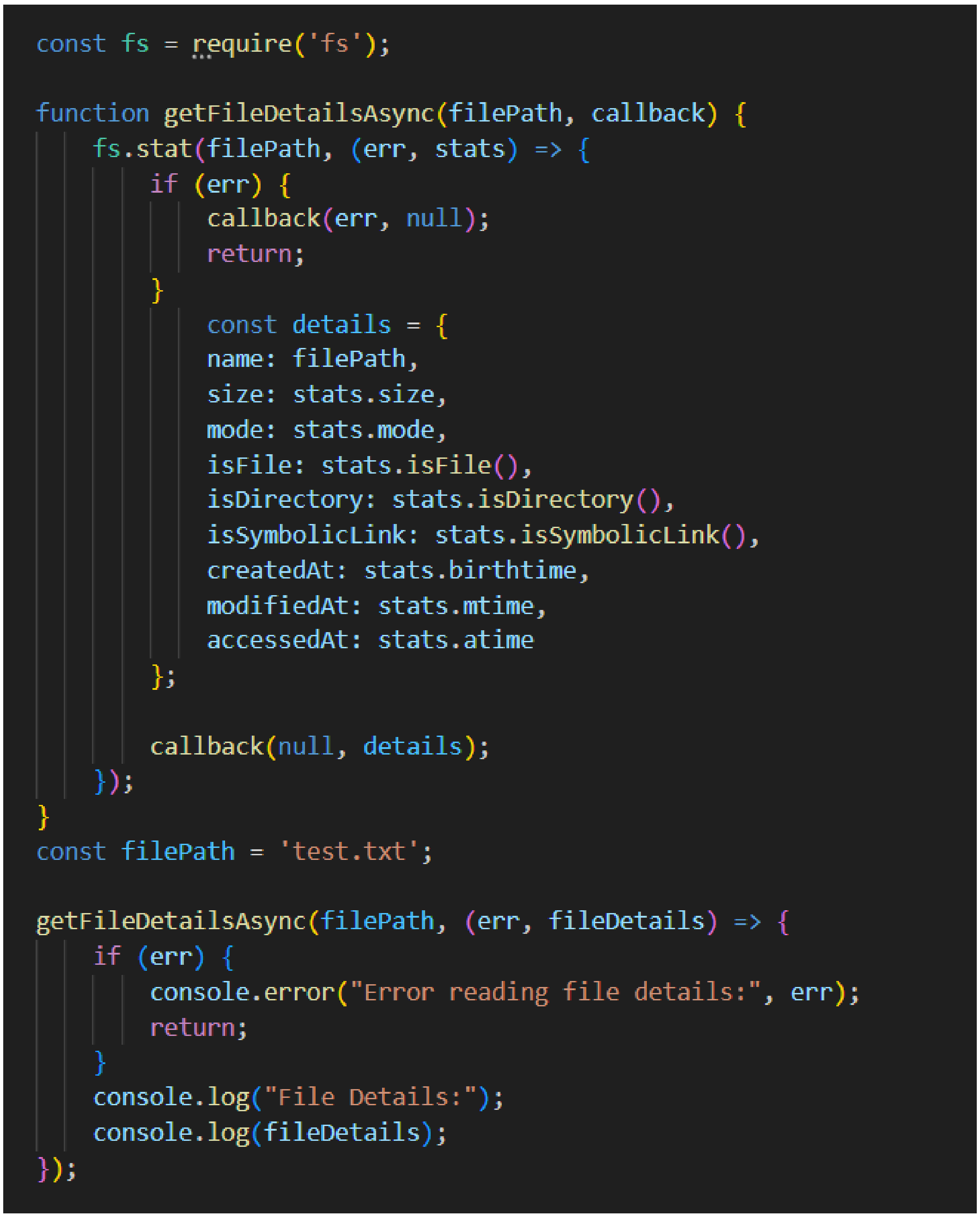


```
const fs = require('fs');

function getFileDetailsSync(filePath) {
    try {
        const stats = fs.statSync(filePath);

       // Extract file details
        const details = {
            name: filePath,
            size: stats.size,
            mode: stats.mode,
            isFile: stats.isFile(),
            isDirectory: stats.isDirectory(),
            isSymbolicLink: stats.isSymbolicLink(),
            createdAt: stats.birthtime,
            modifiedAt: stats.mtime,
            accessedAt: stats.atime
        };
          return details;
    } catch (err) {
        console.error("Error reading file details:", err);
        return null;
    }
}
// Example usage
const filePath = 'example.txt';
const fileDetails = getFileDetailsSync(filePath);

if (fileDetails) {
    console.log("File Details:");
    console.log(fileDetails);
}
```

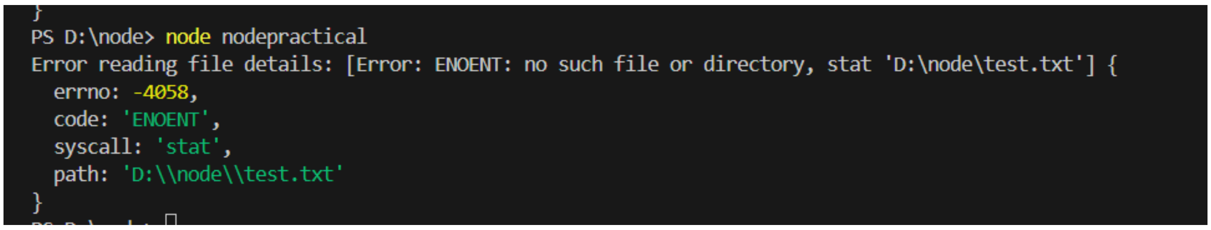


```
PS D:\node> node nodepractical
File Details:
{
  name: 'example.txt',
  size: 13,
  mode: 33206,
  isFile: true,
  isDirectory: false,
  isSymbolicLink: false,
  createdAt: 2024-05-06T19:08:20.061Z,
  modifiedAt: 2024-05-06T19:08:20.061Z,
  accessedAt: 2024-05-06T19:08:20.061Z
}
```

Q49. Write a Node JS program to demonstrated status/details of file through asynchronous function.

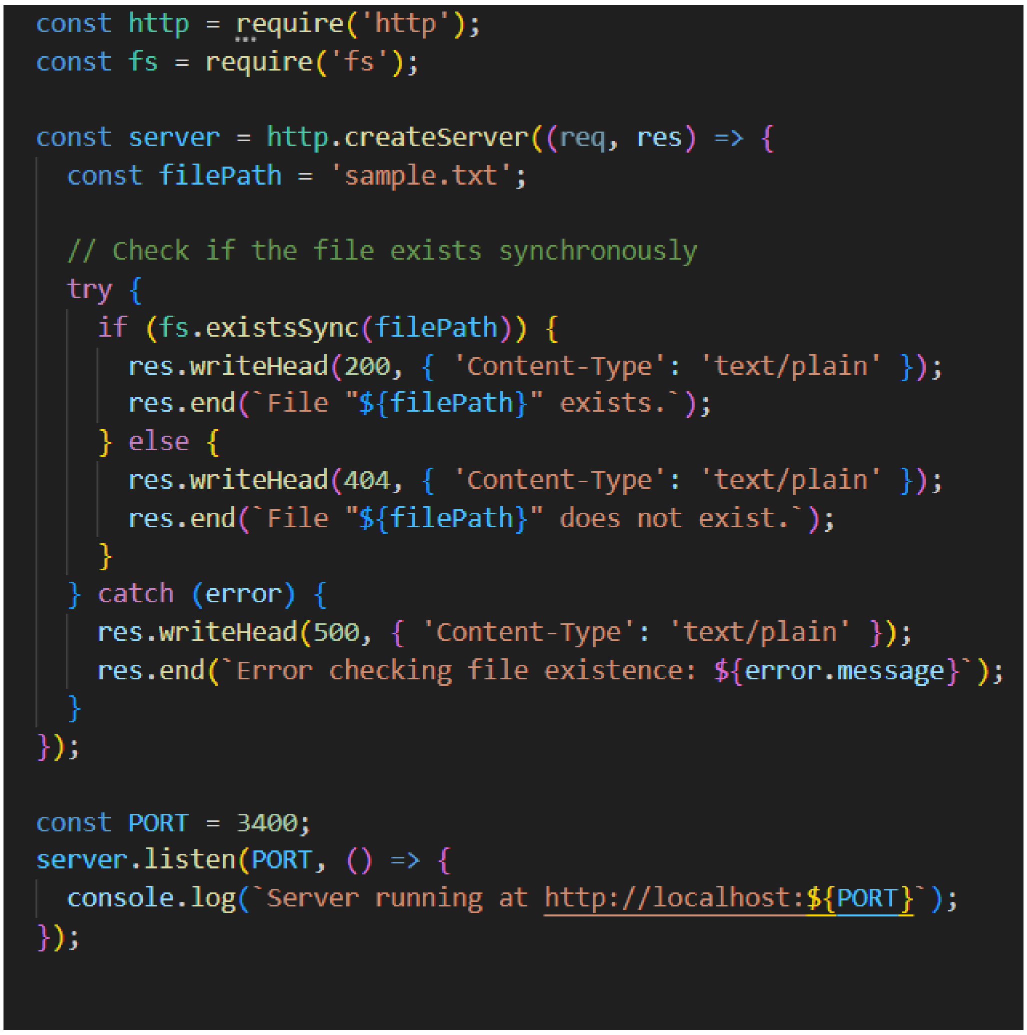


```javascript
const fs = require('fs');

function getFileDetailsAsync(filePath, callback) {
    fs.stat(filePath, (err, stats) => {
        if (err) {
            callback(err, null);
            return;
        }
            const details = {
            name: filePath,
            size: stats.size,
            mode: stats.mode,
            isFile: stats.isFile(),
            isDirectory: stats.isDirectory(),
            isSymbolicLink: stats.isSymbolicLink(),
            createdAt: stats.birthtime,
            modifiedAt: stats.mtime,
            accessedAt: stats.atime
        };

        callback(null, details);
    });
}
const filePath = 'test.txt';

getFileDetailsAsync(filePath, (err, fileDetails) => {
    if (err) {
        console.error("Error reading file details:", err);
        return;
    }
    console.log("File Details:");
    console.log(fileDetails);
});
```

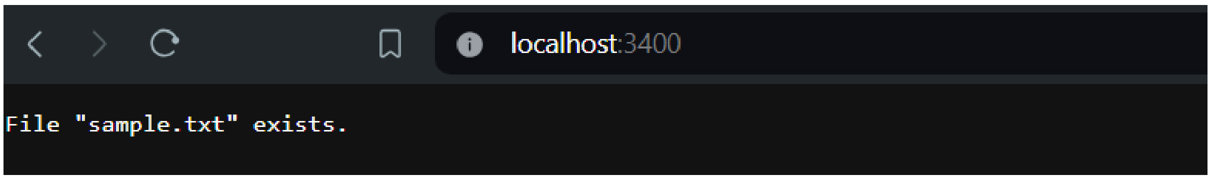


```
}
PS D:\node> node nodepractical
Error reading file details: [Error: ENOENT: no such file or directory, stat 'D:\node\test.txt'] {
  errno: -4058,
  code: 'ENOENT',
  syscall: 'stat',
  path: 'D:\\node\\test.txt'
}
```

Q50. Write a Node JS program to demonstrated status/details of file through synchronous function inside a server.

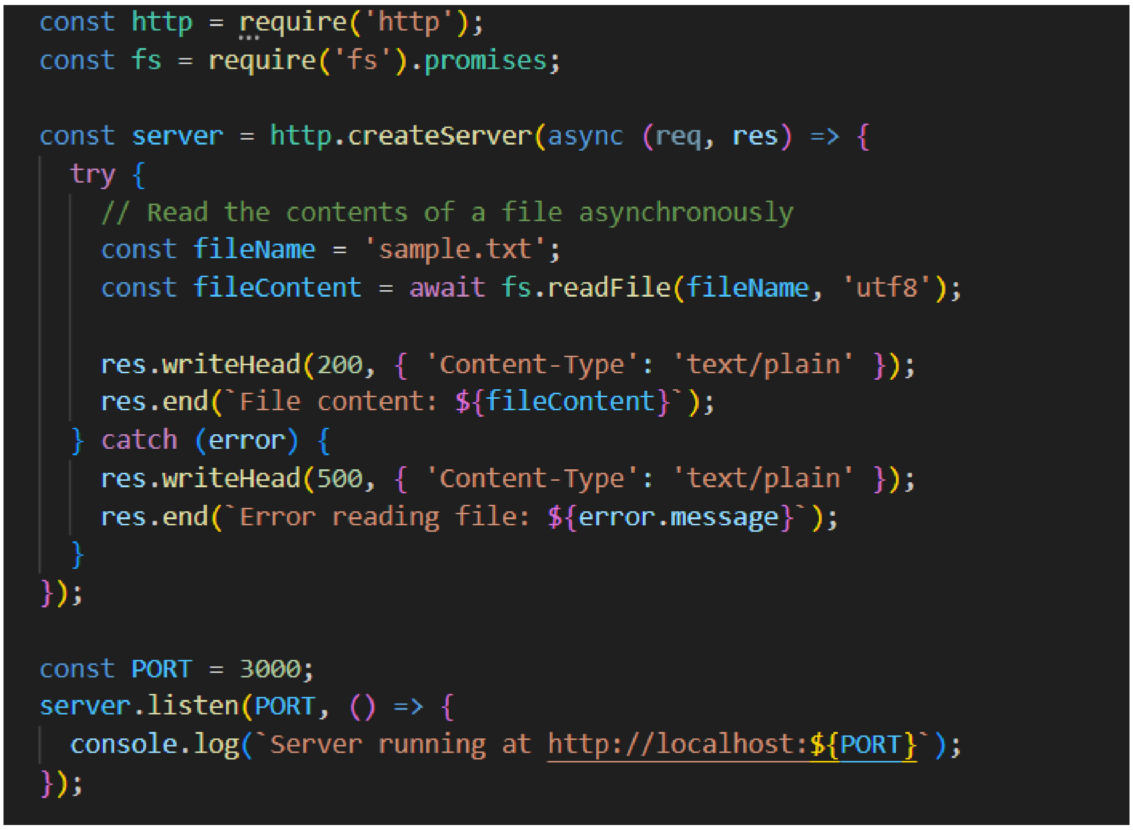


```
const http = require('http');
const fs = require('fs');

const server = http.createServer((req, res) => {
  const filePath = 'sample.txt';

  // Check if the file exists synchronously
  try {
    if (fs.existsSync(filePath)) {
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end(`File "${filePath}" exists.`);
    } else {
      res.writeHead(404, { 'Content-Type': 'text/plain' });
      res.end(`File "${filePath}" does not exist.`);
    }
  } catch (error) {
    res.writeHead(500, { 'Content-Type': 'text/plain' });
    res.end(`Error checking file existence: ${error.message}`);
  }
});

const PORT = 3400;
server.listen(PORT, () => {
  console.log(`Server running at http://localhost:${PORT}`);
});
```

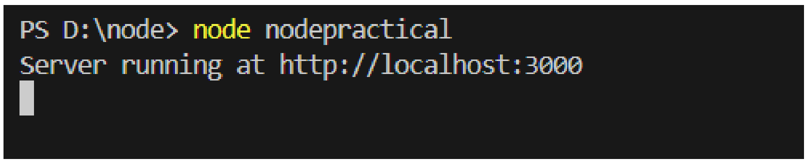


```
File "sample.txt" exists.
```

Q51. Write a Node JS program to demonstrated status/details of file through asynchronous function inside a server.



```
const http = require('http');
const fs = require('fs').promises;

const server = http.createServer(async (req, res) => {
  try {
    // Read the contents of a file asynchronously
    const fileName = 'sample.txt';
    const fileContent = await fs.readFile(fileName, 'utf8');

    res.writeHead(200, { 'Content-Type': 'text/plain' });
    res.end(`File content: ${fileContent}`);
  } catch (error) {
    res.writeHead(500, { 'Content-Type': 'text/plain' });
    res.end(`Error reading file: ${error.message}`);
  }
});

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



```
PS D:\node> node nodepractical
Server running at http://localhost:3000
```

In this program:

- We create an HTTP server using the http module.
- When a request is made to the server, it checks if the file named sample.txt exists asynchronously.
- If the file exists, it responds with the file content; otherwise, it sends an error message.

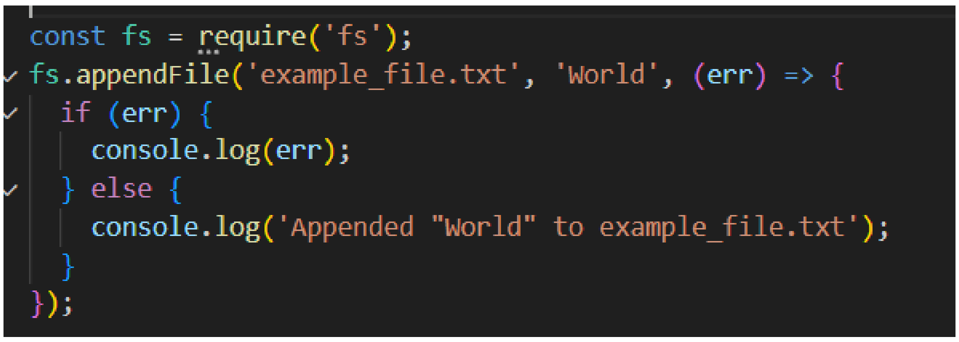
Q52. Explain appendFile, unlink and unlinkSync function through advantages, syntax and example.

Fs.appendFile():

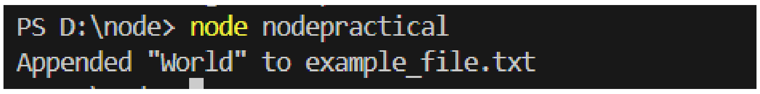
Advantages:

- Asynchronous does not block the event loop, allowing other tasks to continue.
- Appends data to an existing file without overwriting its content.
- Creates a new file if it doesn't exist.

Syntax: **fs.appendFile(path, data[, options], callback)**



```
const fs = require('fs');
fs.appendFile('example_file.txt', 'World', (err) => {
  if (err) {
    console.log(err);
  } else {
    console.log('Appended "World" to example_file.txt');
  }
});
```



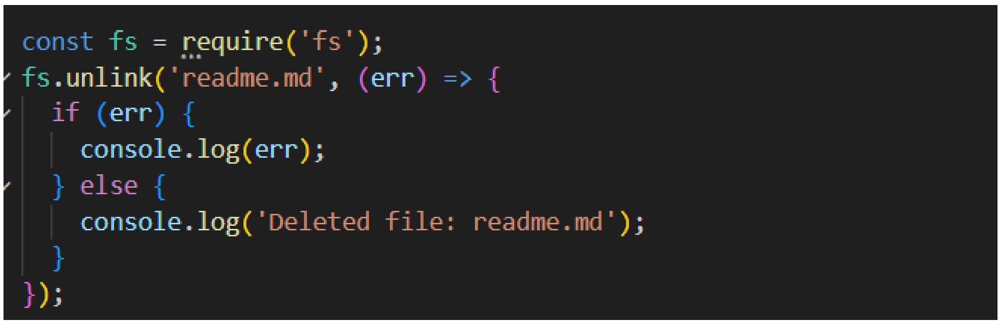
```
PS D:\node> node nodepractical
Appended "World" to example_file.txt
```

fs.unlink():

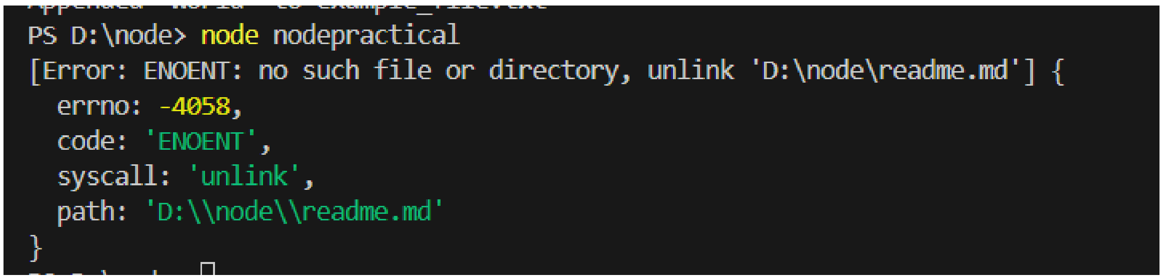
Advantages:

- Asynchronous non-blocking operation.
- Removes a file or symbolic link from the filesystem.

Syntax: **fs.unlink(path, callback)**



```
const fs = require('fs');
fs.unlink('readme.md', (err) => {
  if (err) {
    console.log(err);
  } else {
    console.log('Deleted file: readme.md');
  }
});
```



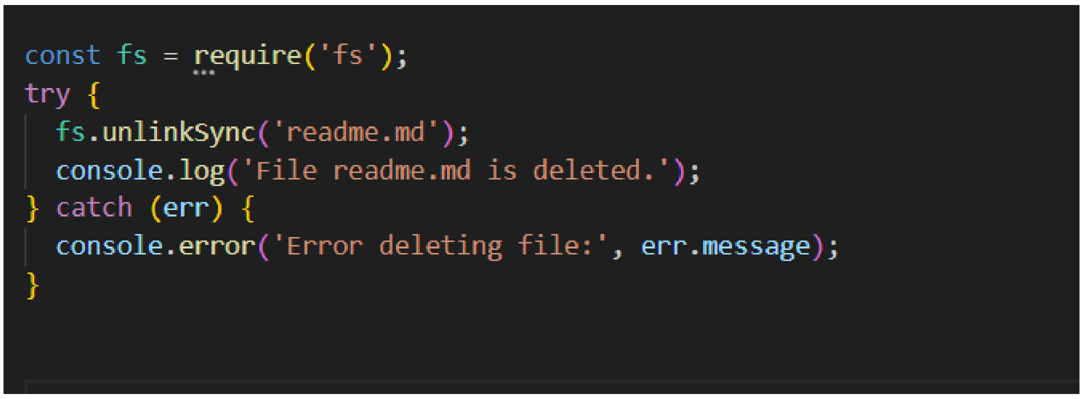
```
PS D:\node> node nodepractical
[Error: ENOENT: no such file or directory, unlink 'D:\node\readme.md'] {
  errno: -4058,
  code: 'ENOENT',
  syscall: 'unlink',
  path: 'D:\\node\\readme.md'
}
```

fs.unlinkSync():

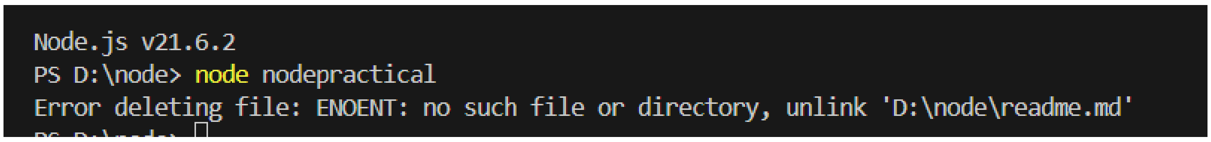
Advantages:

- Synchronous: Blocks execution until the file is removed.
- Removes a file or symbolic link.

Syntax: **fs.unlinkSync(path)**



```javascript
const fs = require('fs');
try {
  fs.unlinkSync('readme.md');
  console.log('File readme.md is deleted.');
} catch (err) {
  console.error('Error deleting file:', err.message);
}
```



```
Node.js v21.6.2
PS D:\node> node nodepractical
Error deleting file: ENOENT: no such file or directory, unlink 'D:\node\readme.md'
```

Q53. Write a steps to synchronously appending a content in file through example.

Synchronously Appending Content to a File

Import the fs Module:

- First, ensure you have Node.js installed on your system.
- Create a new Node.js project or use an existing one.
- Import the fs module, which provides file system-related functions.

Choose a File to Append Data:

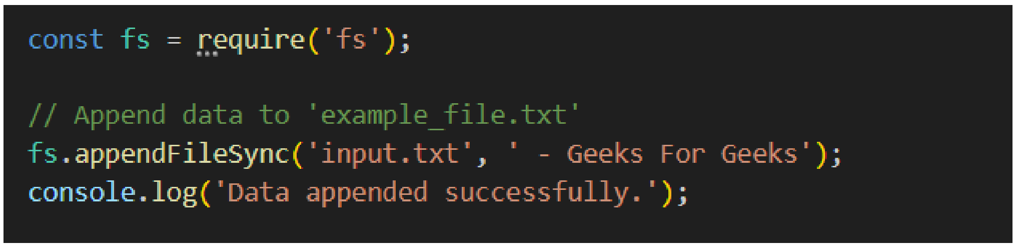
- Decide which file you want to append data to. For this example, let's assume you have a file named example_file.txt.

Append Data Synchronously:

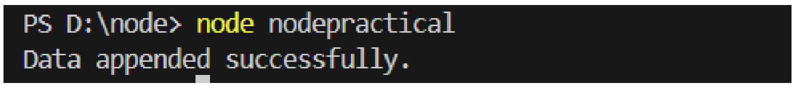
- Use the fs.appendFileSync() function to append data to the file in synchronous mode.

Syntax: **const fs = require('fs');**

fs.appendFileSync('example_file.txt', ' – hey there!');



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

// Append data to 'example_file.txt'
fs.appendFileSync('input.txt', ' - Geeks For Geeks');
console.log('Data appended successfully.');
```



```
PS D:\node> node nodepractical
Data appended successfully.
```

Q54. Write a steps to asynchronously appending a content in file through example.

Asynchronously Appending Content to a File

Import the fs Module:

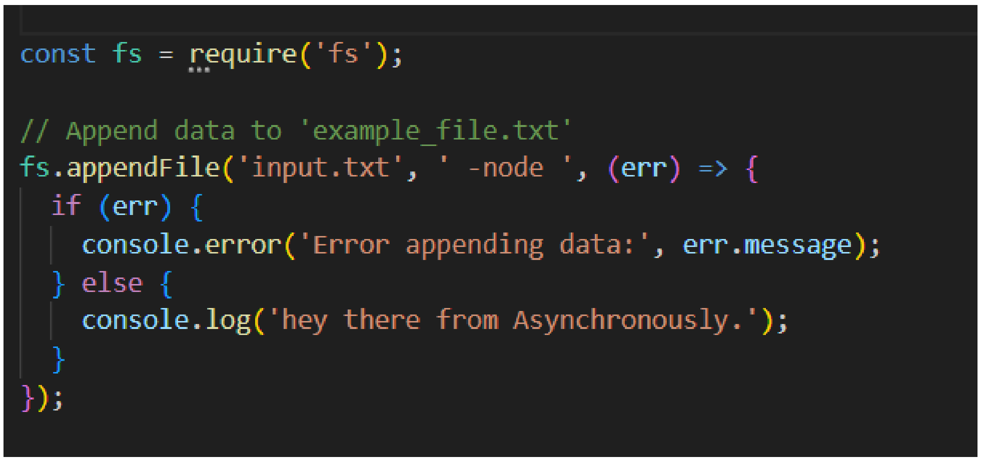
- First, ensure you have Node.js installed on your system.
- Create a new Node.js project or use an existing one.
- Import the fs module, which provides file system-related functions.

Choose a File to Append Data:

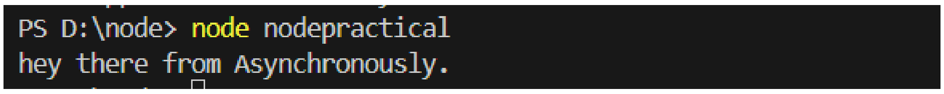
- Decide which file you want to append data to. For this example, let's assume you have a file named example_file.txt.

Append Data Asynchronously:

- Use the fs.appendFile() function to append data to the file in asynchronous mode.

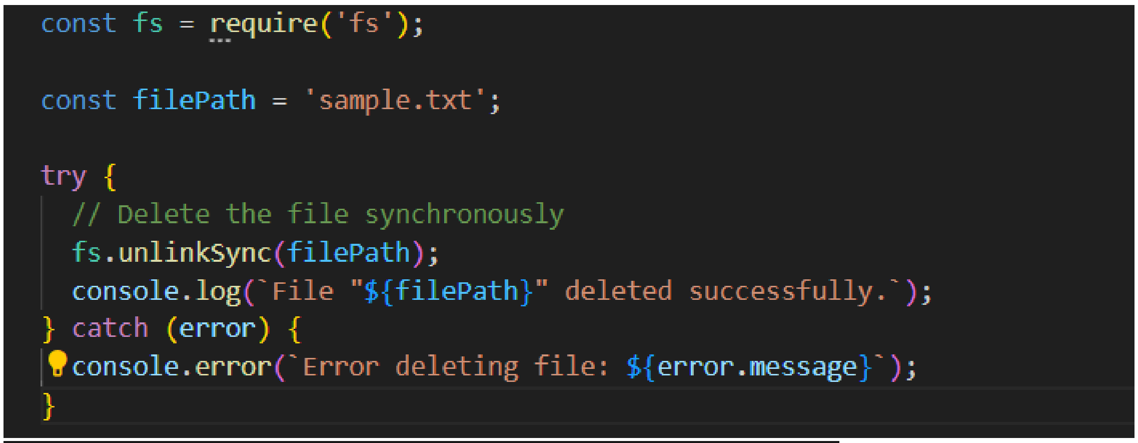


```
const fs = require('fs');

// Append data to 'example_file.txt'
fs.appendFile('input.txt', ' -node ', (err) => {
  if (err) {
    console.error('Error appending data:', err.message);
  } else {
    console.log('hey there from Asynchronously.');
  }
});
```

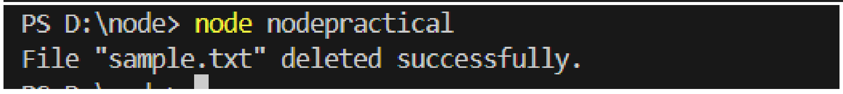


```
PS D:\node> node nodepractical
hey there from Asynchronously.
```

Q55. Write a Node JS program to demonstrated deleting a file through synchronous function.



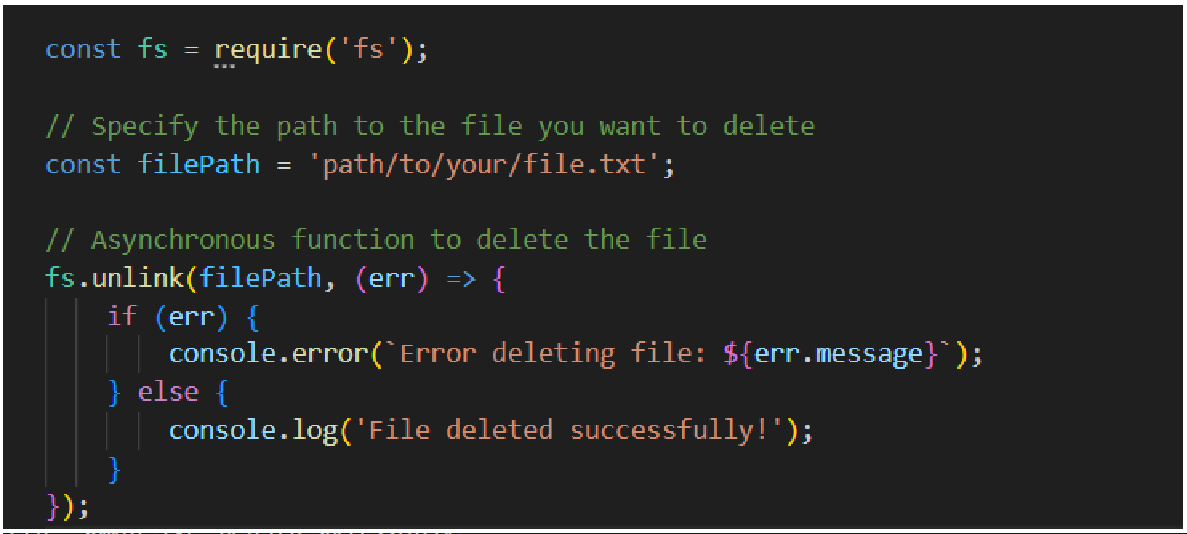
```
const fs = require('fs');

const filePath = 'sample.txt';

try {
  // Delete the file synchronously
  fs.unlinkSync(filePath);
  console.log(`File "${filePath}" deleted successfully.`);
} catch (error) {
  console.error(`Error deleting file: ${error.message}`);
}
```



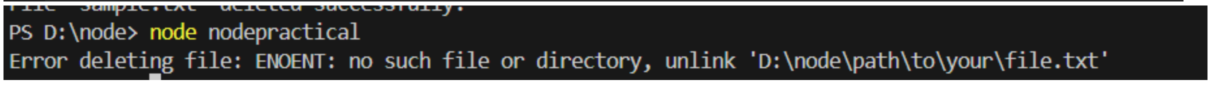
```
PS D:\node> node nodepractical
File "sample.txt" deleted successfully.
```

- We specify the file path (e.g., sample.txt).
- We use fs.unlinkSync(filePath) to delete the file synchronously.
- If the file exists, it will be removed; otherwise, an error will be thrown.

Q56. Write a Node JS program to demonstrated deleting a file through asynchronous function.



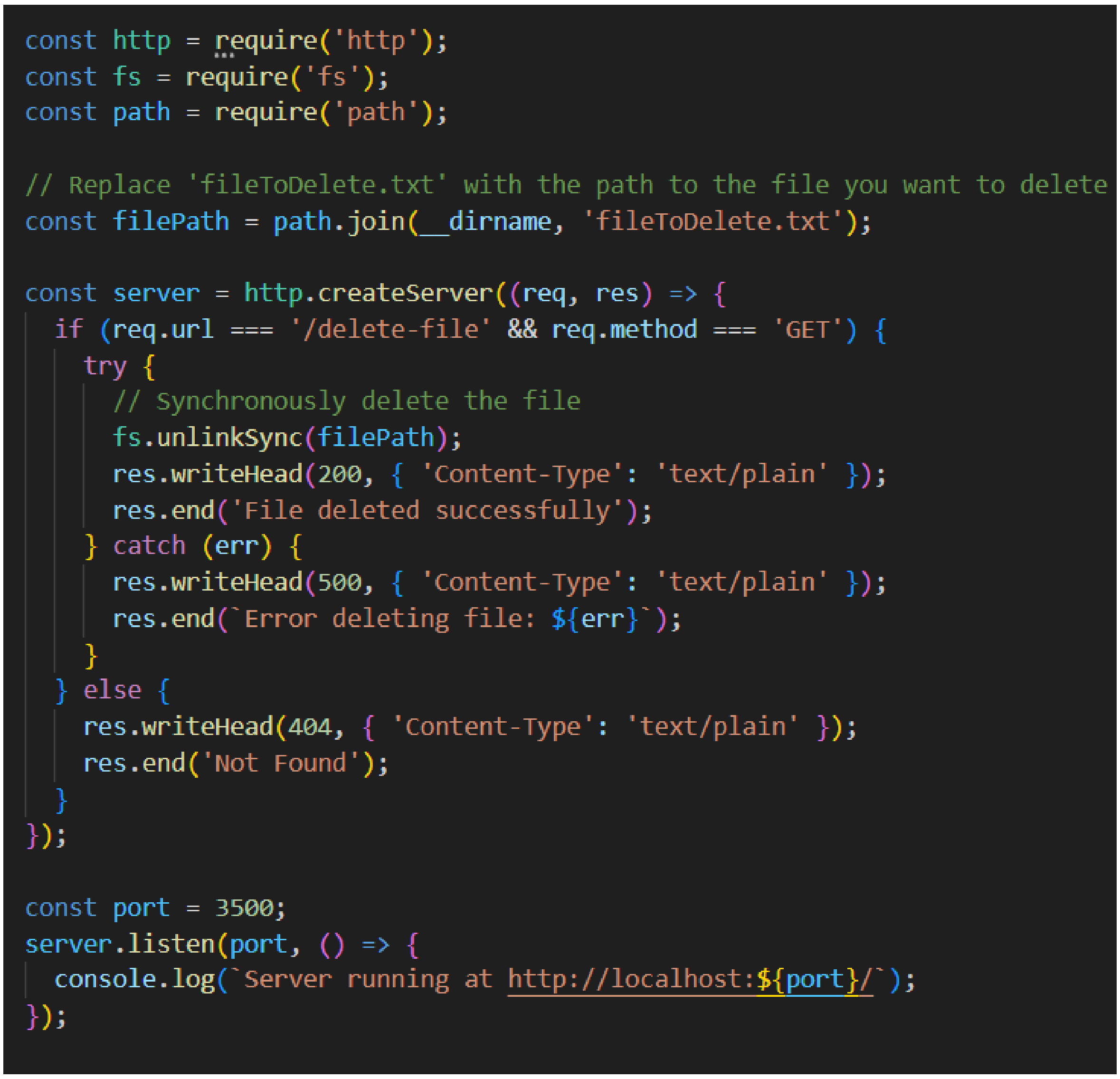
```
const fs = require('fs');

// Specify the path to the file you want to delete
const filePath = 'path/to/your/file.txt';

// Asynchronous function to delete the file
fs.unlink(filePath, (err) => {
    if (err) {
        console.error(`Error deleting file: ${err.message}`);
    } else {
        console.log('File deleted successfully!');
    }
});
```



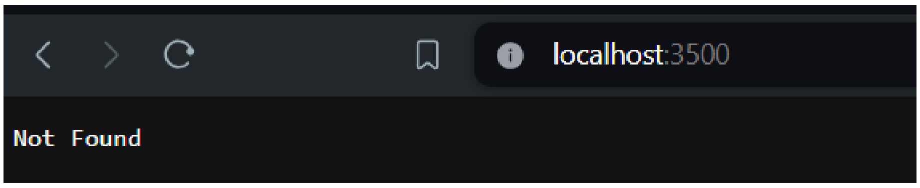
```
PS D:\node> node nodepractical
Error deleting file: ENOENT: no such file or directory, unlink 'D:\node\path\to\your\file.txt'
```

- Replace 'path/to/your/file.txt' with the actual path to the file you want to delete.
- The fs.unlink() function is used to delete the specified file asynchronously.
- If an error occurs during deletion, it will be logged to the console.
- Otherwise, a success message will be displayed.

Q57. Write a Node JS program to demonstrated deleting a file through synchronous function inside a server.



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

// Replace 'fileToDelete.txt' with the path to the file you want to delete
const filePath = path.join(__dirname, 'fileToDelete.txt');

const server = http.createServer((req, res) => {
  if (req.url === '/delete-file' && req.method === 'GET') {
    try {
      // Synchronously delete the file
      fs.unlinkSync(filePath);
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end('File deleted successfully');
    } catch (err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      res.end(`Error deleting file: ${err}`);
    }
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found');
  }
});

const port = 3500;
server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```



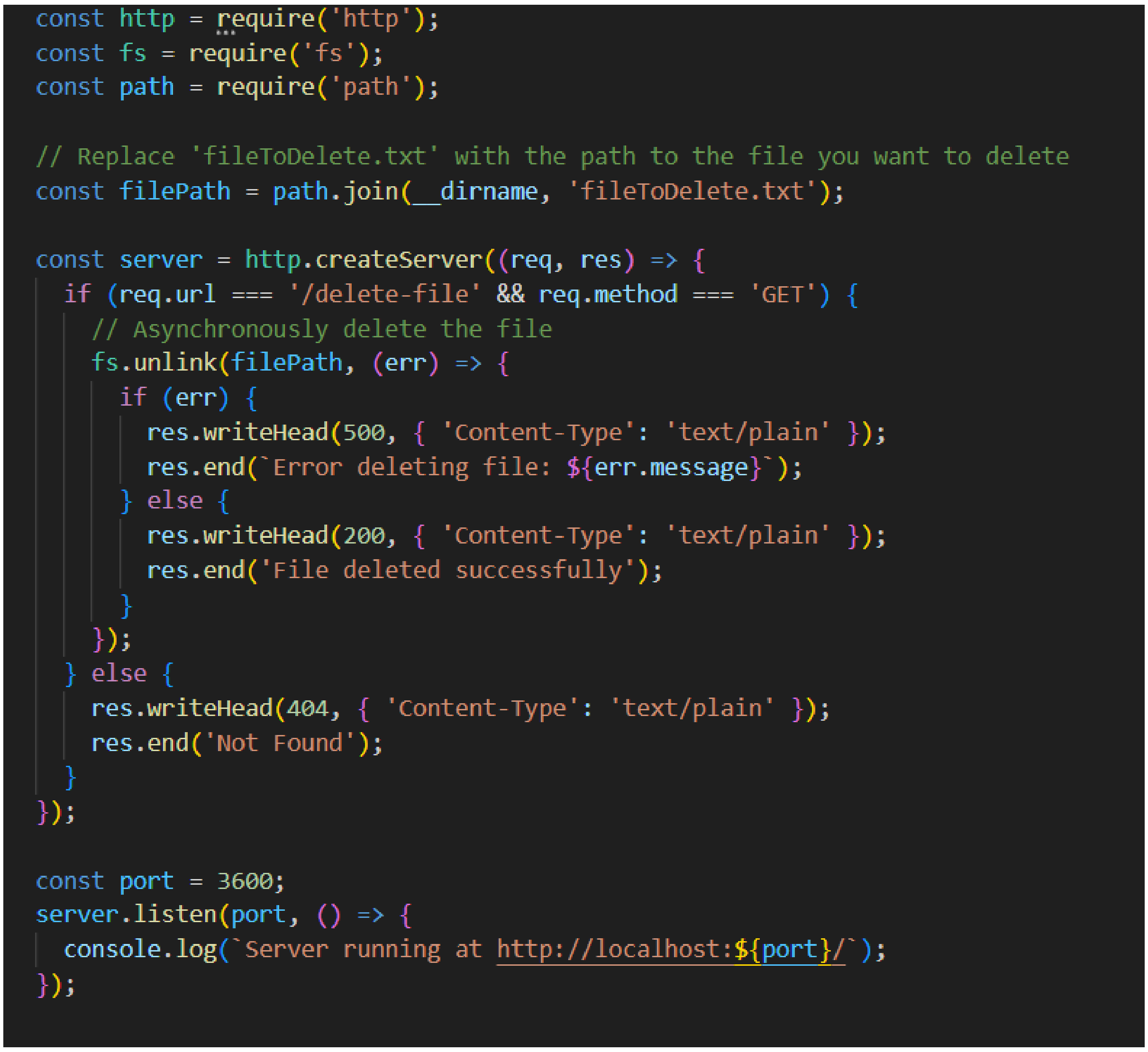
```
Not Found
```

To test this server:

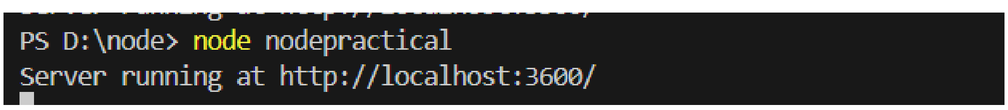
- Save the code in a file named server.js.
- Run the server using the command node server.js.
- Access http://localhost:3500/delete-file from your browser or a tool like curl.

Q58. Write a Node JS program to demonstrated deleting a file through asynchronous function inside a server.

Asynchronous function inside a server-



```javascript
const http = require('http');
const fs = require('fs');
const path = require('path');

// Replace 'fileToDelete.txt' with the path to the file you want to delete
const filePath = path.join(__dirname, 'fileToDelete.txt');

const server = http.createServer((req, res) => {
  if (req.url === '/delete-file' && req.method === 'GET') {
    // Asynchronously delete the file
    fs.unlink(filePath, (err) => {
      if (err) {
        res.writeHead(500, { 'Content-Type': 'text/plain' });
        res.end(`Error deleting file: ${err.message}`);
      } else {
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end('File deleted successfully');
      }
    });
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found');
  }
});

const port = 3600;
server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

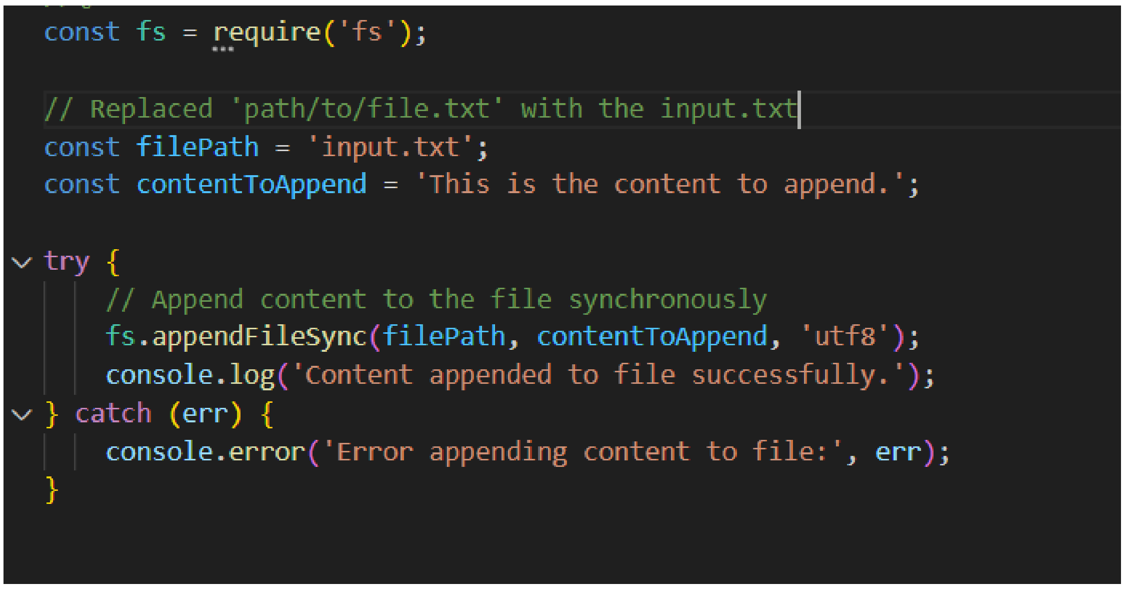


```
PS D:\node> node nodepractical
Server running at http://localhost:3600/
```

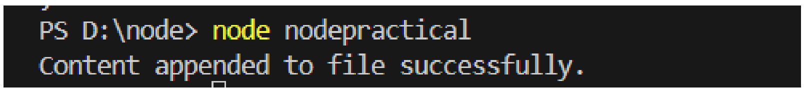
To test this server:

- Save the code in a file named server.js.
- Run the server using the command node server.js.
- Access http://localhost:3600/delete-file from your browser or a tool like curl.

Q59. Write a Node JS program to demonstrated appending a content in file through synchronous function.



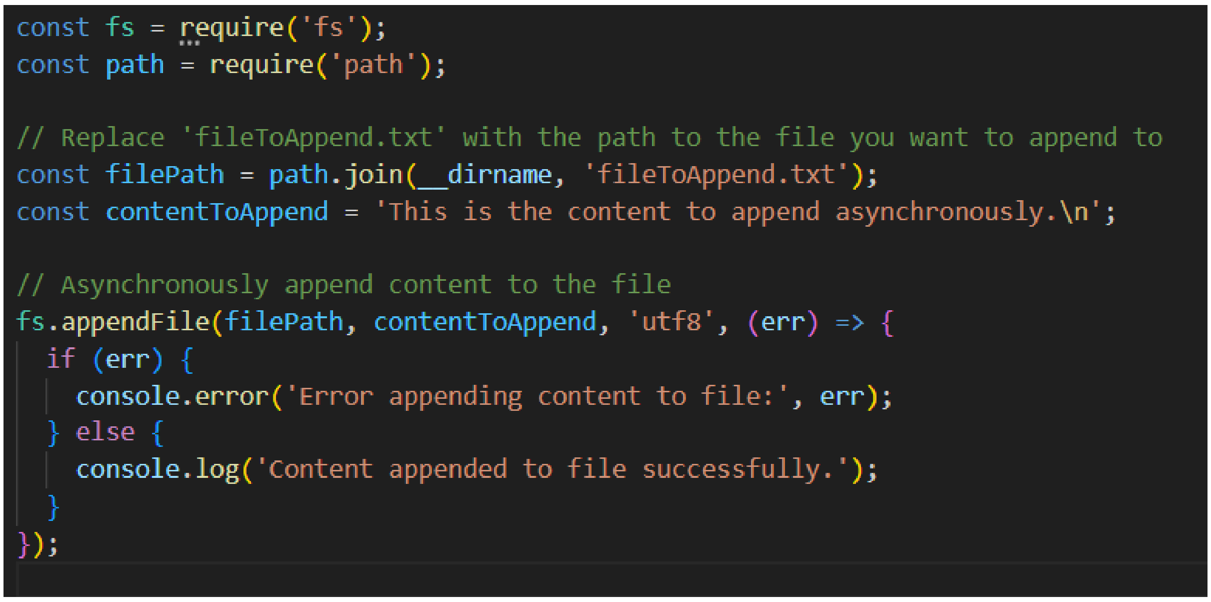
```
const fs = require('fs');

// Replaced 'path/to/file.txt' with the input.txt
const filePath = 'input.txt';
const contentToAppend = 'This is the content to append.';

try {
    // Append content to the file synchronously
    fs.appendFileSync(filePath, contentToAppend, 'utf8');
    console.log('Content appended to file successfully.');
} catch (err) {
    console.error('Error appending content to file:', err);
}
```



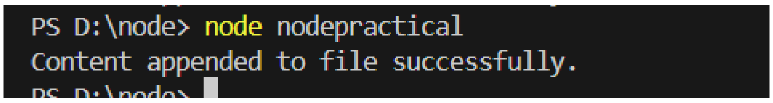
```
PS D:\node> node nodepractical
Content appended to file successfully.
```

- Replace 'path/to/input.txt' with the actual path to the file you want to append content to.
- The contentToAppend variable holds the string that will be added to the file.
- The fs.appendFileSync function is used for synchronous file appending. It takes the file path, the content to append, and the encoding as arguments.
- The try-catch block is used to handle any errors that may occur during the file operation.

Q60. Write a Node JS program to demonstrated appending a content in file through asynchronous function.



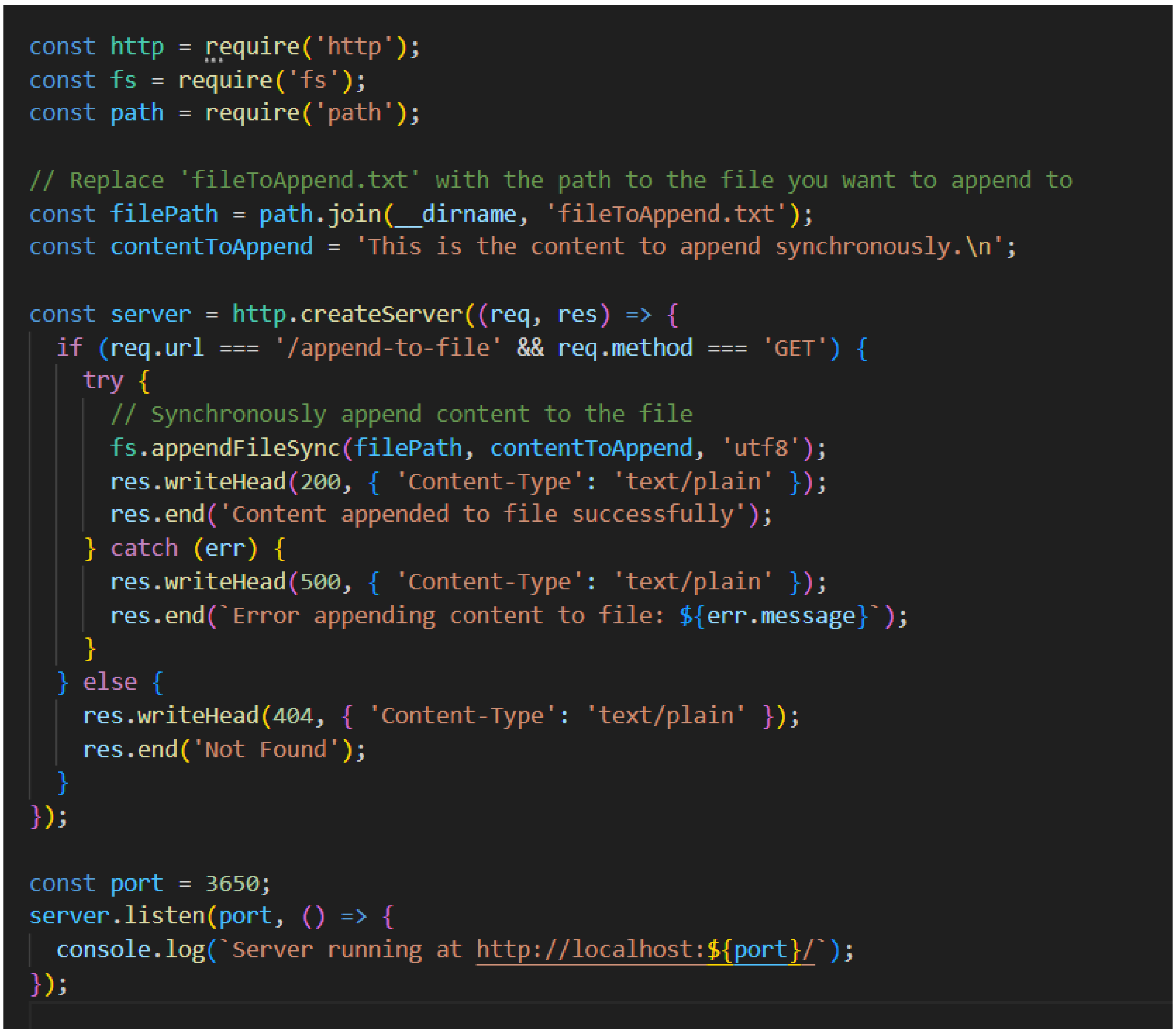
```
const fs = require('fs');
const path = require('path');

// Replace 'fileToAppend.txt' with the path to the file you want to append to
const filePath = path.join(__dirname, 'fileToAppend.txt');
const contentToAppend = 'This is the content to append asynchronously.\n';

// Asynchronously append content to the file
fs.appendFile(filePath, contentToAppend, 'utf8', (err) => {
  if (err) {
    console.error('Error appending content to file:', err);
  } else {
    console.log('Content appended to file successfully.');
  }
});
```



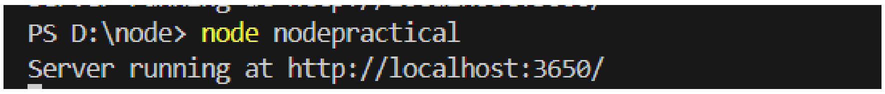
```
PS D:\node> node nodepractical
Content appended to file successfully.
PS D:\node>
```

- Replace 'fileToAppend.txt' with the actual path to the file you want to append content to.
- The contentToAppend variable holds the string that will be added to the file.
- The fs.appendFile function is used for asynchronous file appending. It takes the file path, the content to append, the encoding ('utf8' in this case), and a callback function as arguments.
- The callback function is called after the operation completes. If there's an error, it will be passed as the first argument; otherwise, the operation was successful.

Q61. Write a Node JS program to demonstrated appending a content in file through synchronous function inside a server.



```javascript
const http = require('http');
const fs = require('fs');
const path = require('path');

// Replace 'fileToAppend.txt' with the path to the file you want to append to
const filePath = path.join(__dirname, 'fileToAppend.txt');
const contentToAppend = 'This is the content to append synchronously.\n';

const server = http.createServer((req, res) => {
  if (req.url === '/append-to-file' && req.method === 'GET') {
    try {
      // Synchronously append content to the file
      fs.appendFileSync(filePath, contentToAppend, 'utf8');
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end('Content appended to file successfully');
    } catch (err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      res.end(`Error appending content to file: ${err.message}`);
    }
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found');
  }
});

const port = 3650;
server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

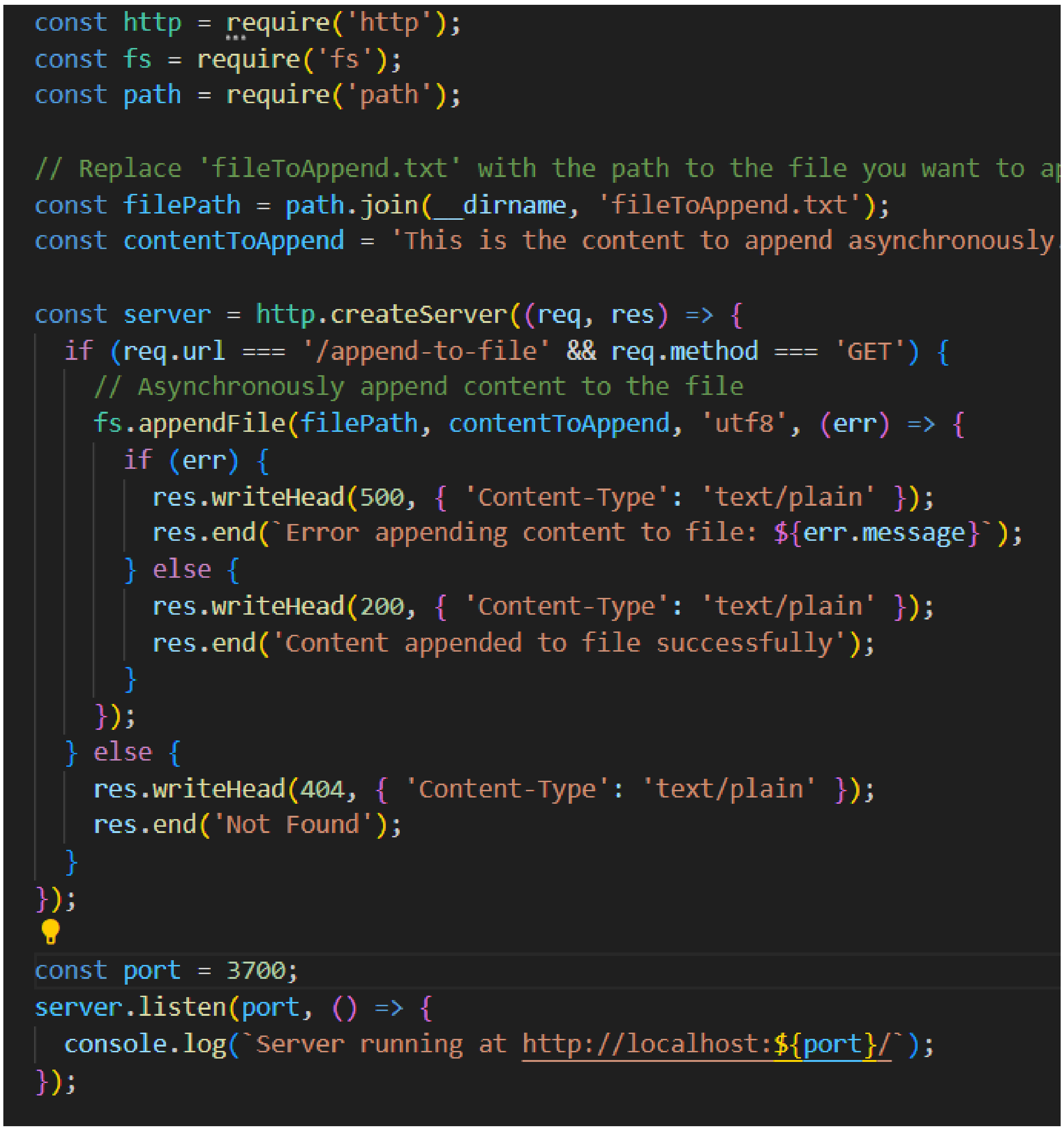


```
PS D:\node> node nodepractical
Server running at http://localhost:3650/
```

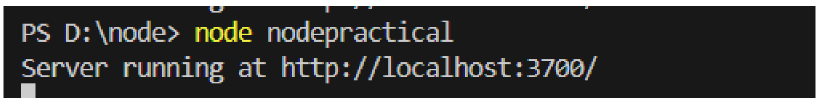
To run this program:

- Save the code in a file with .js.
- Replace 'fileToAppend.txt' with the actual path to the file you want to append content to.
- Run the server using the command node filename.js.
- Access http://localhost:3650/append-to-file from your browser or a tool like curl.

Q62. Write a Node JS program to demonstrated appending a content in file through asynchronous function inside a server.



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

// Replace 'fileToAppend.txt' with the path to the file you want to ap
const filePath = path.join(__dirname, 'fileToAppend.txt');
const contentToAppend = 'This is the content to append asynchronously

const server = http.createServer((req, res) => {
  if (req.url === '/append-to-file' && req.method === 'GET') {
    // Asynchronously append content to the file
    fs.appendFile(filePath, contentToAppend, 'utf8', (err) => {
      if (err) {
        res.writeHead(500, { 'Content-Type': 'text/plain' });
        res.end(`Error appending content to file: ${err.message}`);
      } else {
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end('Content appended to file successfully');
      }
    });
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found');
  }
});

const port = 3700;
server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

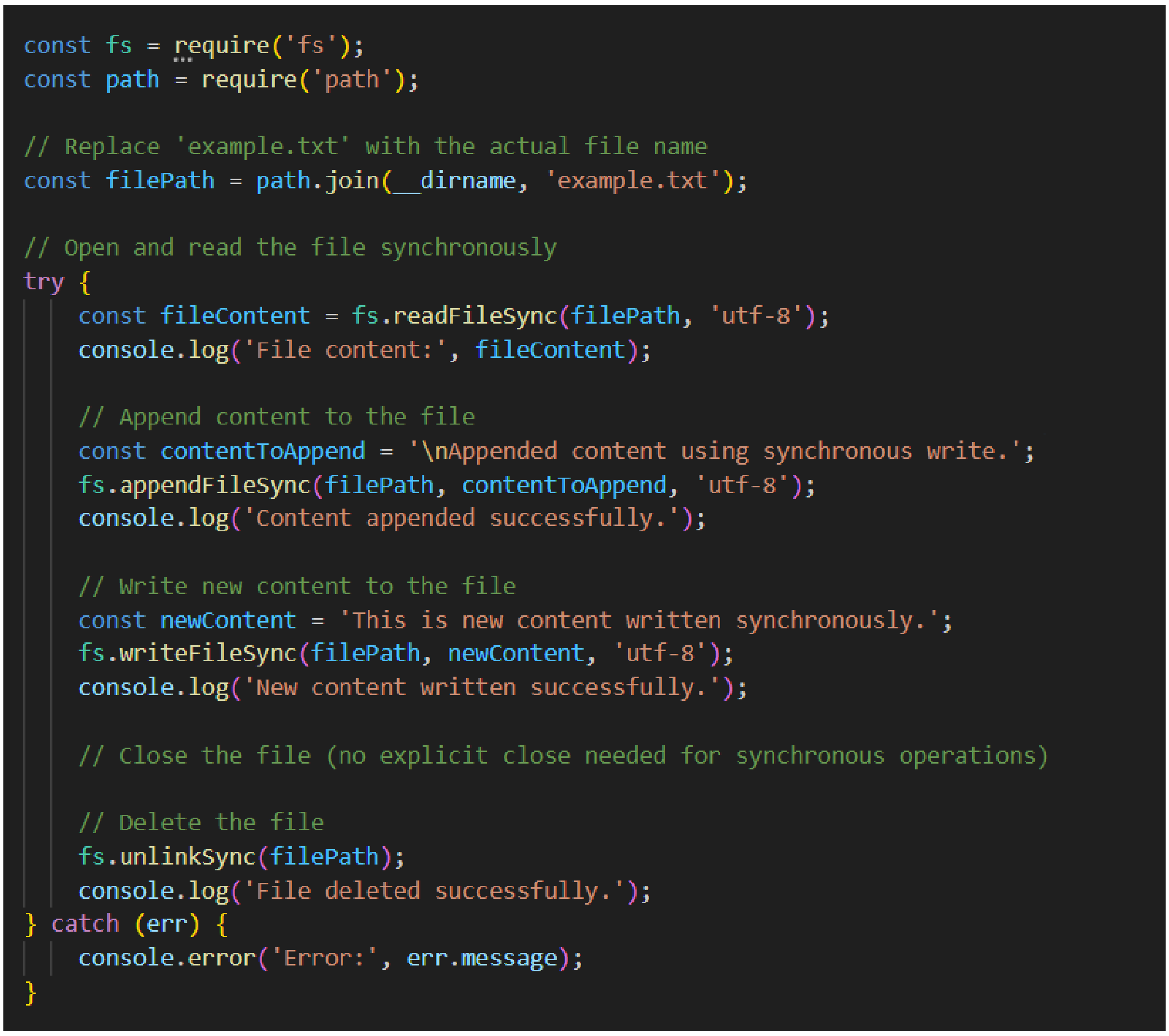


```
PS D:\node> node nodepractical
Server running at http://localhost:3700/
```

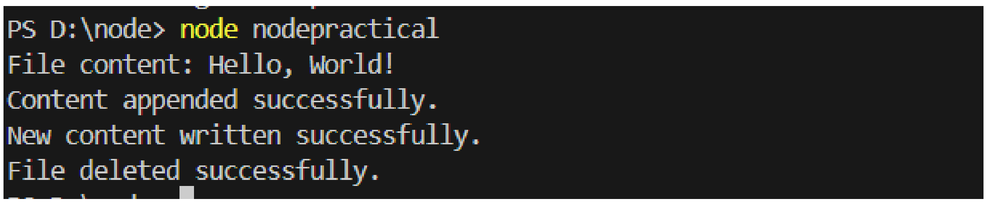
To test this code:

- Save the code in a file with .js extension(nodepractical).
- Replace 'fileToAppend.txt' with the actual path to the file you want to append content to.
- Run the server using the command node practical.js or node practical.
- Access http://localhost:3700/append-to-file from your browser or a tool like curl.

Q63. Write a complete file handling operation(open, read, write, append, close, delete) Node JS program using synchronous function.



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

// Replace 'example.txt' with the actual file name
const filePath = path.join(__dirname, 'example.txt');

// Open and read the file synchronously
try {
    const fileContent = fs.readFileSync(filePath, 'utf-8');
    console.log('File content:', fileContent);

    // Append content to the file
    const contentToAppend = '\nAppended content using synchronous write.';
    fs.appendFileSync(filePath, contentToAppend, 'utf-8');
    console.log('Content appended successfully.');

    // Write new content to the file
    const newContent = 'This is new content written synchronously.';
    fs.writeFileSync(filePath, newContent, 'utf-8');
    console.log('New content written successfully.');

    // Close the file (no explicit close needed for synchronous operations)

    // Delete the file
    fs.unlinkSync(filePath);
    console.log('File deleted successfully.');
} catch (err) {
    console.error('Error:', err.message);
}
```

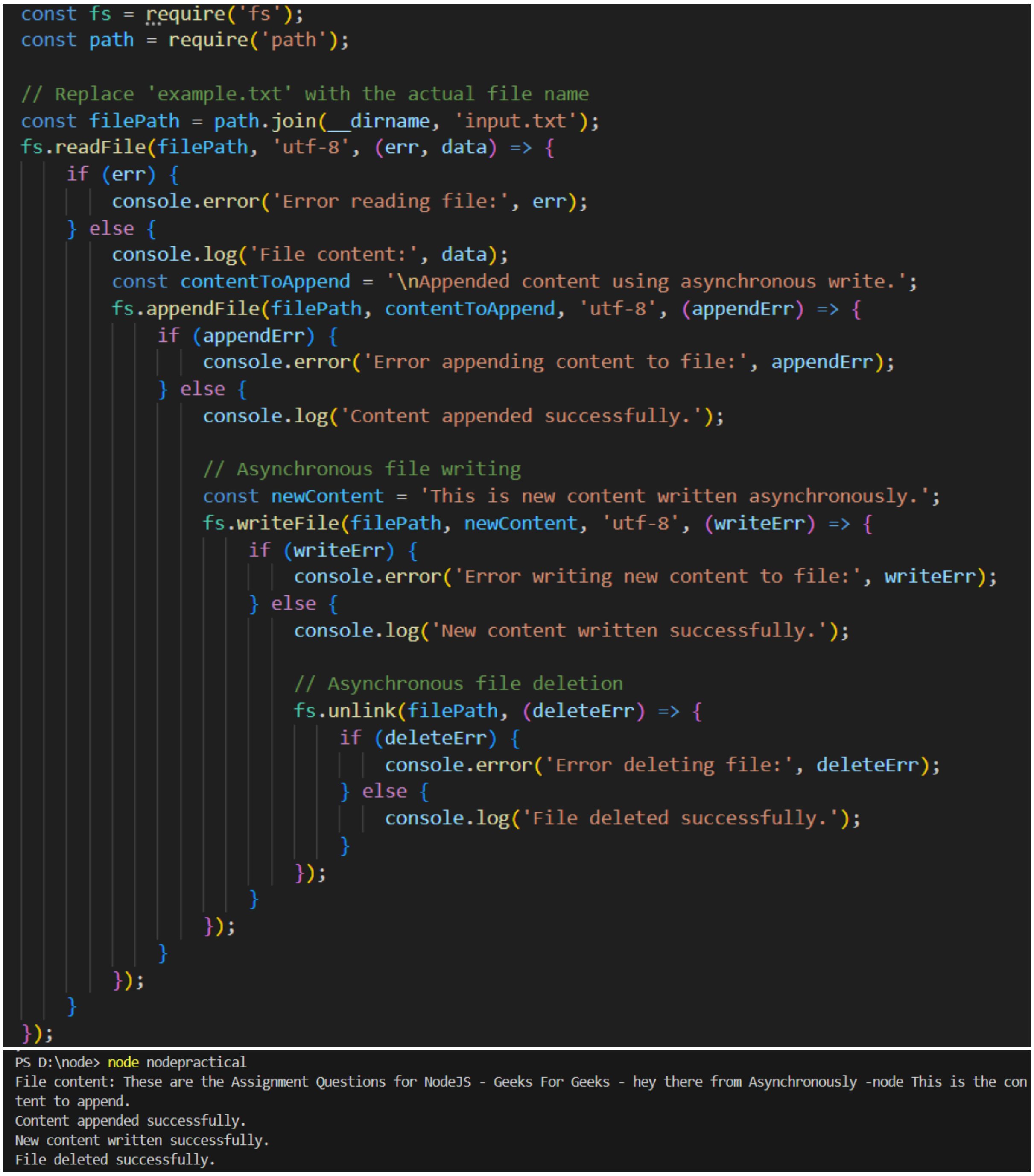


```
PS D:\node> node nodepractical
File content: Hello, World!
Content appended successfully.
New content written successfully.
File deleted successfully.
```

In this program:

- Replace 'example.txt' with the actual file name you want to work with.
- The fs.readFileSync method reads the file synchronously.
- The fs.appendFileSync method appends content to the file synchronously.
- The fs.writeFileSync method writes new content to the file synchronously.
- The fs.unlinkSync method deletes the file synchronously.

Q64. Write a complete file handling operation (open, read, write, append, close, delete) Node JS program using asynchronous function.



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

// Replace 'example.txt' with the actual file name
const filePath = path.join(__dirname, 'input.txt');
fs.readFile(filePath, 'utf-8', (err, data) => {
    if (err) {
        console.error('Error reading file:', err);
    } else {
        console.log('File content:', data);
        const contentToAppend = '\nAppended content using asynchronous write.';
        fs.appendFile(filePath, contentToAppend, 'utf-8', (appendErr) => {
            if (appendErr) {
                console.error('Error appending content to file:', appendErr);
            } else {
                console.log('Content appended successfully.');

                // Asynchronous file writing
                const newContent = 'This is new content written asynchronously.';
                fs.writeFile(filePath, newContent, 'utf-8', (writeErr) => {
                    if (writeErr) {
                        console.error('Error writing new content to file:', writeErr);
                    } else {
                        console.log('New content written successfully.');

                        // Asynchronous file deletion
                        fs.unlink(filePath, (deleteErr) => {
                            if (deleteErr) {
                                console.error('Error deleting file:', deleteErr);
                            } else {
                                console.log('File deleted successfully.');
                            }
                        });
                    }
                });
            }
        });
    }
});
```

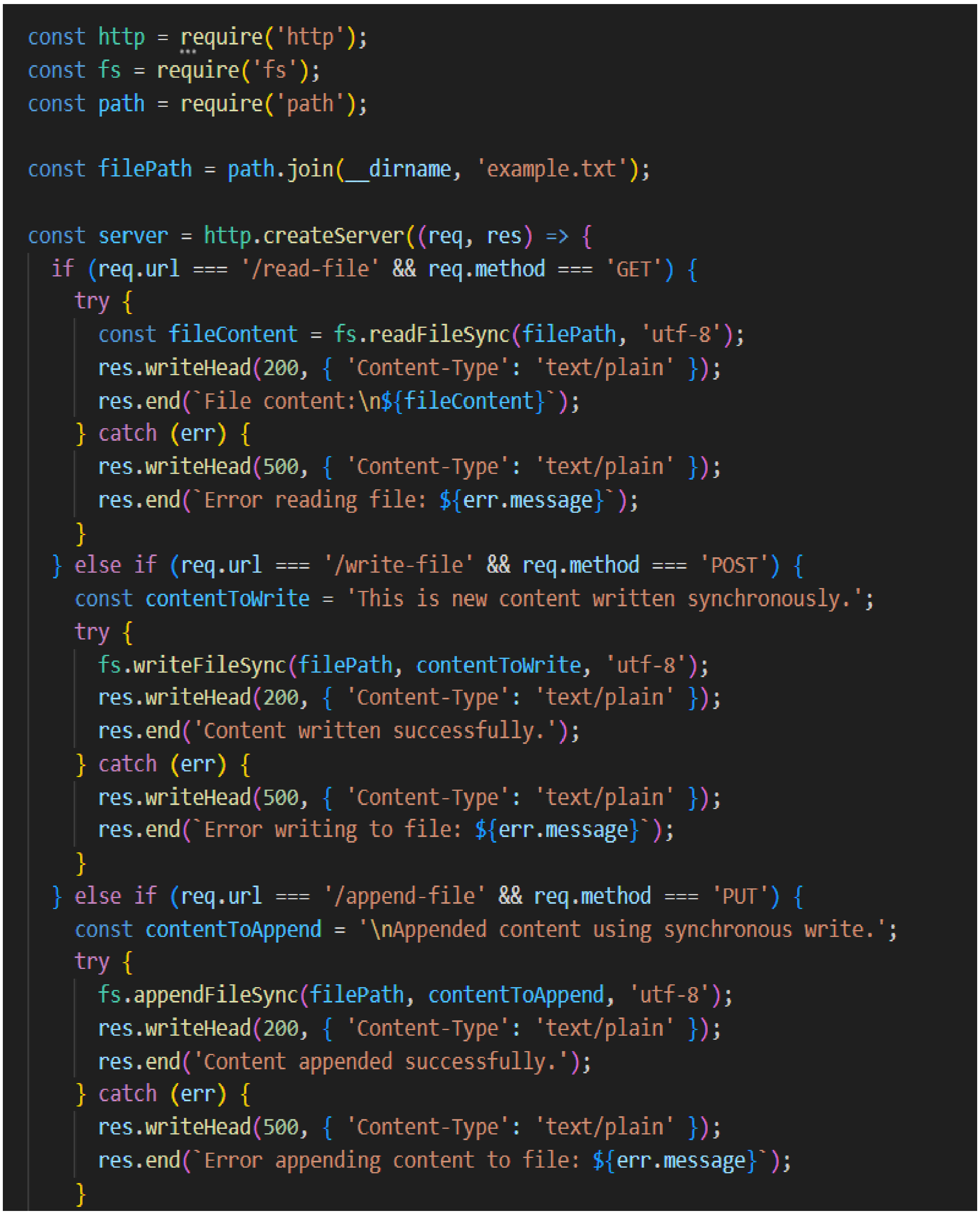
```
PS D:\node> node nodepractical
File content: These are the Assignment Questions for NodeJS - Geeks For Geeks - hey there from Asynchronously -node This is the con
tent to append.
Content appended successfully.
New content written successfully.
File deleted successfully.
```

In this program:

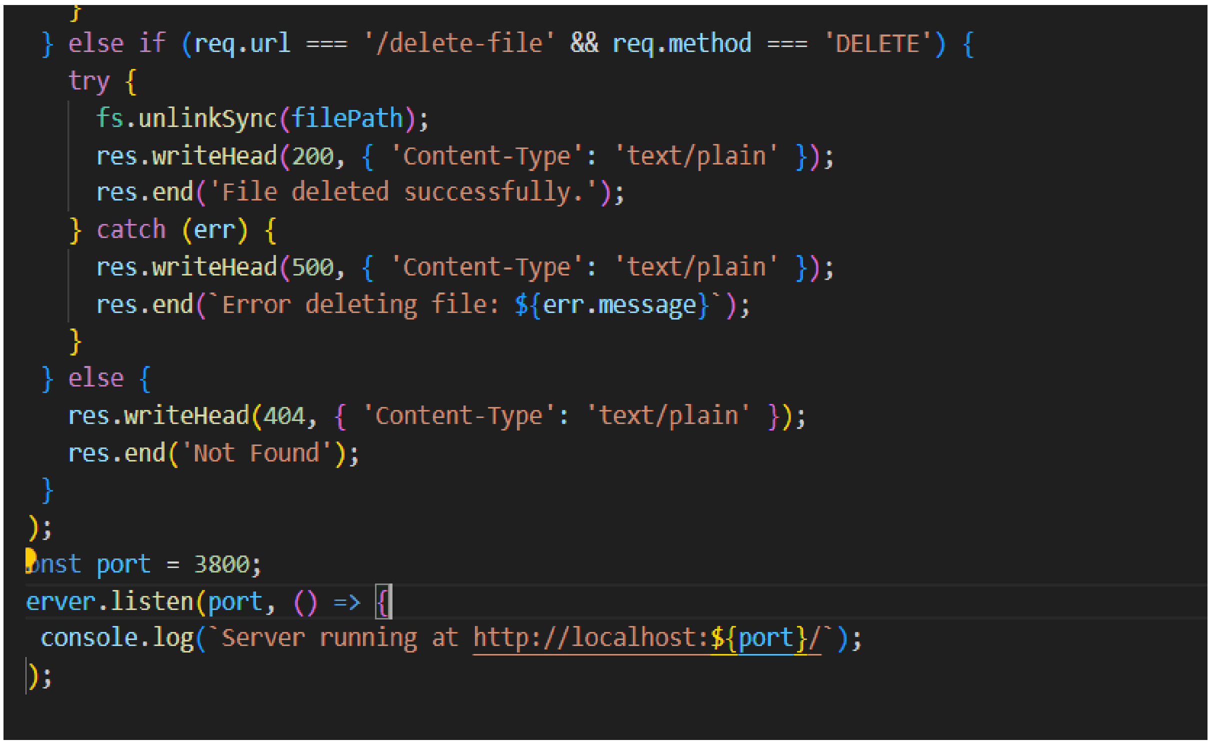
- Replace 'input.txt' with the actual file name you want to work with.
- The fs.readFile method reads the file asynchronously.
- The fs.appendFile method appends content to the file asynchronously.
- The fs.writeFile method writes new content to the file asynchronously.
- The fs.unlink method deletes the file asynchronously.

Q65. Write a complete file handling operation(open, read, write, append, close, delete) Node JS program using synchronous function inside a server.

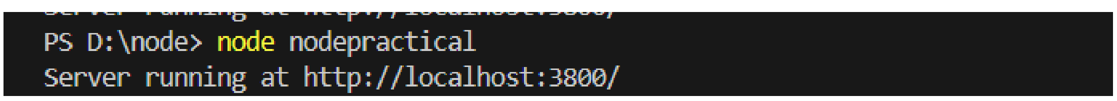
Synchronous Function:



```javascript
const http = require('http');
const fs = require('fs');
const path = require('path');

const filePath = path.join(__dirname, 'example.txt');

const server = http.createServer((req, res) => {
  if (req.url === '/read-file' && req.method === 'GET') {
    try {
      const fileContent = fs.readFileSync(filePath, 'utf-8');
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end(`File content:\n${fileContent}`);
    } catch (err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      res.end(`Error reading file: ${err.message}`);
    }
  } else if (req.url === '/write-file' && req.method === 'POST') {
    const contentToWrite = 'This is new content written synchronously.';
    try {
      fs.writeFileSync(filePath, contentToWrite, 'utf-8');
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end('Content written successfully.');
    } catch (err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      res.end(`Error writing to file: ${err.message}`);
    }
  } else if (req.url === '/append-file' && req.method === 'PUT') {
    const contentToAppend = '\nAppended content using synchronous write.';
    try {
      fs.appendFileSync(filePath, contentToAppend, 'utf-8');
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end('Content appended successfully.');
    } catch (err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      res.end(`Error appending content to file: ${err.message}`);
    }
```



```
  } else if (req.url === '/delete-file' && req.method === 'DELETE') {
    try {
      fs.unlinkSync(filePath);
      res.writeHead(200, { 'Content-Type': 'text/plain' });
      res.end('File deleted successfully.');
    } catch (err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      res.end(`Error deleting file: ${err.message}`);
    }
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found');
  }
);
onst port = 3800;
erver.listen(port, () => {
 console.log(`Server running at http://localhost:${port}/`);
);
```

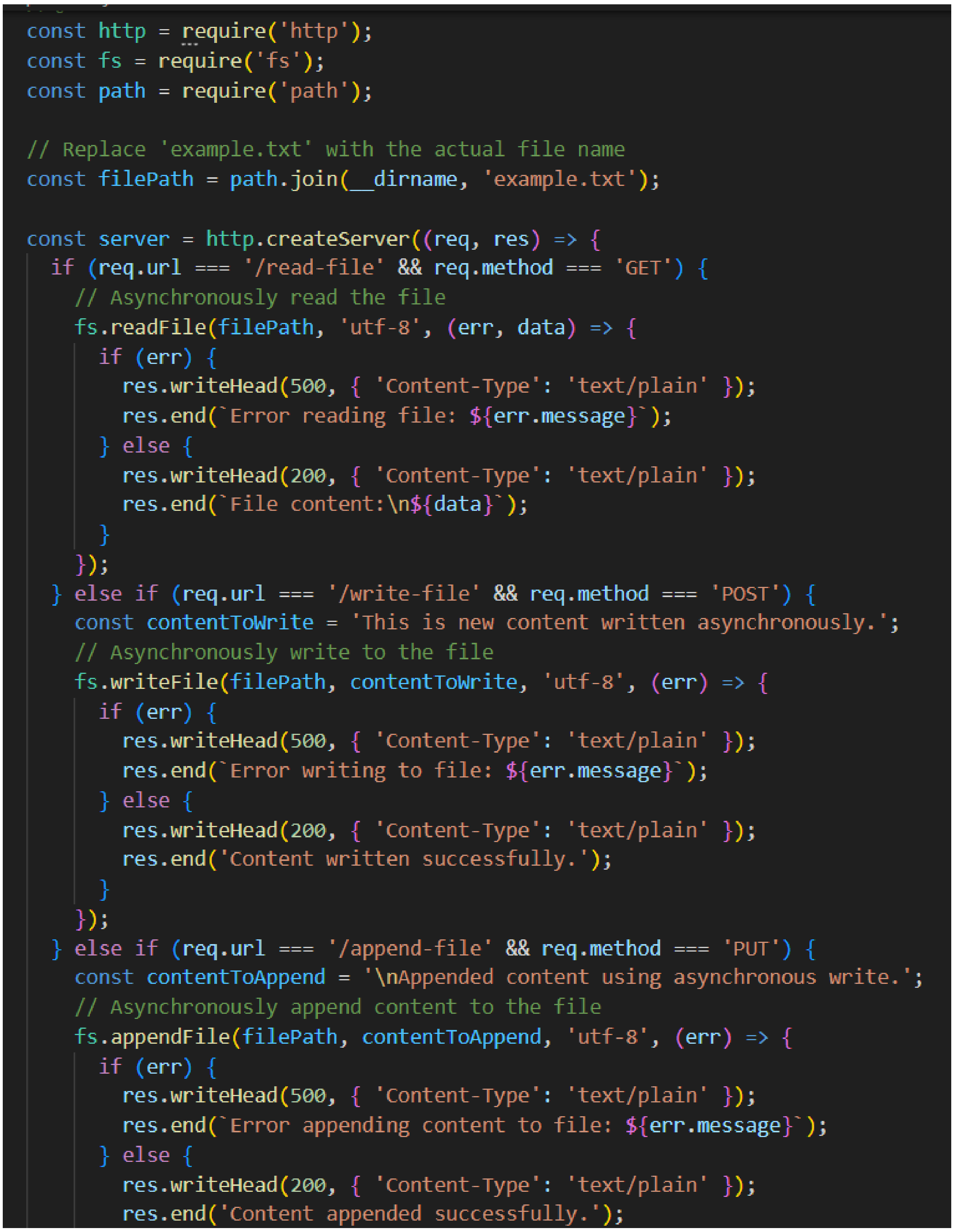


```
PS D:\node> node nodepractical
Server running at http://localhost:3800/
```

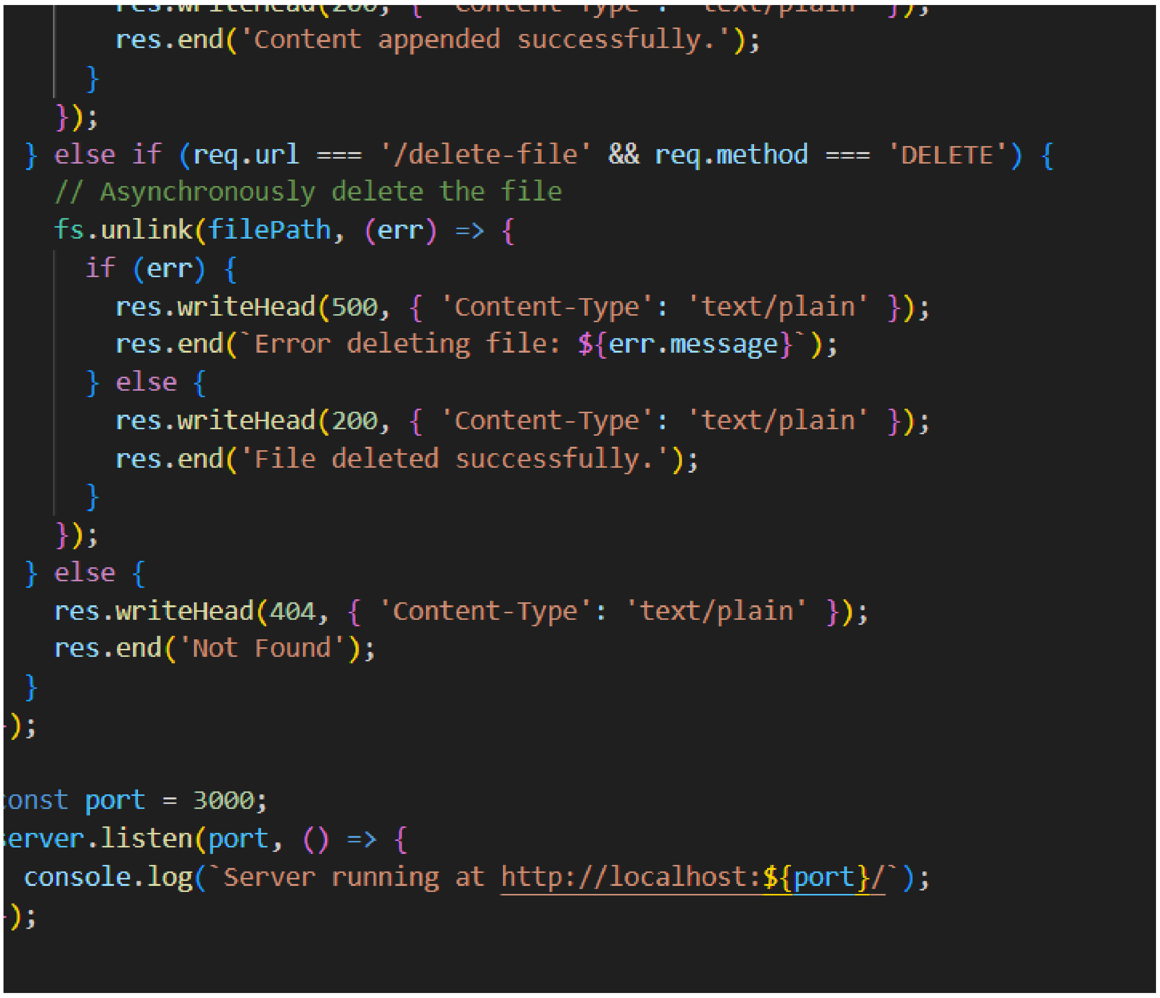
To test this server:

- Save the code in a file named server.js.
- Replace 'example.txt' with the actual path to the file you want to work with.
- Run the server using the command node server.js.
- Access the following endpoints:
- GET http://localhost:3800/read-file to read the file content.
- POST http://localhost:3800/write-file to write new content to the file.
- PUT http://localhost:3800/append-file to append content to the file.
- DELETE http://localhost:3800/delete-file to delete the file.

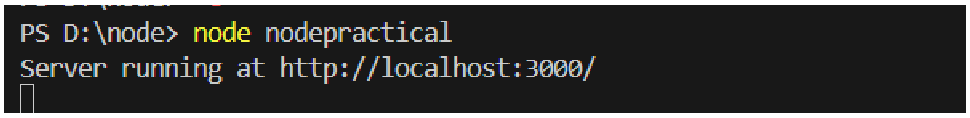
Q66. Write a complete file handling operation (open, read, write, append, close, delete) Node JS program using asynchronous function inside a server.



```javascript
const http = require('http');
const fs = require('fs');
const path = require('path');

// Replace 'example.txt' with the actual file name
const filePath = path.join(__dirname, 'example.txt');

const server = http.createServer((req, res) => {
  if (req.url === '/read-file' && req.method === 'GET') {
    // Asynchronously read the file
    fs.readFile(filePath, 'utf-8', (err, data) => {
      if (err) {
        res.writeHead(500, { 'Content-Type': 'text/plain' });
        res.end(`Error reading file: ${err.message}`);
      } else {
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end(`File content:\n${data}`);
      }
    });
  } else if (req.url === '/write-file' && req.method === 'POST') {
    const contentToWrite = 'This is new content written asynchronously.';
    // Asynchronously write to the file
    fs.writeFile(filePath, contentToWrite, 'utf-8', (err) => {
      if (err) {
        res.writeHead(500, { 'Content-Type': 'text/plain' });
        res.end(`Error writing to file: ${err.message}`);
      } else {
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end('Content written successfully.');
      }
    });
  } else if (req.url === '/append-file' && req.method === 'PUT') {
    const contentToAppend = '\nAppended content using asynchronous write.';
    // Asynchronously append content to the file
    fs.appendFile(filePath, contentToAppend, 'utf-8', (err) => {
      if (err) {
        res.writeHead(500, { 'Content-Type': 'text/plain' });
        res.end(`Error appending content to file: ${err.message}`);
      } else {
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end('Content appended successfully.');
```



```
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end('Content appended successfully.');
      }
    });
  } else if (req.url === '/delete-file' && req.method === 'DELETE') {
    // Asynchronously delete the file
    fs.unlink(filePath, (err) => {
      if (err) {
        res.writeHead(500, { 'Content-Type': 'text/plain' });
        res.end(`Error deleting file: ${err.message}`);
      } else {
        res.writeHead(200, { 'Content-Type': 'text/plain' });
        res.end('File deleted successfully.');
      }
    });
  } else {
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('Not Found');
  }
);

onst port = 3000;
erver.listen(port, () => {
 console.log(`Server running at http://localhost:${port}/`);
);
```



```
PS D:\node> node nodepractical
Server running at http://localhost:3000/
```

To test this server:

- Save the code in a file named server.js.
- Replace 'example.txt' with the actual path to the file you want to work with.
- Run the server using the command node server.js.
- Access the following endpoints:
- GET http://localhost:3000/read-file to read the file content.
- POST http://localhost:3000/write-file to write new content to the file.
- PUT http://localhost:3000/append-file to append content to the file.
- DELETE http://localhost:3000/delete-file to delete the file.

Q67. What is NPM. Explain it advantages through example.

NPM: npm stands for Node Package Manager. It’s a library and registry for JavaScript software packages.

npm also has command-line tools to help you install the different packages and manage their dependencies.

npm comes with Node.js, which means you have to install Node.js to get installed automatically on your personal computer.

npm is free and relied on by over 11 million developers worldwide. You could say it’s kind of a big deal. They’re open-source and have become the center of Javascript code sharing. There are more than a million packages available on npm.

In short, npm is:

- an online repository for the publishing of open-source Node.js projects
- a command-line utility for interacting with said repository helping with installing packages and managing package versions and dependencies.

Components of npm

npm mainly consists of three different components, these are:

- **Website:** The npm official website is used to find packages for your project, create and set up profiles to manage and access private and public packages.
- **Command Line Interface (CLI):** The CLI runs from your computer's terminal to interact with npm packages and repositories.
- **Registry:** The registry is a large and public database of JavaScript projects and meta-information. You can use any supported npm registry that you want or even your own. You can even use someone else's registry as per their terms of use.

Advantages of NPM:

- **Ease of Use:** NPM makes it incredibly easy to install, update, and manage dependencies in your projects with simple commands.

- **Vast Registry:** The NPM registry hosts a massive collection of packages, which means you can find a package for almost any functionality you need.
- **Semantic Versioning:** NPM uses semantic versioning to manage package versions, ensuring that updates are consistent and do not break existing code.
- **Package.json:** This file holds various metadata relevant to the project, which serves as a manifest about the project's dependencies, scripts, and more.
- **Reusable Code:** With NPM, developers can reuse code across projects, which saves time and effort.
- **Community Support:** A large community of developers contributes to and uses NPM, providing a robust support network.

Imagine you're building a Node.js application and you need to use Express.js, a web application framework. Instead of writing the entire framework from scratch, you can simply run:

Syntax: **npm install express**

This command will download Express.js and its dependencies into your project, allowing you to use its features immediately. The package.json file will be updated to include Express.js as a dependency, which means anyone else working on the project can install all necessary packages with just:

Syntax: **npm install**

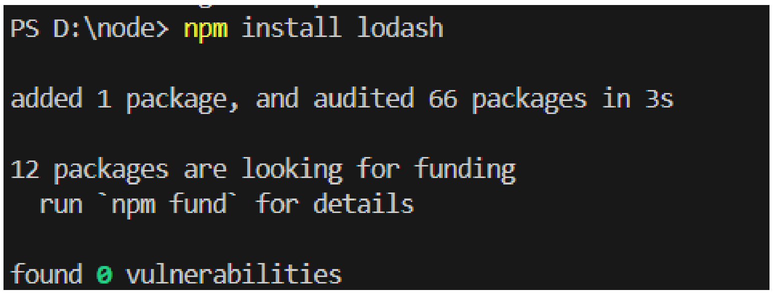
Q68. Explain local, global and dependencies based installation process through syntax and example.

- **ocal Installation:**
- When you install a package locally, it becomes available only within the specific project where you installed it.
- Use the following syntax to install a package locally:

npm install <package-name>

For example, if you're working on a project and want to add the popular express package, you'd run:

npm install lodash

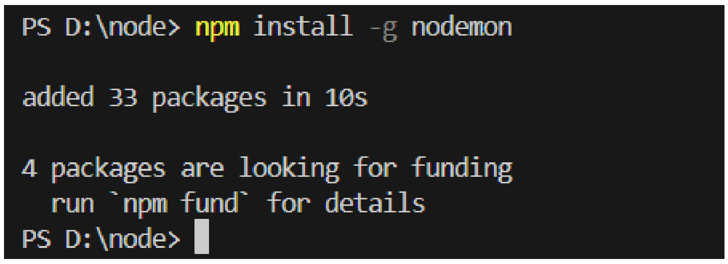
- This command will create a node_modules folder within your project directory and install lodash there.
- 

```
PS D:\node> npm install lodash

added 1 package, and audited 66 packages in 3s

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Global Installation:

- Global installation places packages in a central location on your system, making them accessible from any project.
- Use the -g flag to install a package globally:
- For instance, if you want to install the nodemon package globally (a handy tool for automatically restarting your Node.js server during development), you'd run:

Syntax: **npm install -g <package-name>**



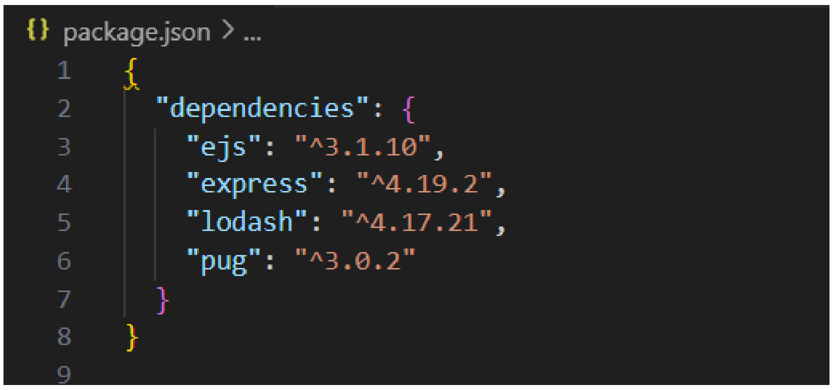
```
PS D:\node> npm install -g nodemon

added 33 packages in 10s

4 packages are looking for funding
  run `npm fund` for details
PS D:\node>
```

- Keep in mind that global installations are less common because they can lead to dependency conflicts. It's generally better to use local installations for project-specific dependencies.

Dependencies in package.json:

- The package.json file in your project contains metadata about your project, including its dependencies.
- You can specify whether a package should be installed globally or locally using the preferGlobal property in package.json.

Example:



```
{} package.json > ...
{
  "dependencies": {
    "ejs": "^3.1.10",
    "express": "^4.19.2",
    "lodash": "^4.17.21",
    "pug": "^3.0.2"
  }
}
```

In this example, express will be installed locally.

Q69. Explain local, global and dependencies based un-installation process through syntax and example.

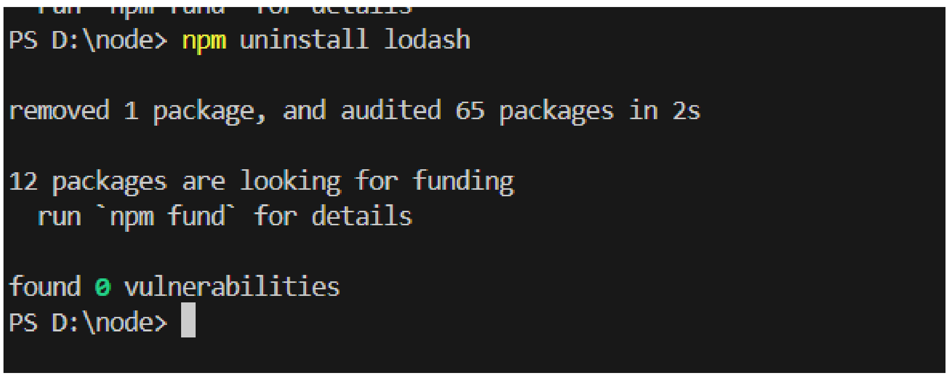
Uninstalling packages in Node.js can be done locally within a project, globally across the system, or by removing specific dependencies.

Local Uninstallation:

- To uninstall a package locally, navigate to your project directory and use the following command:

npm uninstall <package-name>

Example:



```
PS D:\node> npm uninstall lodash

removed 1 package, and audited 65 packages in 2s

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS D:\node>
```

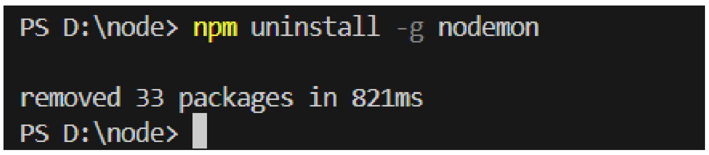
- This will remove the lodash package from the local node_modules directory and update the package.json and package-lock.json files accordingly.

Global Uninstallation:

- To uninstall a package globally, use the -g flag with the uninstall command:

npm uninstall -g <package-name>

Example:



```
PS D:\node> npm uninstall -g nodemon

removed 33 packages in 821ms
PS D:\node>
```

- This will remove the create-react-app package from the global node_modules directory, making it no longer available system-wide.

Uninstalling Dependencies:

If you want to remove a dependency from your project, you can manually edit the **package.json** file to remove the unwanted package from the dependencies list and then run:

```
npm install
```

This will install all the dependencies listed in the package.json file, excluding the one you removed, effectively uninstalling it from your project.

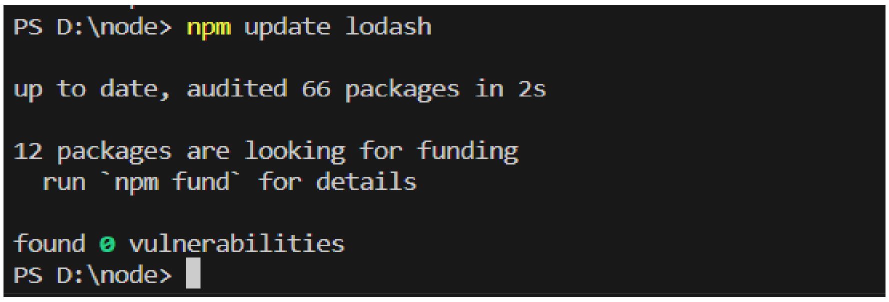
Q70. Explain local, global and dependencies based update process through syntax and example.

Local Update (Dependencies):

- When you want to update the dependencies listed in your project's package.json, follow these steps:
- Navigate to your project directory.
- Open the terminal or command prompt.
- Run the following command:

npm update <package-name>

- Example (local):



```
PS D:\node> npm update lodash

up to date, audited 66 packages in 2s

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
PS D:\node>
```

- on the version ranges specified in your package.json.

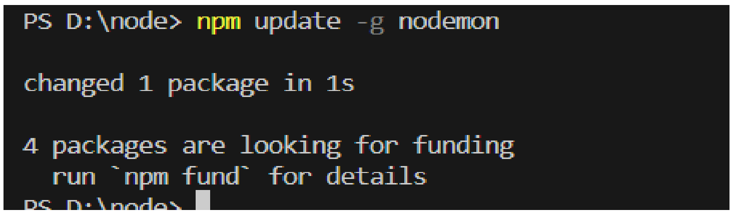
- This command will update all dependencies to their latest compatible versions based

Global Update:

- To update a global package (installed system-wide), use the -g flag:

npm update -g <package-name>

Example:



```
PS D:\node> npm update -g nodemon

changed 1 package in 1s

4 packages are looking for funding
  run `npm fund` for details
PS D:\node>
```

This will update the global nodemon package to the latest version.

Q71. What is advantages of package.json file. Explain it attributes and creating process.

The package.json file is a fundamental component of any Node.js project or module.

Advantages of package.json:

- **Project Management:** It lists the packages your project depends on, making it easy to manage and install them.
- **Version Control:** Specifies versions of a package that your project can use, using semantic versioning rules.
- **Reproducibility:** Makes your build reproducible, which is crucial for consistent development and deployment across environments.
- **Collaboration:** Facilitates collaboration by allowing other developers to understand project dependencies and scripts.
- **Publishing:** Essential for publishing your project to the NPM registry, making it available for others to use.

Key Attributes of package.json:

- **name:** The name of your project, which must be unique if you plan to publish it.
- **version:** Follows semantic versioning and is required if you plan to publish the package.
- **description:** A brief description of your project, which helps others understand its purpose.
- **main:** The entry point of your project, typically the initial file that runs when starting the application.
- **scripts: Defines script commands that can be executed at various stages of development**, like start, test, and build.
- **dependencies:** Lists the NPM packages required for your project to run in production.
- **devDependencies:** Lists the NPM packages required for development and testing, but not in production.
- **repository:** Provides information about where your project's source code resides.
- **author:** Information about the author of the project.
- **license**: Indicates the license type under which the project is provided.

Creating a package.json File:

You can create a package.json file using the following methods:

Using npm init:

Navigate to the root directory of your project.

Run npm init in the terminal.

Answer the questions in the command line questionnaire to fill out the package.json file.

Using npm init -y:

For a default package.json with values extracted from the current directory, use the --yes or -y flag with npm init.

Example:

```
npm init -y
```

This will create a package.json file with default values.

Q72. Explain the advantages of express js over to node js.

Express.js is a web application framework that sits on top of Node.js. It provides a robust set of features to develop web and mobile applications efficiently.

Advantages:

- **Simplicity:** Express.js simplifies the process of writing server code, eliminating the need to repeat the same code, as is often the case with Node.js.
- **Middleware:** Express.js uses middleware to respond to HTTP requests. This means you can define a stack of actions that you want to occur in response to a request.
- **Routing:** Express.js provides a powerful routing API that allows you to map URLs to server-side functions, making it easier to design single-page applications and RESTful APIs.
- **Speed:** Express.js is designed for building web applications quickly, which makes it a great choice for projects with tight deadlines.
- **Flexibility:** Express.js is very flexible, allowing developers to structure their applications in the way they see fit.
- **Minimalism:** Express.js is minimal and unopinionated, giving you the tools to build an application with the most freedom.
- **Community:** There is a large community of developers who use Express.js, which means there's a wealth of plugins and middlewares available to extend its capabilities.
- **Ease of Integration:** Express.js can be easily integrated with databases like MongoDB, Redis, etc., providing a seamless development experience.
- **Less Coding Time:** With Express.js, you can write less code and accomplish more compared to using Node.js alone, which requires more boilerplate code for common tasks.
- **Enhanced Developer Experience:** Express.js streamlines the development process and enhances the overall developer experience by simplifying common tasks such as handling HTTP requests and working with databases.

Q73. Explain the code structure and execution process of express js.

Code Structure in Express.js:

Project Directory:

- Create a directory for your Express.js project.
- Inside this directory, you'll organize your files and folders.

Entry Point (app.js or index.js):

- The entry point file (usually named app.js or index.js) initializes your Express application.
- It sets up the server, defines routes, and configures middleware.

Routes:

- Routes define how your application responds to different HTTP requests (GET, POST, etc.).
- Routes are organized in separate files (e.g., routes/users.js, routes/products.js).
- Each route handles specific endpoints (URL paths).

Middleware:

- Middleware functions are executed before reaching the route handler.
- They can perform tasks like authentication, logging, error handling, etc.
- Common middleware includes body-parser, cors, and custom middleware.

Controllers:

- Controllers handle the business logic for specific routes.
- They interact with models, databases, and other services.
- Controllers are often organized in separate files (e.g., controllers/userController.js).

Models:

- Models represent data structures (e.g., database tables, documents).
- They handle data retrieval, storage, and manipulation.
- Models are often organized in separate files (e.g., models/userModel.js).

Views (Optional):

Express.js is primarily used for building APIs, but you can also render views (HTML templates) using a template engine like EJS or Pug.

Views are typically stored in a views directory.

Execution Process in Express.js:

- **Initialization:**
  Create an instance of the Express application using const app = express();.
- **Middleware Configuration:**
  Configure middleware using app.use():
  Body parsing middleware (e.g., express.json(), express.urlencoded()).
  CORS middleware (e.g., cors).
  Custom middleware (e.g., authentication, logging).
- **Route Definitions:**
  Define routes using app.get(), app.post(), etc.
  Specify the route path and the corresponding controller function.
- **Controller Logic:**
  Controllers handle the logic for each route.
  They interact with models, perform data validation, and send responses.
- **Model Interaction:**
  Models interact with databases (e.g., MongoDB, MySQL) or other data sources.
  They handle data retrieval, storage, and manipulation.
- **Response:**
  Send responses using res.send(), res.json(), or res.render() (for views).
  Handle errors using next(err) or custom error handlers.
- **Server Initialization:**
  Start the server using app.listen(PORT, () => { ... }).
  The server listens for incoming requests on the specified port.

Q74. Write a express JS code to demonstrated to handle any response as html file and message.

An Express.js application that serves an HTML file and a custom message when specific routes are accessed:

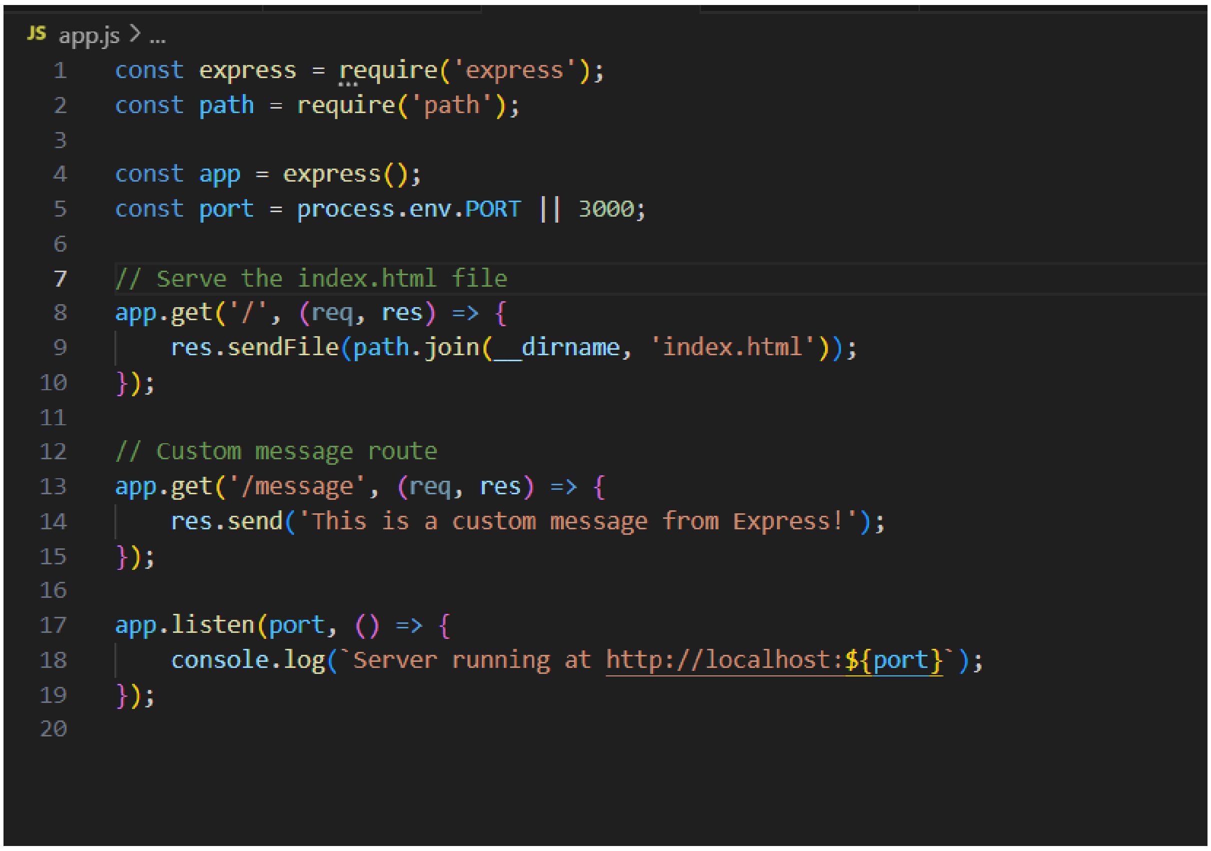
- Create a new directory for your project (e.g., express-html-example).
- Navigate to the project directory and initialize a new Node.js project:

```
mkdir express-html-example
cd express-html-example
npm init -y
```

- **Install Express**

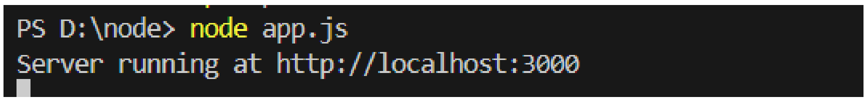
```
npm install express
```

- **Create an index.html file**
- **Create an app.js**



JS app.js > ...

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

const app = express();
const port = process.env.PORT || 3000;

// Serve the index.html file
app.get('/', (req, res) => {
    res.sendFile(path.join(__dirname, 'index.html'));
});

// Custom message route
app.get('/message', (req, res) => {
    res.send('This is a custom message from Express!');
});

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



```
PS D:\node> node app.js
Server running at http://localhost:3000
```

Q75. Explain the process of request handling of client through GET method.

In Express.js, handling a client's request through the GET method involves a few steps that define how the server responds to a specific endpoint.

Process:

- **Define a Route:** You start by defining a route using app.get(), where app is an instance of Express. The get method takes two arguments: the path and a callback function.
- **Callback Function:** The callback function has two parameters, request (often abbreviated as req) and response (often abbreviated as res). This function is executed when a GET request is made to the specified path.
- **Request Object:** The request object contains all the information about the client's request, such as URL parameters, query strings, headers, and more.
- **Response Object:** The response object is used to send back a response to the client. You can use methods like res.send(), res.json(), or res.sendFile() to send back text, JSON, or files, respectively.
- **Route Parameters:** If your route requires parameters (like /users/:userId), you can access the value passed by the client in the request URL via req.params.
- **Query Strings:** For non-parameterized data, like /search?query=express, you can access the query string via req.query.
- **Sending a Response:** Once you have processed the request, you use the response object to send a response back to the client. This could be a simple message, a JSON object, or an HTML file.

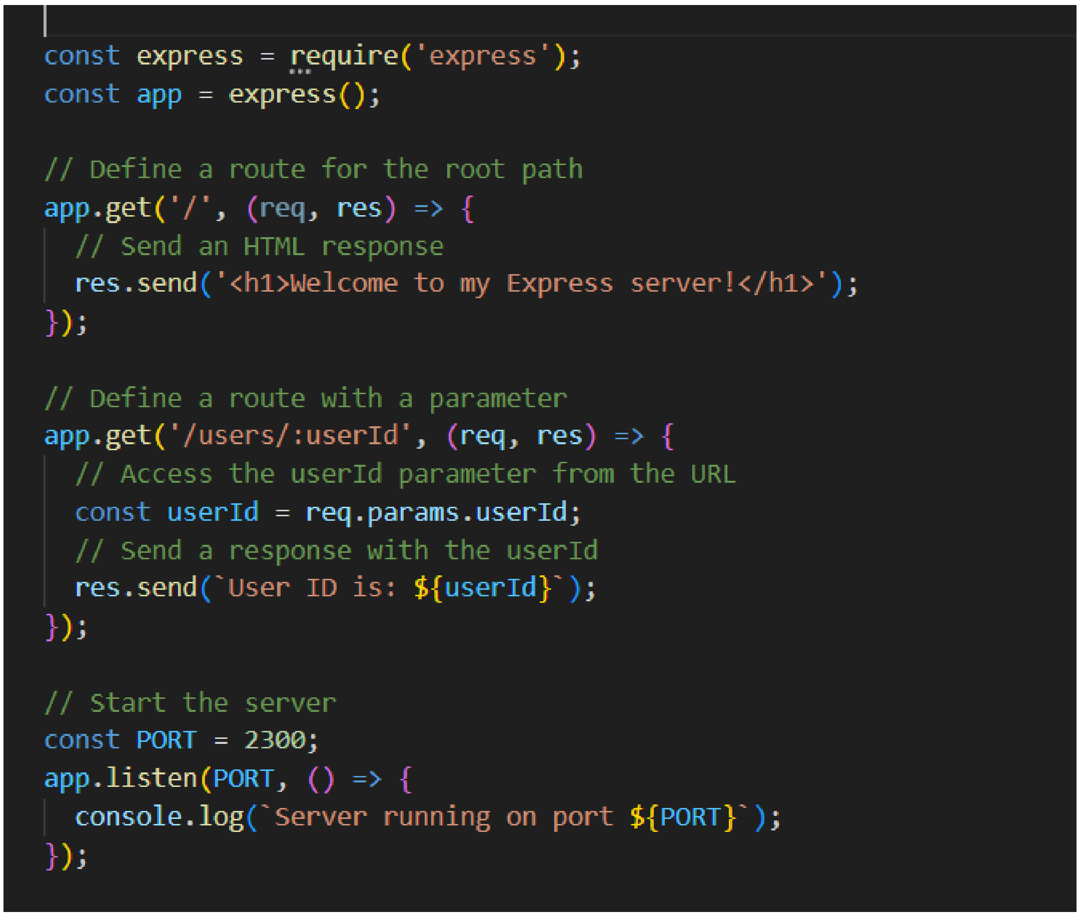
Q76. Explain the process of request handling of client through POST method.

Handling a client’s request through the POST method in Express.js involves several steps:

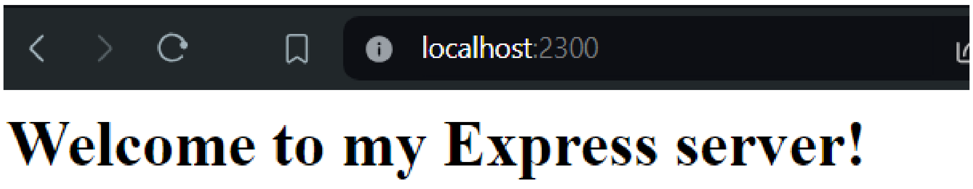
Steps:

- **Set Up Express:** Initialize an Express application by requiring the express module and creating an app instance.
- **Middleware:** Use middleware to parse the incoming request bodies. For JSON data, use express.json(), and for URL-encoded data (from forms), use express.urlencoded().
- **Define POST Route:** Define a route handler for the POST request using app.post(). Specify the path and a callback function that will handle the request.
- **Access Request Data:** Inside the callback function, access the data sent by the client through req.body. This object contains the parsed data from the request.
- **Process Request:** Use the data from req.body to perform operations such as saving to a database, validating input, or any other server-side logic.
- **Send Response:** Once the processing is complete, send a response back to the client using res.send(), res.json(), or any other response method provided by Express.
- **Error Handling:** Implement error handling within the route or by using separate error-handling middleware to catch and respond to any errors that occur during the process.

Q77. Write a express JS code to demonstrated to handle request of client through GET method.



```
const express = require('express');
const app = express();

// Define a route for the root path
app.get('/', (req, res) => {
  // Send an HTML response
  res.send('<h1>Welcome to my Express server!</h1>');
});

// Define a route with a parameter
app.get('/users/:userId', (req, res) => {
  // Access the userId parameter from the URL
  const userId = req.params.userId;
  // Send a response with the userId
  res.send(`User ID is: ${userId}`);
});

// Start the server
const PORT = 2300;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

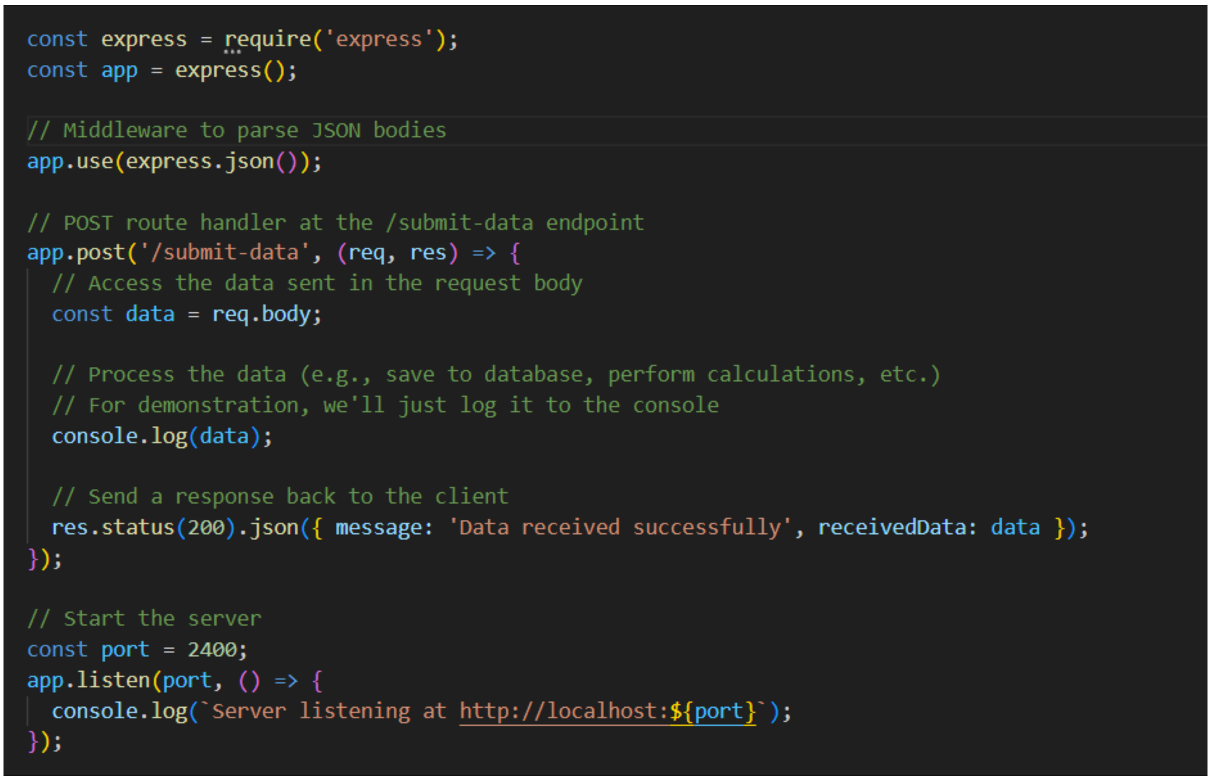


Welcome to my Express server!

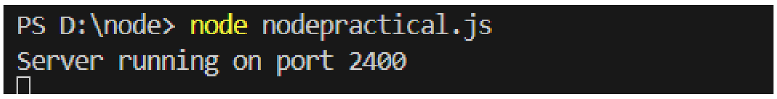
In this code:

- The server listens for GET requests at the root path (/) and sends back an HTML message.
- It also listens for GET requests at /users/:userId and responds with the user ID provided in the URL.

Q78. Write a express JS code to demonstrated to handle request of client through POST method.



```
const express = require('express');
const app = express();

// Middleware to parse JSON bodies
app.use(express.json());

// POST route handler at the /submit-data endpoint
app.post('/submit-data', (req, res) => {
  // Access the data sent in the request body
  const data = req.body;

  // Process the data (e.g., save to database, perform calculations, etc.)
  // For demonstration, we'll just log it to the console
  console.log(data);

  // Send a response back to the client
  res.status(200).json({ message: 'Data received successfully', receivedData: data });
});

// Start the server
const port = 2400;
app.listen(port, () => {
  console.log(`Server listening at http://localhost:${port}`);
});
```



```
PS D:\node> node nodepractical.js
Server running on port 2400
```

To test this server,

you can use a tool like curl or Postman to send a POST request with JSON data to http://localhost:2400/submit-data.

Here's an example curl command to test the server:

curl -X POST -H "Content-Type: application/json" -d '{"key":"value"}' http://localhost:2400/submit-data

This command sends a POST request with a JSON body containing { "key": "value" } to the server, which will log the data to the console and respond with a confirmation message.

Q79. What is routing. Explain various routing approach through example.

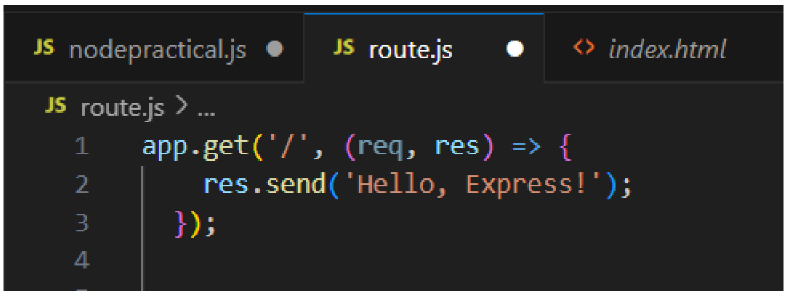
Routing in Express.js refers to how an application’s endpoints (URIs) respond to client requests. It defines how different URLs are handled by the server, allowing you to map specific routes to specific actions or handlers. In other words, routing determines how your application responds to different HTTP methods (such as GET, POST, PUT, DELETE) and specific URL paths.

Routing Approaches:

Basic Route:

- Define a route using app.get(), app.post(), etc.
- Specify the path and a callback function that handles the request.

Example:

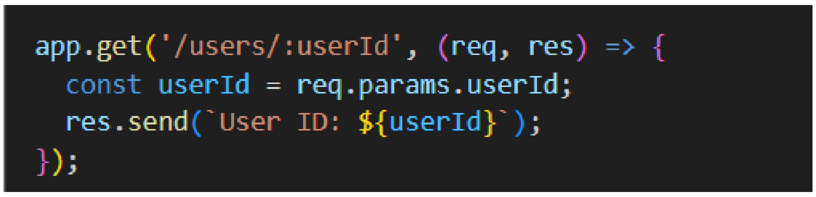


```javascript
app.get('/', (req, res) => {
    res.send('Hello, Express!');
  });

```

Route Parameters:

- Use route parameters to capture dynamic values from the URL.
- Access parameters using req.params.

Example:

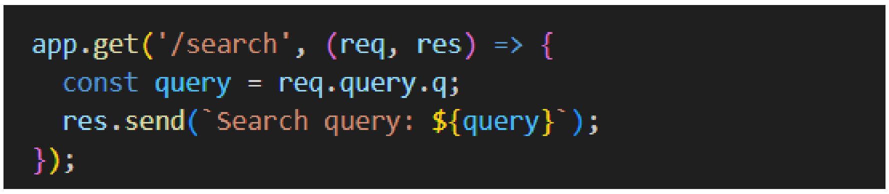


```javascript
app.get('/users/:userId', (req, res) => {
  const userId = req.params.userId;
  res.send(`User ID: ${userId}`);
});
```

Query Strings:

- Extract query parameters from the URL using req.query.

Example:

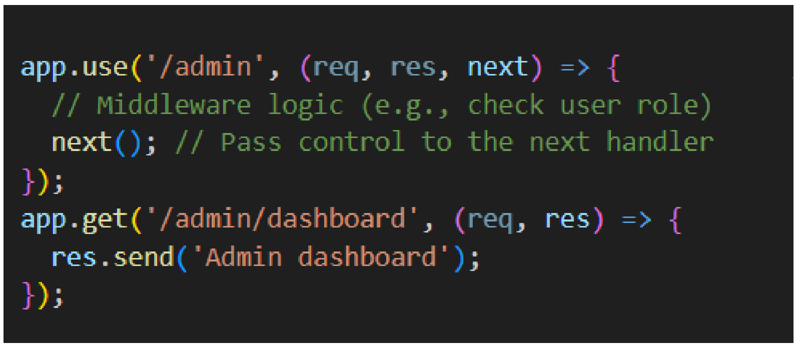


```javascript
app.get('/search', (req, res) => {
  const query = req.query.q;
  res.send(`Search query: ${query}`);
});
```

Middleware Routes:

- Use middleware functions to handle specific routes.
- Middleware can perform tasks like authentication, logging, etc.

Example:

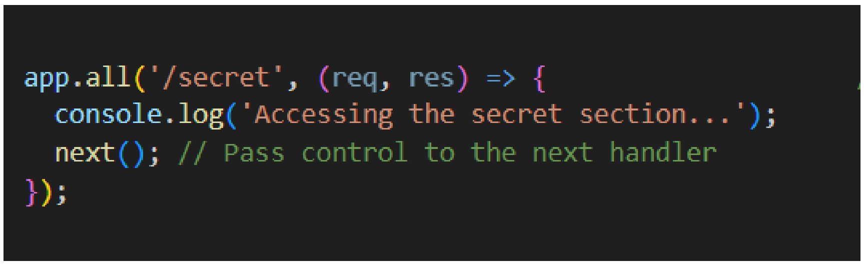


```
app.use('/admin', (req, res, next) => {
  // Middleware logic (e.g., check user role)
  next(); // Pass control to the next handler
});
app.get('/admin/dashboard', (req, res) => {
  res.send('Admin dashboard');
});
```

Wildcard Routes:

- Handle multiple routes with a common prefix using app.all().

Example:

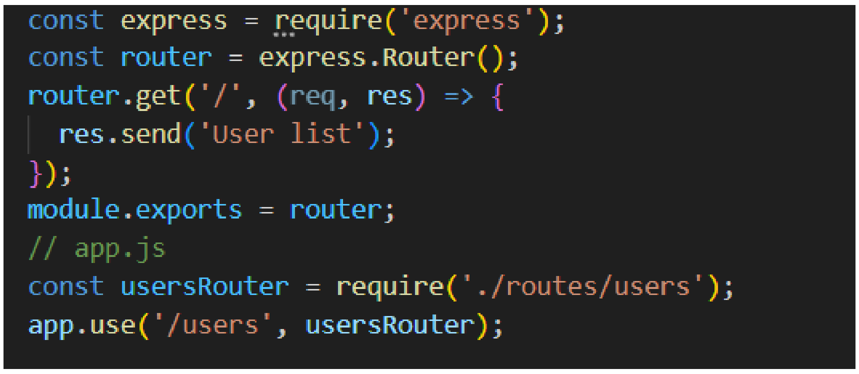


```
app.all('/secret', (req, res) => {
  console.log('Accessing the secret section...');
  next(); // Pass control to the next handler
});
```

Route Groups (Modular Routes):

- Organize routes into separate files (controllers).
- Use express.Router() to create modular route handlers.

Example:



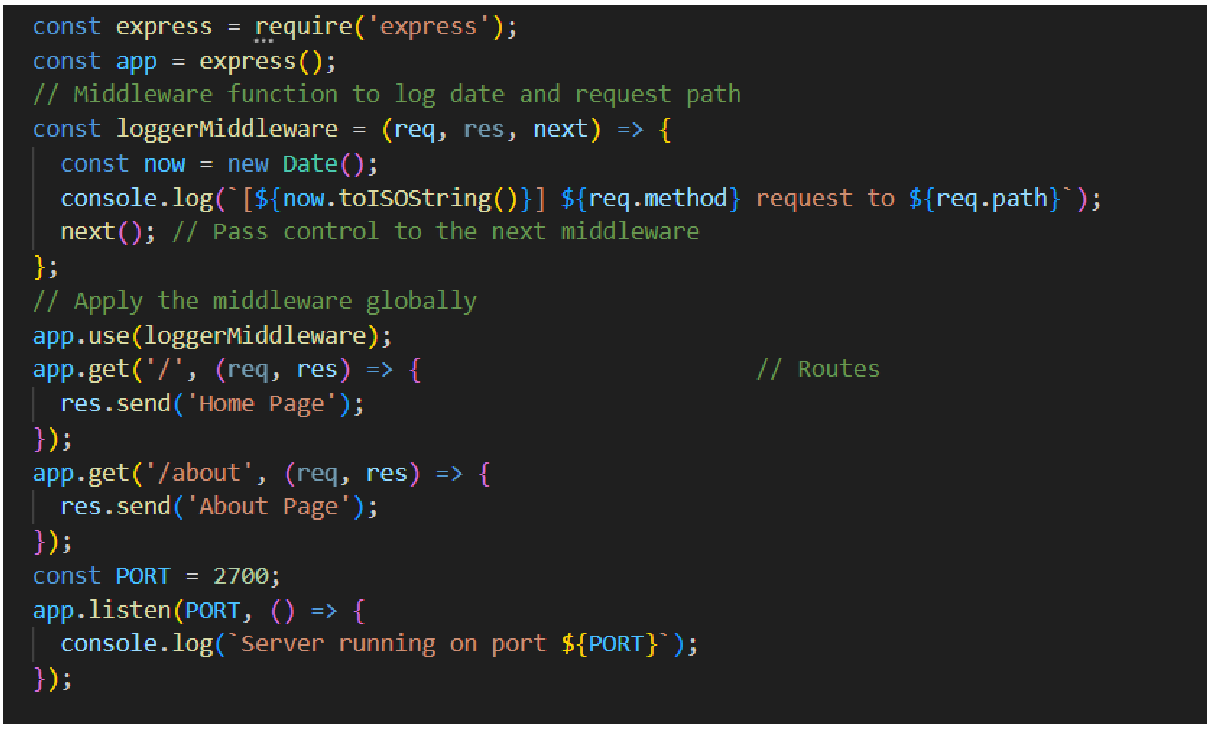
```
const express = require('express');
const router = express.Router();
router.get('/', (req, res) => {
  res.send('User list');
});
module.exports = router;
// app.js
const usersRouter = require('./routes/users');
app.use('/users', usersRouter);
```

Q80. What is express middleware. Explain the process of implementation of middleware through example.

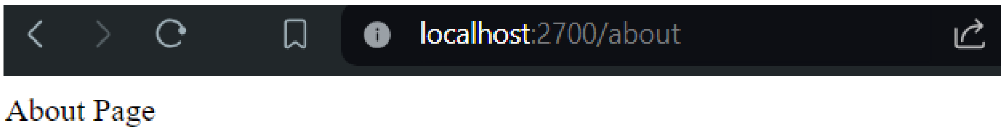
Express middleware are functions that have access to the request object (req), the response object (res), and the next middleware function in the application’s request-response cycle. These functions can perform a variety of tasks such as executing code, making changes to the request and response objects, ending the request-response cycle, or calling the next middleware in the stack.

Implementation of middleware in an Express.js application:

- **Define Middleware Function:** Create a function that takes three arguments: req, res, and next.
- **Use Middleware:** Apply the middleware to your application using app.use(). You can apply it globally or to specific routes.
- **Next Function**: Call next() within your middleware to pass control to the next middleware function. If not called, the request will be left hanging.



```
const express = require('express');
const app = express();
// Middleware function to log date and request path
const loggerMiddleware = (req, res, next) => {
  const now = new Date();
  console.log(`[${now.toISOString()}] ${req.method} request to ${req.path}`);
  next(); // Pass control to the next middleware
};
// Apply the middleware globally
app.use(loggerMiddleware);
app.get('/', (req, res) => {                          // Routes
  res.send('Home Page');
});
app.get('/about', (req, res) => {
  res.send('About Page');
});
const PORT = 2700;
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```



localhost:2700/about

About Page

In this code, every time a request is made to any route, the loggerMiddleware will log the current date and the request path before the route handler is executed.

Q81. What is express.js template. Explain component of express js template and justify each component through steps and example.

Express.js allow you to dynamically generate HTML pages by combining static template files with data. They are essential for creating reusable components and rendering dynamic content.

Components of Express.js Templates:

Template Engine:

- A template engine is a library or module that enables you to use static template files in your application.
- It replaces variables in the template with actual values and transforms the template into an HTML file sent to the client.
- Popular template engines for Express include Pug (formerly Jade), EJS (Embedded JavaScript), Handlebars, and Mustache.

Views Directory:

- The views directory is where you store your template files.
- By default, Express looks for views in this directory.
- Set the views directory using app.set('views', './views').

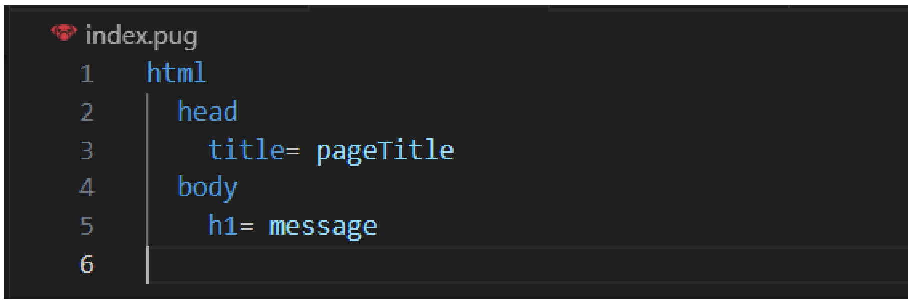
View Engine:

- The view engine specifies which template engine to use.
- For example, to use Pug, set app.set('view engine', 'pug').
- Express loads the specified template engine internally.

Template Files:

- Template files contain the HTML structure with placeholders for dynamic content.
- These placeholders are typically variables or expressions.

Example(Pug template):



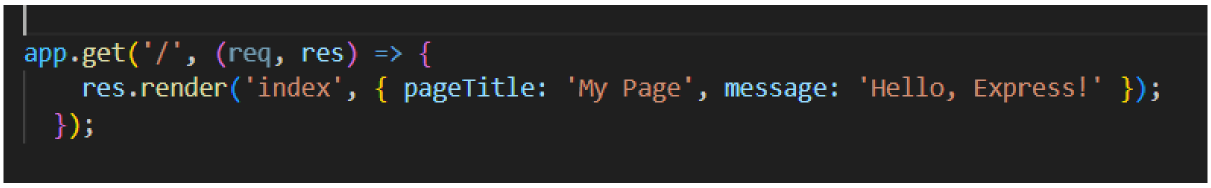
index.pug

```pug
html
  head
    title= pageTitle
  body
    h1= message

```

Route Handlers:

- In your route handlers, use res.render() to render a template.
- Pass data (variables) to the template using an object.

Example:



```javascript
app.get('/', (req, res) => {
    res.render('index', { pageTitle: 'My Page', message: 'Hello, Express!' });
  });
```

Q82. Write a express JS code to demonstrated to different routing approach.

In Express.js, there are several ways to define routes for handling different HTTP methods and endpoints.

1. **Basic Route Handling:** You can define routes using methods of the Express app object that correspond to HTTP methods.

```
const express = require('express');
const app = express();

// Respond with "hello world" when a GET request is made to the homepage
app.get('/', (req, res) => {
    res.send('hello world');
});

// GET method route
app.get('/about', (req, res) => {
    res.send('This is the about page');
});

// POST method route
app.post('/contact', (req, res) => {
    res.send('Contact form submitted');
});

// Other HTTP methods (PUT, DELETE, etc.) can be handled similarly
// ...
```

2. **Route Paths:** Route paths define the endpoints at which requests can be made. They can be strings, string patterns, or regular expressions.

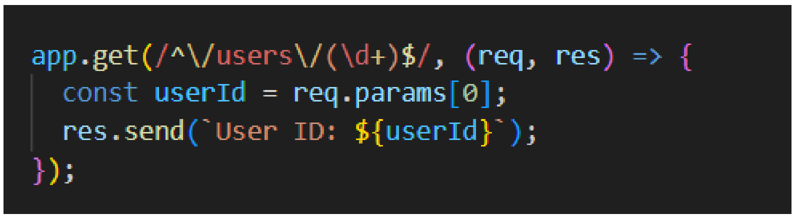
- **String-based path:**

```
app.get('/products', (req, res) => {
  res.send('List of products');
});
```

- **Parameterized path:**

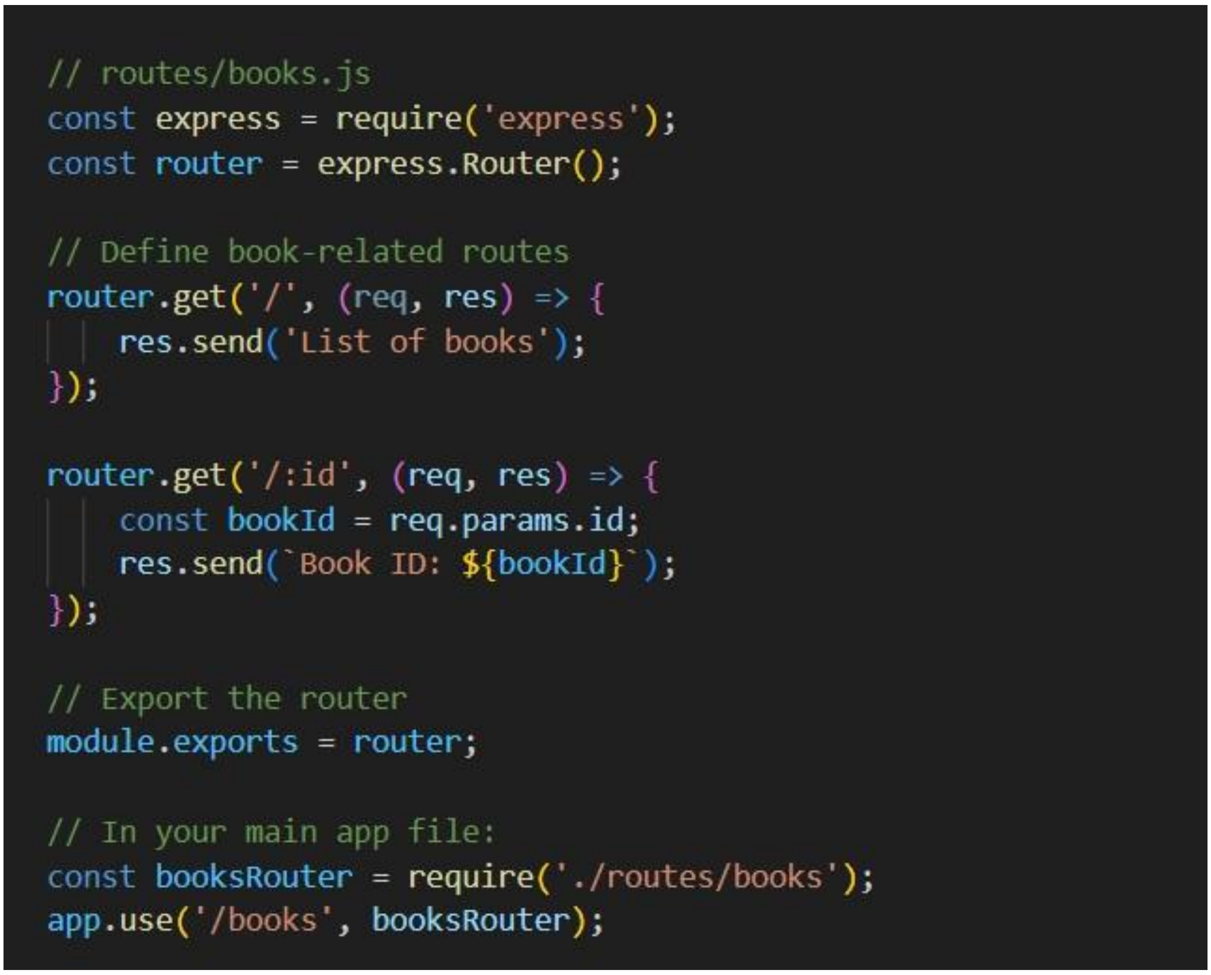
```
app.get('/products/:id', (req, res) => {
  const productId = req.params.id;
  res.send(`Product ID: ${productId}`);
});
```

- **Rgular expression-based path:**



```
app.get(/^\/users\/(\d+)$/, (req, res) => {
  const userId = req.params[0];
  res.send(`User ID: ${userId}`);
});
```

3. **Express Router Middleware:** You can use express.Router to group route handlers together and access them using a common route prefix. This helps organize your routes into modular components:

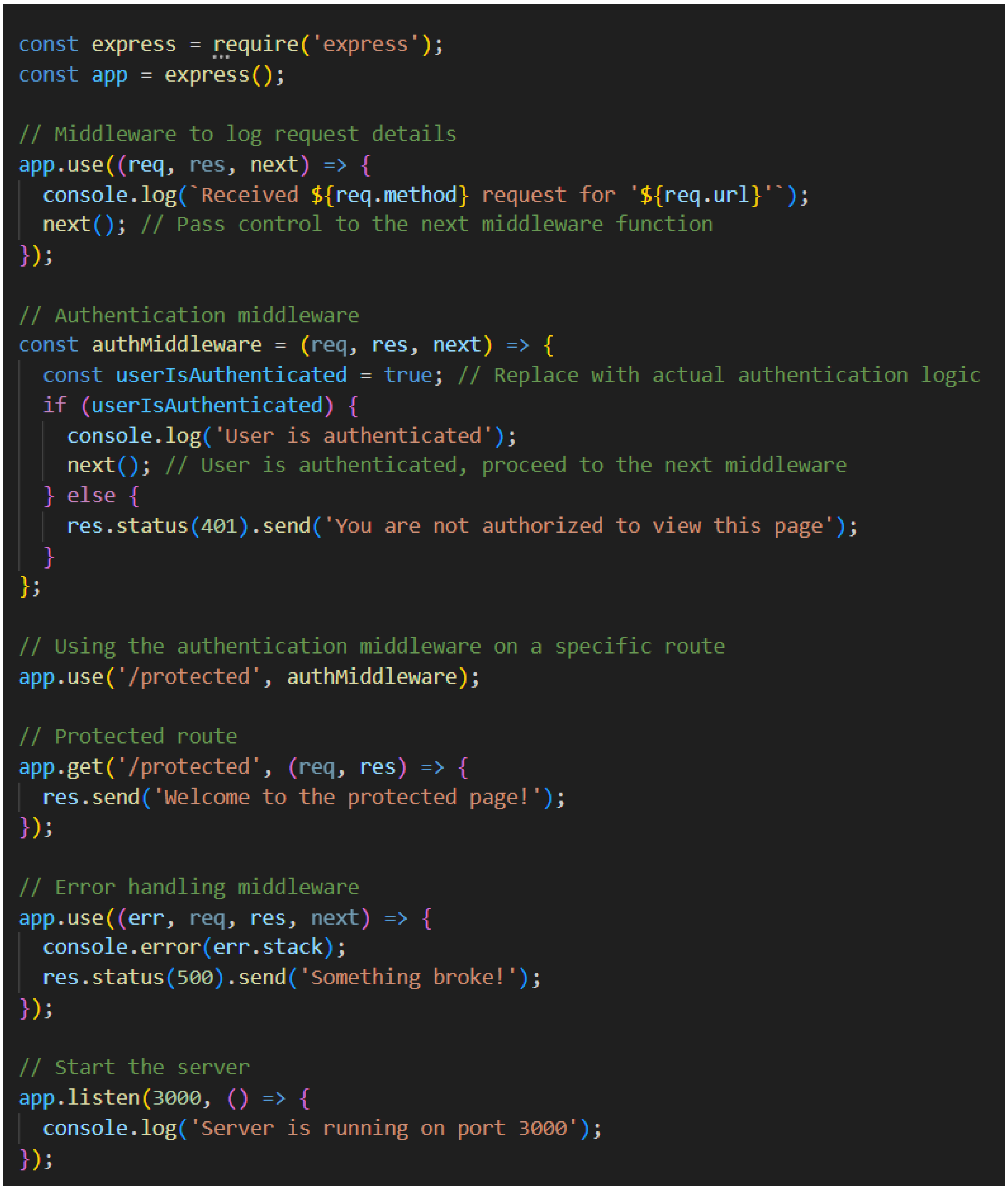


```
// routes/books.js
const express = require('express');
const router = express.Router();

// Define book-related routes
router.get('/', (req, res) => {
    res.send('List of books');
});

router.get('/:id', (req, res) => {
    const bookId = req.params.id;
    res.send(`Book ID: ${bookId}`);
});

// Export the router
module.exports = router;

// In your main app file:
const booksRouter = require('./routes/books');
app.use('/books', booksRouter);
```

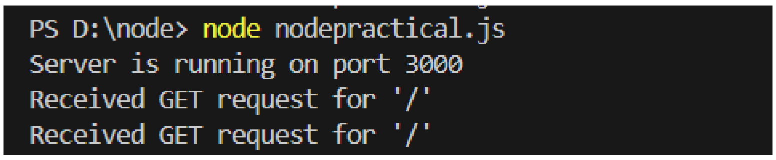
4. **Controller Functions:** Consider separating route handling logic from the controller functions that process requests. Controllers retrieve data from models, create HTML pages, and return them to users. This separation improves code organization and maintainability.

Q83. Write a express JS code to demonstrated to express middleware approach.

Middleware in Express.js is a powerful feature that allows you to run code before the final request handler is executed.



```javascript
const express = require('express');
const app = express();

// Middleware to log request details
app.use((req, res, next) => {
  console.log(`Received ${req.method} request for '${req.url}'`);
  next(); // Pass control to the next middleware function
});

// Authentication middleware
const authMiddleware = (req, res, next) => {
  const userIsAuthenticated = true; // Replace with actual authentication logic
  if (userIsAuthenticated) {
    console.log('User is authenticated');
    next(); // User is authenticated, proceed to the next middleware
  } else {
    res.status(401).send('You are not authorized to view this page');
  }
};

// Using the authentication middleware on a specific route
app.use('/protected', authMiddleware);

// Protected route
app.get('/protected', (req, res) => {
  res.send('Welcome to the protected page!');
});

// Error handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});

// Start the server
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

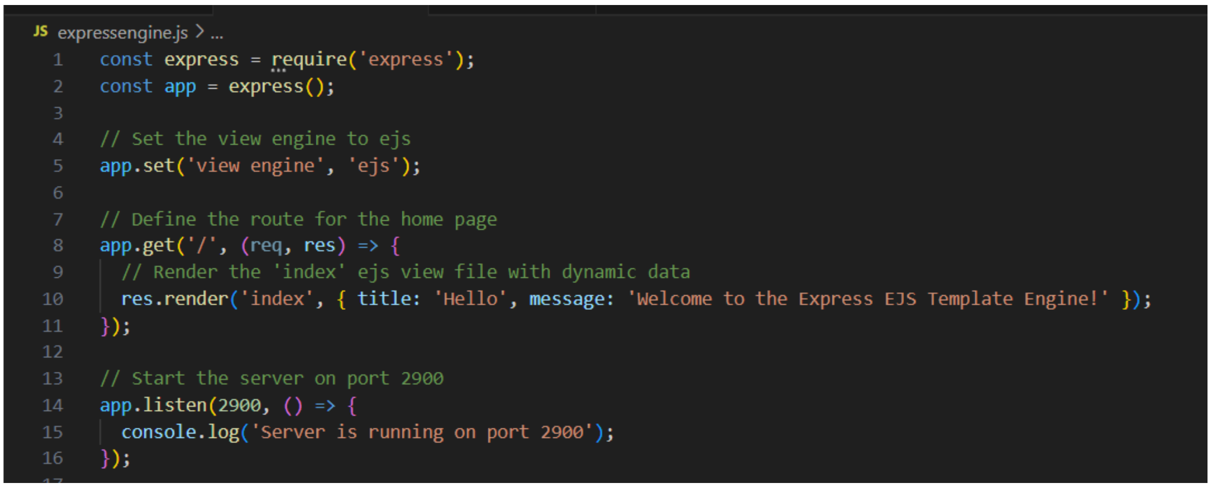


```
PS D:\node> node nodepractical.js
Server is running on port 3000
Received GET request for '/'
Received GET request for '/'
```

In this code:

- The first middleware logs the type of HTTP request and the requested URL.
- The authMiddleware function checks if a user is authenticated. If not, it sends a 401 Unauthorized response.
- The /protected route uses the authMiddleware to ensure only authenticated users can access it.
- The error handling middleware catches any errors that occur in the routing and logs them, then sends a 500 Internal Server Error response.

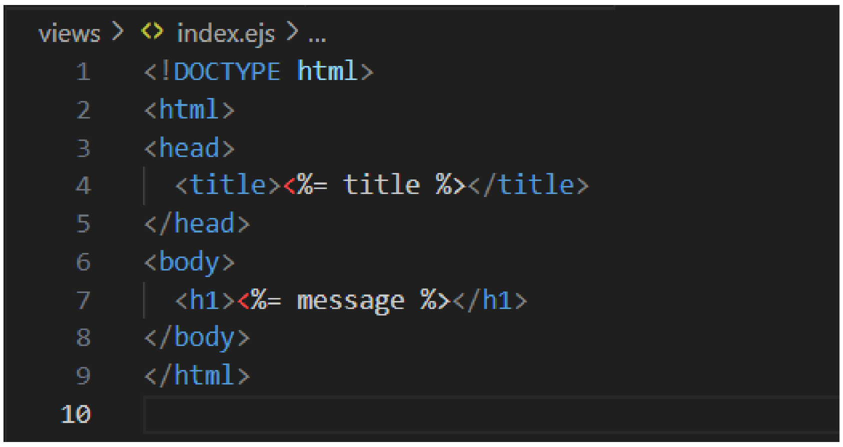
Q84. Write a express JS code to demonstrated to express template engine.



expressengine.js > ...

```javascript
const express = require('express');
const app = express();

// Set the view engine to ejs
app.set('view engine', 'ejs');

// Define the route for the home page
app.get('/', (req, res) => {
  // Render the 'index' ejs view file with dynamic data
  res.render('index', { title: 'Hello', message: 'Welcome to the Express EJS Template Engine!' });
});

// Start the server on port 2900
app.listen(2900, () => {
  console.log('Server is running on port 2900');
});
```

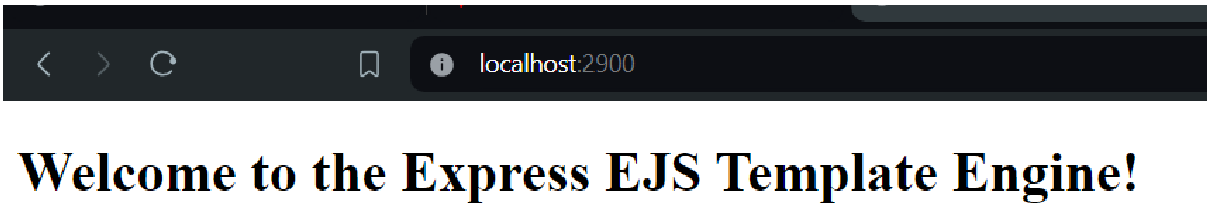
Make sure you have EJS installed in your project by running npm install ejs. Then, create a file named index.ejs in a directory called views with the following content:



views > index.ejs > ...

```html
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
</head>
<body>
  <h1><%= message %></h1>
</body>
</html>
```

This index.ejs file is a template that will be rendered by the server when you visit the home page ('/' route). The <%= title %> and <%= message %> are placeholders that get replaced with the actual values passed from the server when rendering the page.



localhost:2900

Welcome to the Express EJS Template Engine!

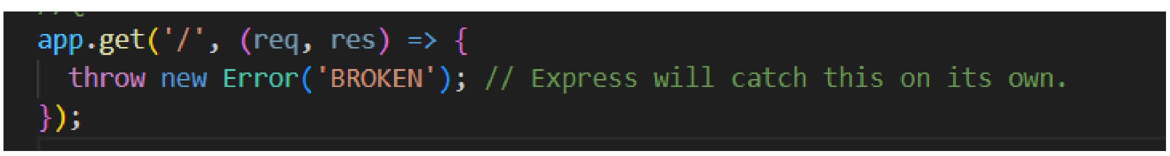
Q85. What is Error Handling. Explain various types of error in express JS.

Error handling in Express.js refers to how the framework catches and processes errors that occur both synchronously and asynchronously.

Synchronous Errors:

- These errors occur during the execution of synchronous code inside route handlers and middleware.
- If synchronous code throws an error, Express will catch and process it automatically.

Example:

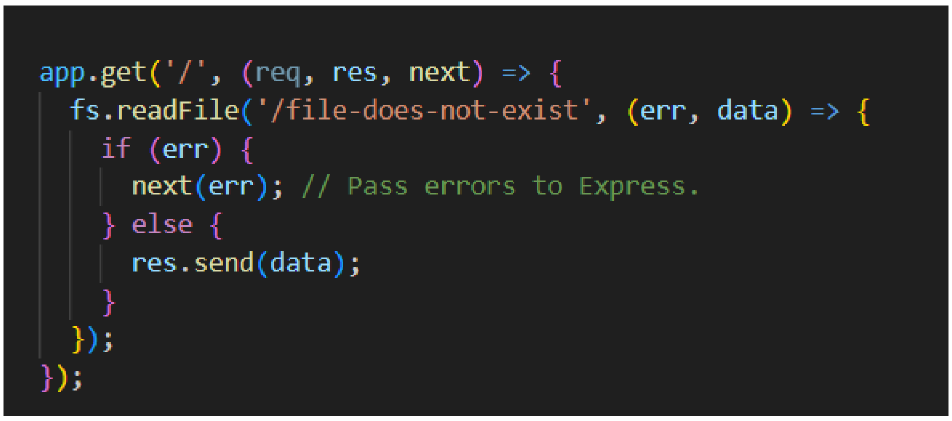


```
app.get('/', (req, res) => {
  throw new Error('BROKEN'); // Express will catch this on its own.
});
```

Asynchronous Errors:

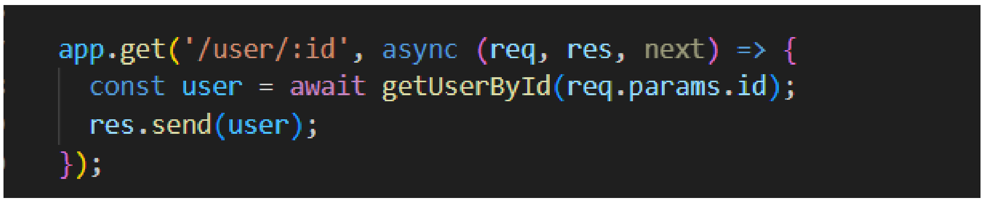
- These errors occur in asynchronous functions invoked by route handlers and middleware.
- You must pass these errors to the next() function, where Express will catch and process them.

Examples:



```
app.get('/', (req, res, next) => {
  fs.readFile('/file-does-not-exist', (err, data) => {
    if (err) {
      next(err); // Pass errors to Express.
    } else {
      res.send(data);
    }
  });
});
```

Starting with Express 5, route handlers and middleware that return a Promise will automatically call next(value) when they reject or throw an error. For instance:



```
app.get('/user/:id', async (req, res, next) => {
  const user = await getUserById(req.params.id);
  res.send(user);
});
```

If getUserById throws an error or rejects, next will be called with either the thrown error or the rejected value.

Logical Errors:

- These errors occur due to incorrect logic in the program.
- For example, applying the wrong logic gate or misunderstanding program requirements.
- Logical errors are harder to detect because they don't cause exceptions but lead to incorrect behavior.

Syntax Errors:

- Syntax errors occur when the code violates the language's syntax rules.
- These errors prevent successful compilation and execution.
- For example, incorrect indentation in Python can cause a syntax error.

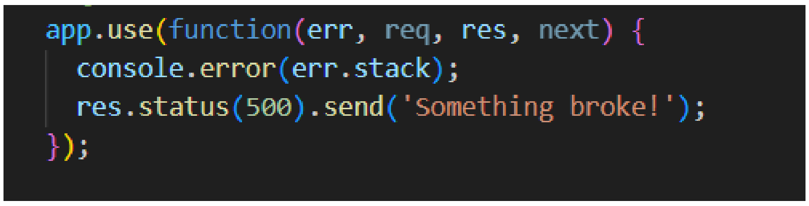
Runtime Errors:

- Runtime errors occur during program execution.
- They are detected only when the program runs or the command is executed.
- Examples include division by zero, null pointer dereference, or file not found.

Q86. Explain various approach of error handling in express JS.

Middleware Functions:

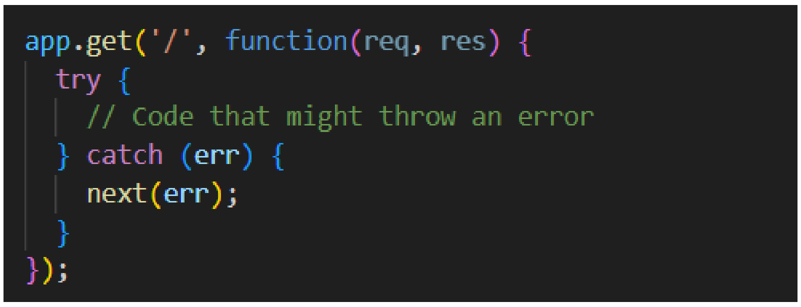
- Express.js has built-in support for error-handling middleware.
- These are special functions that have four arguments: (err, req, res, next).
- They should be defined after all other app.use() and route calls.



```
app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

Try-Catch Blocks:

- For synchronous code, you can use try-catch blocks to handle errors.
- This ensures that any errors that occur are caught and handled in a controlled manner.

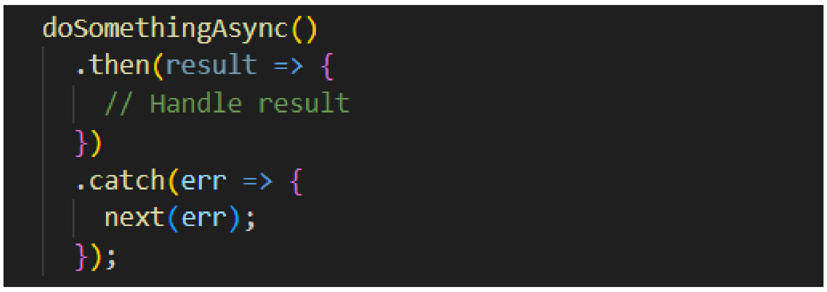


```
app.get('/', function(req, res) {
  try {
    // Code that might throw an error
  } catch (err) {
    next(err);
  }
});
```

Promise Rejection:

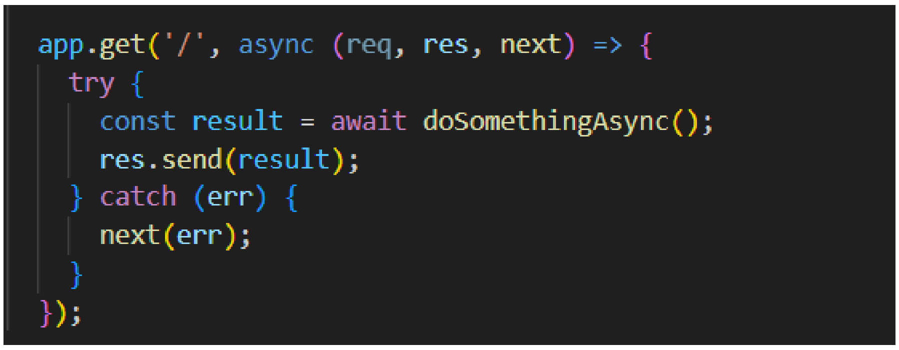
- In asynchronous code that uses promises, you can catch errors using .catch() or async/await with try-catch.

Example with .catch()



```
doSomethingAsync()
  .then(result => {
    // Handle result
  })
  .catch(err => {
    next(err);
  });
```

Example with async/await:



```
app.get('/', async (req, res, next) => {
  try {
    const result = await doSomethingAsync();
    res.send(result);
  } catch (err) {
    next(err);
  }
});
```

Third-Party Middleware:

- Packages like express-async-errors automatically wrap your route handlers in try-catch blocks.
- This simplifies error handling in asynchronous operations.

Custom Error Objects:

- You can create custom error objects to handle specific types of errors.
- This allows for more granular control over error handling and responses.

Error Propagation:

- You can propagate errors to centralized error-handling middleware by calling next(err).
- This allows you to handle errors in one place rather than scattering error-handling logic throughout your code.

Logging and Monitoring:

- It's important to log errors for debugging and monitoring purposes.
- You can use logging libraries to record error details, which can help in troubleshooting.

Q87. Explain various applications of template engine.

Template engines are powerful tools in web development, allowing developers to create dynamic, reusable, and maintainable web pages.

Dynamic Content Generation:

- Template engines enable the creation of web pages with content that can change based on user input, server-side variables, or other dynamic data sources.

Separation of Concerns:

- They help separate the presentation layer from the business logic, making it easier for designers and developers to work on their respective areas without interference.

Code Reusability:

- With template engines, you can create reusable components like headers, footers, and navigation bars that can be included across different pages, promoting DRY (Don't Repeat Yourself) principles.

Simplified Maintenance:

- Changes to the website's layout or design can be made in one place, and those changes will propagate to all pages that use the template, simplifying maintenance.

Rapid Development:

- Template engines can speed up the development process by allowing developers to use pre-built templates and focus on implementing features rather than writing repetitive HTML code.

Server-Side Rendering (SSR):

- They are often used to implement SSR, where the server generates the full HTML for a page in response to a user's request, leading to faster page loads and improved SEO.

Localization and Internationalization:

- Template engines can dynamically render content in different languages, making it easier to create multi-lingual websites
.

Email Templates:

- They are used to generate custom emails with dynamic content, such as personalized greetings or transaction details, for email marketing campaigns.

Configuration Files:

- Some template engines can be used to generate configuration files for different environments, such as development, testing, or production1.

Testing and Mockups:

- Developers can use template engines to quickly create mockups or prototypes by injecting mock data into templates, facilitating rapid testing and iteration.

Q88. What is process manager. Explain how to apply process manager in express JS.

A process manager is a tool that helps manage and keep Node.js applications, like those built with Express.js, running in the background. It can restart applications if they crash, help with load balancing, and improve performance through insights and settings adjustments.

To apply a process manager in an Express.js application, you can use one of the popular tools like **PM2, Forever,** or **Strong Loop Process Manager**.

How to Apply PM2:

Install PM2

Syntax: **npm install pm2 -g**

Start your application

Syntax: **pm2 start app.js**

Manage your application

- List all processes:

  Syntax: **pm2 list**

- Monitor in real-time:

  Syntax: **pm2 monit**

- Stop an application:

  Syntax: **pm2 stop app.js**

- Restart an application:

  Syntax: **pm2 restart app.js**

Q89. What is debugging. Explain how to apply debugging in express JS.

Debugging is the process of identifying and fixing errors or defects in software code. It involves analyzing the behavior of a program to understand why it is not functioning as expected and then making corrections to resolve the issues.

In Express.js, debugging helps developers identify problems in their application, track down issues, and improve code quality.

Apply debugging in Express.js:

Using the DEBUG Module:

- Express.js uses the debug module to provide internal logs.
- To see all the internal logs used in Express, set the DEBUG environment variable to express:* when launching your app.

Example:

```
DEBUG=express:* node index.js
```

- This command will display logs related to middleware functions, application mode, requests, and responses.
- You can customize the logs by specifying different namespaces.

Example:

To see logs only from the router implementation:

```
DEBUG=express:router node index.js
```

To see logs only from the application implementation:

```
DEBUG=express:application node index.js
```

To see logs from both router and application:

```
DEBUG=express:application,express:router node index.js
```

Debug Environment Variables:

- You can further customize the debugging experience using environment variables:

- DEBUG_COLORS: Set to 1 or 0 to control whether logs appear in different colors or plain white text.
- DEBUG_HIDE_DATE: Hide the date from debug output.
- DEBUG_SHOW_HIDDEN: Show hidden properties on inspected objects.

Windows Environment:

- For Windows, edit the package.json file and set your start command as follows:

```
"scripts": {
  "start": "set DEBUG=express:* & node
index.js"
}
```

- Run your app using npm start.

Debugging helps you find bugs, understand program behavior, and improve the reliability of your Express.js application. By leveraging the DEBUG module and customizing environment variables, you can efficiently troubleshoot issues during development and production.

Q90. Explain various component of security and deployment component of express JS.

Security Components in Express.js:

1. **Helmet Middleware:**

- Purpose: Enhances security by setting various HTTP headers.
- Usage:

```
const helmet = require('helmet');
app.use(helmet());
```

2. **Keep Dependencies Updated:**

- Regularly update dependencies to avoid vulnerabilities.
- Use tools like npm audit or Snyk to identify and update vulnerable packages.

3. **Rate Limiting:**

- Prevents brute-force attacks by limiting the number of requests per timeframe.
- Use the express-rate-limit middleware:

```
const rateLimit = require('express-rate-limit');
const limiter = rateLimit({ windowMs: 15 * 60 * 1000, max: 100 });
app.use(limiter);
```

4. **Input Sanitization:**

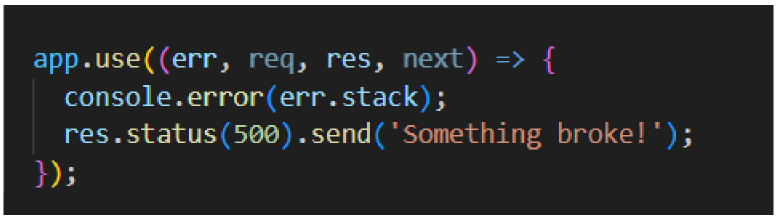
- Validate and sanitize user input to prevent injection attacks.
- Use libraries like express-validator.
-

5. **HTTPS (TLS):**

- Encrypt data in transit using HTTPS.
- Configure TLS certificates for secure communication.

6. **Custom Error Handling Middleware:**
   - Implement custom error handling to catch and manage errors.

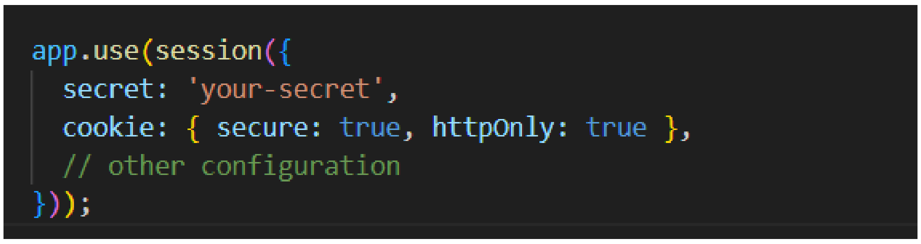
Example:



```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

7. **Secure Cookies:**
   - Use secure and HTTP-only flags for cookies.

Example:



```
app.use(session({
  secret: 'your-secret',
  cookie: { secure: true, httpOnly: true },
  // other configuration
}));
```

8. **Avoid Revealing Stack Traces:**
   - Prevent exposing sensitive information in error responses.

Deployment Components in Express.js:

1. **Choosing a Hosting Environment:**
   - Decide where to host your Express app (e.g., cloud providers, dedicated servers).
   - Consider Infrastructure as a Service (IaaS) or Container as a Service (CaaS).

2. **Prepare for Production:**
   - Make necessary changes to your project settings.
   - Set up a production-level infrastructure (hardware, OS, runtime, web server, databases).

3. **Containerization (Docker):**
   - Package your app, dependencies, and runtime in a container.
   - Use Docker to create and manage containers.

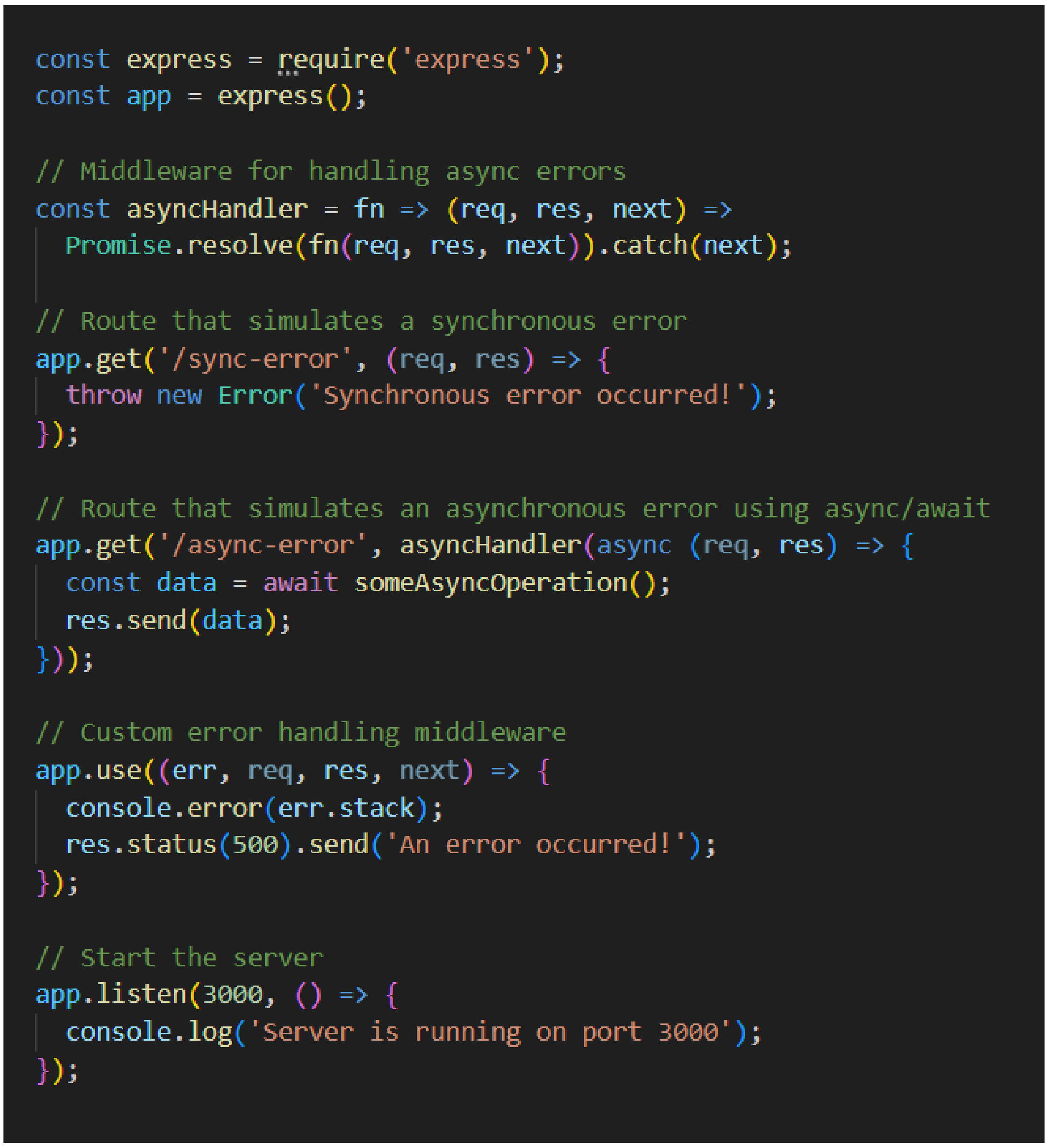
Example:

```
docker build -t my-express-app .

docker run -p 3000:3000 my-express-app
```

4. **Deploy to Cloud Services:**
   - Choose a cloud provider (e.g., AWS, Google Cloud, Heroku).
   - Deploy your containerized app to the cloud.

5. **Optimize Docker Images:**
   - Use multi-stage builds to reduce image size.
   - Separate build and production stages.

Q91. Write express JS code to demonstrated to how to handle different types of error.



```
const express = require('express');
const app = express();

// Middleware for handling async errors
const asyncHandler = fn => (req, res, next) =>
  Promise.resolve(fn(req, res, next)).catch(next);

// Route that simulates a synchronous error
app.get('/sync-error', (req, res) => {
  throw new Error('Synchronous error occurred!');
});

// Route that simulates an asynchronous error using async/await
app.get('/async-error', asyncHandler(async (req, res) => {
  const data = await someAsyncOperation();
  res.send(data);
}));

// Custom error handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('An error occurred!');
});

// Start the server
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

In this code:

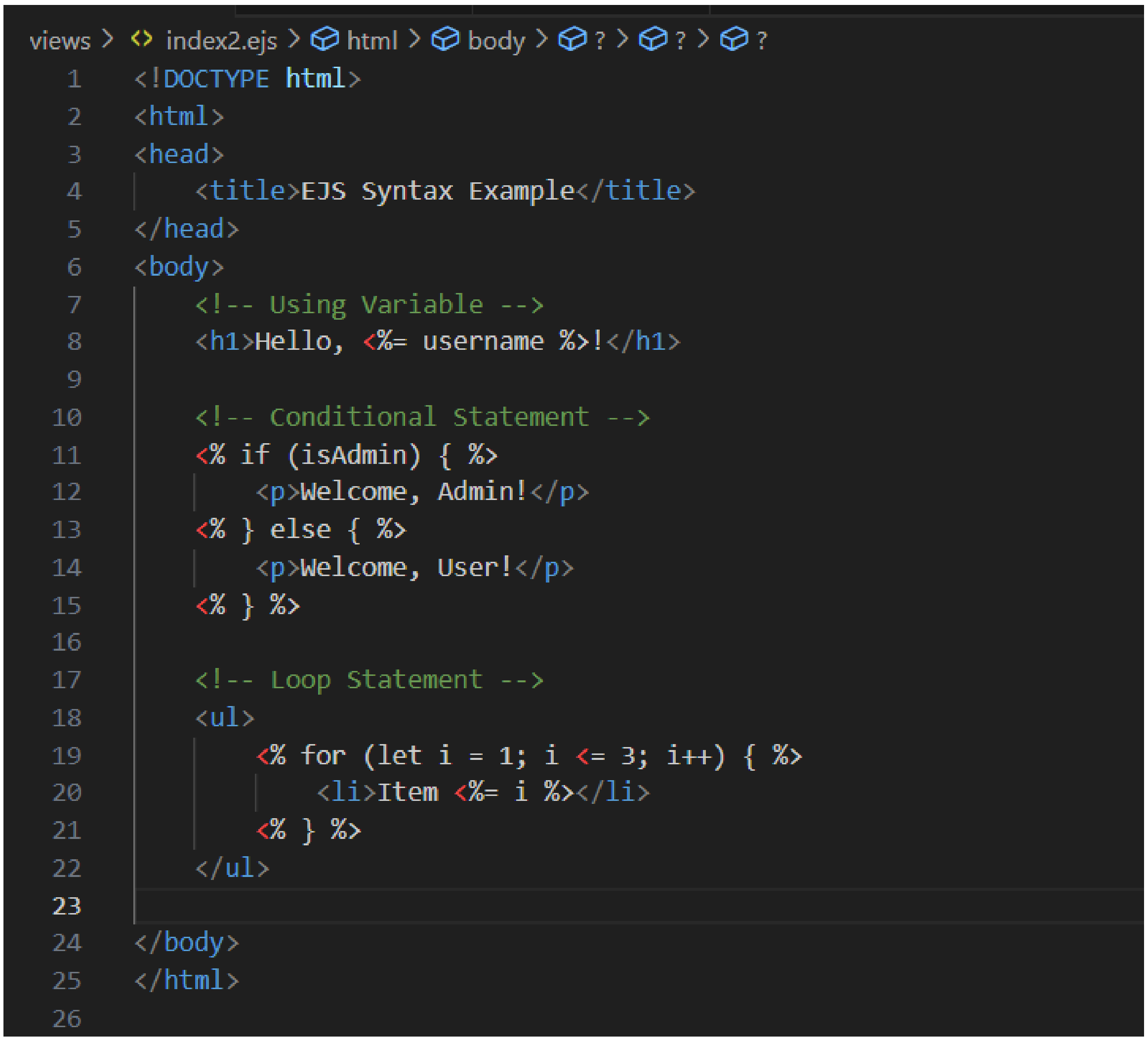
- The /sync-error route demonstrates how to handle synchronous errors. Express will automatically catch any errors thrown synchronously.
- The /async-error route uses an asyncHandler wrapper to handle asynchronous errors. If someAsyncOperation fails, the error will be caught and passed to the next error handling middleware.
- The custom error handling middleware at the end of the middleware stack catches any errors passed to next() and sends a generic error response to the client.

Q92. Write a express JS code to demonstrated to parametrized value accession through template engine.

First, make sure you have Express and EJS installed in your project. If not, install them using:

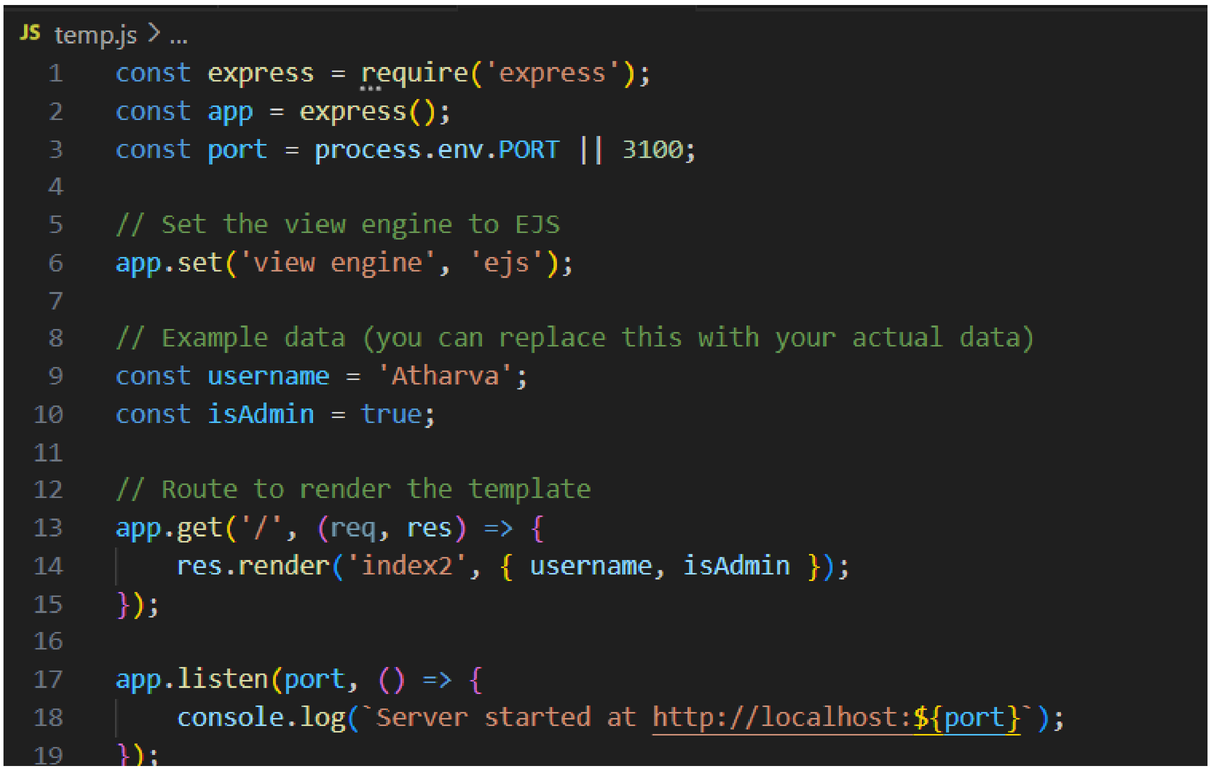
npm install express ejs

Create an index2.ejs file in your views directory (usually named views). This file will be our template.



```
<!DOCTYPE html>
<html>
<head>
    <title>EJS Syntax Example</title>
</head>
<body>
    <!-- Using Variable -->
    <h1>Hello, <%= username %>!</h1>

    <!-- Conditional Statement -->
    <% if (isAdmin) { %>
        <p>Welcome, Admin!</p>
    <% } else { %>
        <p>Welcome, User!</p>
    <% } %>

    <!-- Loop Statement -->
    <ul>
        <% for (let i = 1; i <= 3; i++) { %>
            <li>Item <%= i %></li>
        <% } %>
    </ul>

</body>
</html>
```

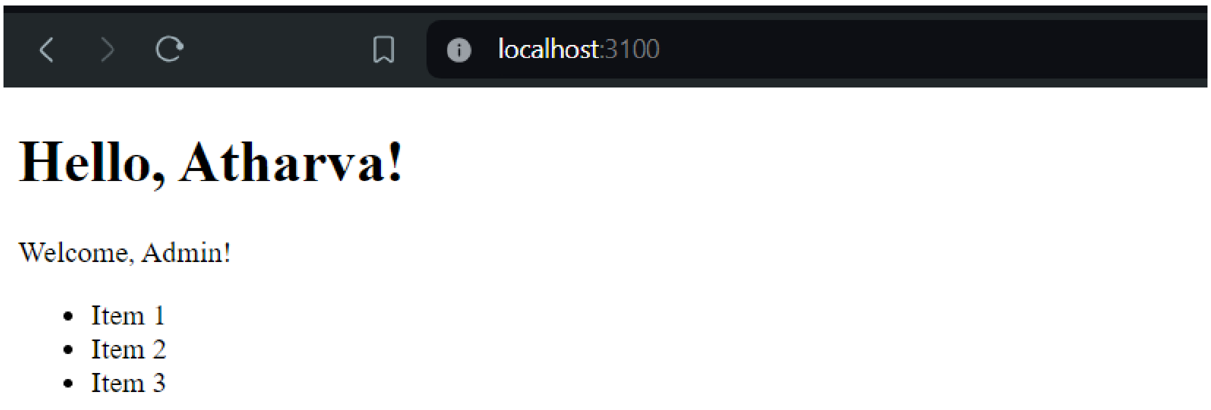
In your Express application (temp.js or similar), set up the view engine and render the index2.ejs template:



```
JS temp.js > ...
const express = require('express');
const app = express();
const port = process.env.PORT || 3100;

// Set the view engine to EJS
app.set('view engine', 'ejs');

// Example data (you can replace this with your actual data)
const username = 'Atharva';
const isAdmin = true;

// Route to render the template
app.get('/', (req, res) => {
    res.render('index2', { username, isAdmin });
});

app.listen(port, () => {
    console.log(`Server started at http://localhost:${port}`);
});
```

Run your Express server:

Syntax: **node temp.js**



```
localhost:3100
```

Hello, Atharva!

Welcome, Admin!

- Item 1
- Item 2
- Item 3

Visit http://localhost:3100 in your browser. You'll see the rendered template with dynamic content based on the provided data.