

FULL STACK DEVELOPMENT

Project Report Submitted In Partial Fulfillment Of The Requirements

Submitted By

Motamarri Jaya Naga Venkata Sai (208W1A12A0)

Under The Guidance Of

Dr . M . Ashok Kumar, Assistant Professor Ph.d

For The Award Of The Degree

BACHELOR OF TECHNOLOGY

IN

INFORMATION TECHNOLOGY



DEPARTMENT OF INFORMATION TECHNOLOGY

V R SIDDHARTHA ENGINEERING COLLEGE

(AUTONOMOUS – AFFILIATED TO JNTU – K, KAKINADA)

Approved by AICTE & Accredited by NBA

KANURU, VIJAYAWADA – 7

ACADEMIC YEAR

(2023 – 2024)

Table Of Contents

CHAPTER – 1 : Node JS	4
1.1 Introduction To Node :	4
1.2 Node Js Installation :	5
1.3 Working with Node Packages :	7
1.4 Writing Data To Console :	8
1.5 Events and Event Models :	10
1.6 Event Queues :	11
1.7 Call Back Implementations :	13
CHAPTER – 2 : DATA HANDLING	15
2.1 Working With JSON :	15
2.2 Buffer Model To Buffer Data :	17
2.3 Stream Module To Stream Data :	18
CHAPTER – 3 : FILE SYSTEM ACCESS	21
3.1 Synchronous and Asynchronous File Calls :	21
3.2 Writing and reading Into Files :	24
CHAPTER – 4 : HTTP SERVICES	29
4.1 Processing URL's :	29
4.2 Understanding Request :	30
4.3 Response and Server objects :	32
4.4 Implementing Http Clients and Servers in Node :	35
4.5 Implementing HTTPS Servers and Clients :	38
CHAPTER – 5 : EXPRESS JS	44
5.1 Implementing Express In Node Js :	44
5.2 Configuring Express Settings :	47
5.3 Starting Express Server :	49
5.4 Configuring Routes :	50
5.4 Using Request Objects :	52
5.5 Using Response Objects :	56
CHAPTER – 6 : ANGULAR	58
6.1 Understanding Angular :	58
6.2 Separation Of Responsibilities :	59
6.3 Adding Angular To The Environment :	60
6.4 Angular CLI :	62
6.5 Creating Basic Angular Application ;	63
6.6 Angular Components :	65
6.7 Angular Expressions :	67
6.8 Angular Bindings :	69

6.9 Directives – Structural and Attributes :	70
6.10 Events and Change Detection :	71
6.11 Using Observables :	73
CHAPTER – 7 : Mongo DB and Node JS	76
7.1 Adding Mongo DB Driver to Node JS :	76
7.2 Connecting To Mongo DB From Node JS :	77
7.3 Understanding The Objects :	78
7.4 Accessing and Manipulating The Database :	80
7.5 Introducing The DataSet :	81
7.6 Understanding The Query Objects and Options :	83
7.7 Finding Specific Sets Of Documents :	84
7.8 Counting Documents :	85
7.9 Limiting and Sorting Result Sets :	85
REFERENCES	87

CHAPTER – 1 : Node JS

1.1 Introduction To Node :

Node.js is an open-source, server-side JavaScript runtime environment that allows developers to run JavaScript code on the server rather than just in web browsers. It was created by Ryan Dahl in 2009 and has since gained widespread popularity in the development community. Node.js is built on the V8 JavaScript engine developed by Google for use in their Chrome web browser.

Here are some key points and features of Node.js :

1. **Non-blocking, Asynchronous I/O** : Node.js is designed to be non-blocking and event-driven, allowing for high concurrency with relatively low system resource consumption. This makes it suitable for handling many concurrent connections efficiently.
2. **JavaScript** : Node.js allows developers to use JavaScript for both client-side and server-side programming, creating a unified development stack. This enables full-stack developers to work with the same language across the entire application.
3. **Package Ecosystem** : Node.js has a vibrant and extensive ecosystem of open-source libraries and modules available through the Node Package Manager (NPM). NPM is a package manager that simplifies the process of adding, updating, and managing third-party packages.
4. **Fast Execution** : Node.js is known for its speed and efficiency, thanks to the V8 JavaScript engine. This makes it suitable for building real-time applications and scalable network applications.
5. **Event-Driven Architecture** : Node.js is event-driven, which means that it uses an event loop to handle requests and events. Developers can write code that responds to events, such as HTTP requests or file system operations, by attaching callback functions to these events.
6. **Cross-Platform** : Node.js is cross-platform and can run on various operating systems, including Windows, macOS, and Linux. This enables developers to write code that is not tied to a specific platform.
7. **Server-Side Applications** : Node.js is commonly used to build server-side applications, such as web servers, APIs, real-time chat applications, and more. It's especially well-suited for building applications that require real-time interaction or handling a large number of concurrent connections.

8. **Scalability** : Node.js is designed for easy scalability, making it a good choice for applications that may need to grow and handle increased loads.

Node.js has a strong and active community, which contributes to its growth and the development of a wide range of modules and tools. It has become a popular choice for building modern web and network applications due to its speed, simplicity, and versatility.

1.2 Node Js Installation :

1. Open the official website of the nodejs i.e; <https://nodejs.org/en/download> .
2. Go to that website and click on the windows 64 bit and download and run it.

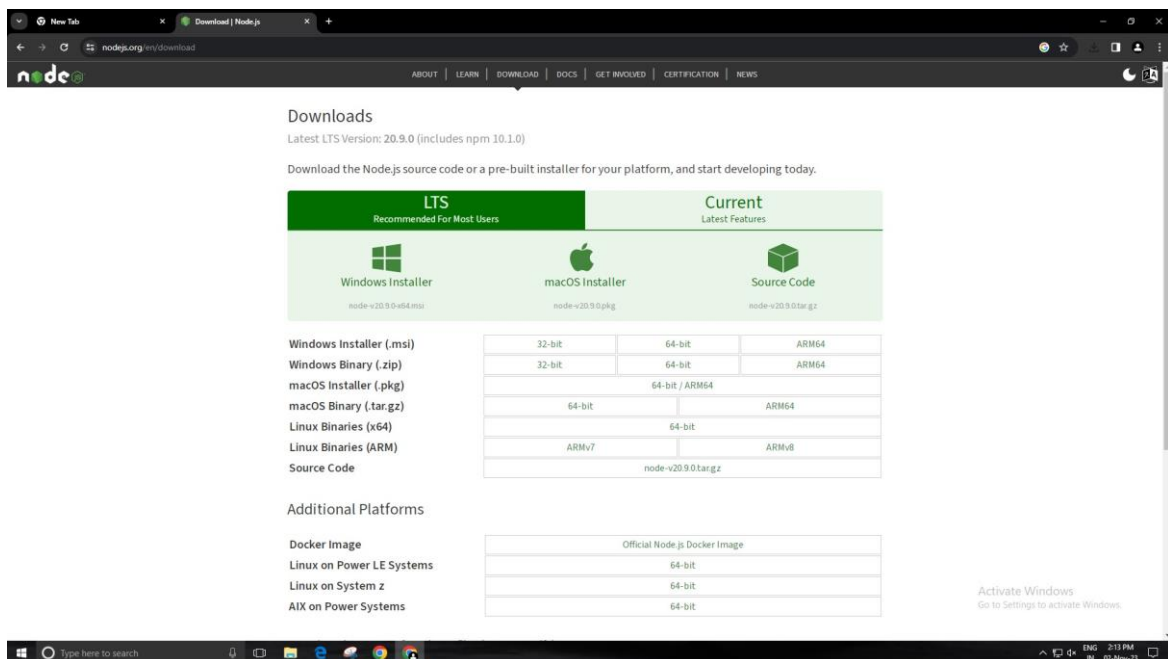


Fig – 1 : Node Download

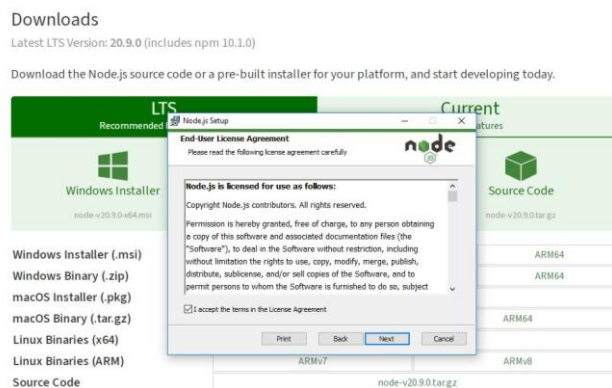


Fig – 2 : installation process

Click on the next button for all popups in the installation and last click on the finish button to install it.

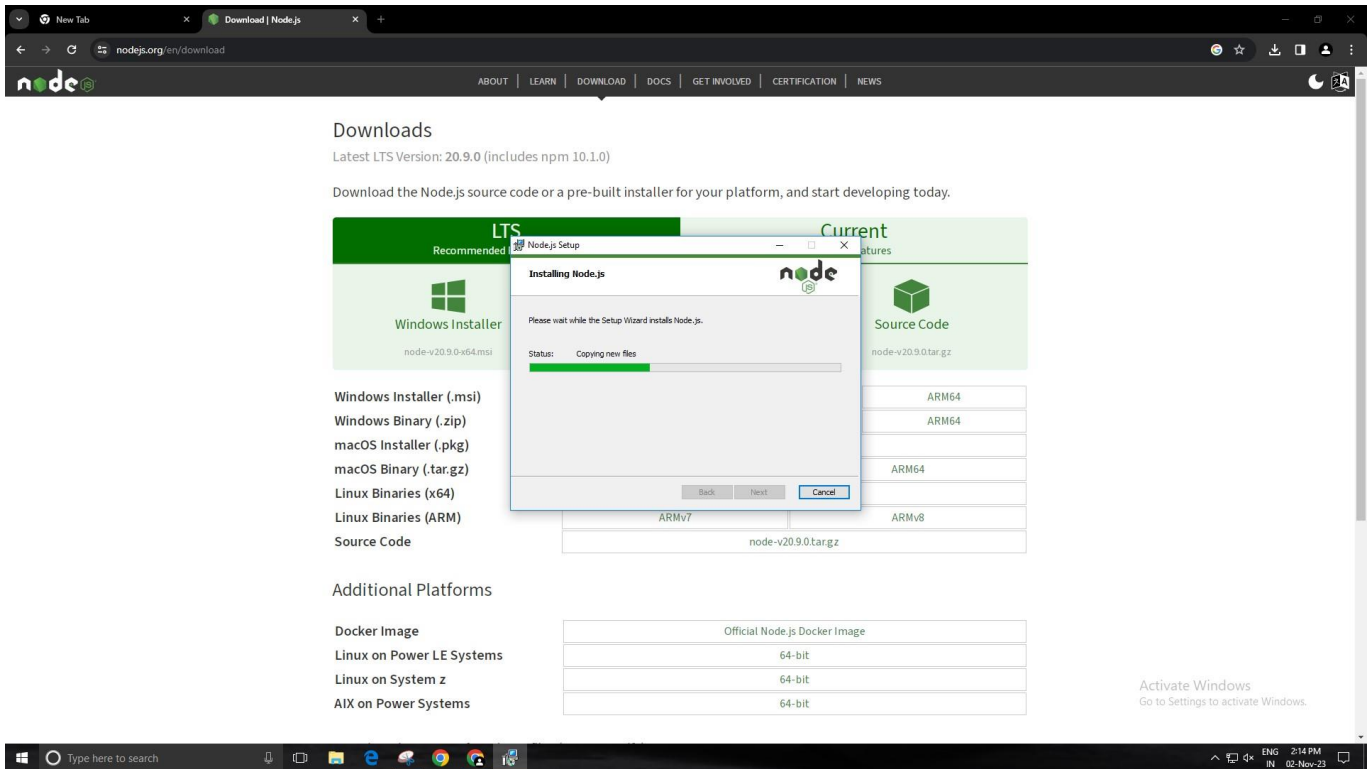


Fig – 3 : Node Installation

Now , Open the windows command prompt and type the command “ node --version “ this command will give the which version of nod you are using in the windows.

Type the same command to check the npm version in your windows the command is “ npm --version “

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\itadmin>node --version
v20.9.0

C:\Users\itadmin>npm --version
10.1.0

C:\Users\itadmin>
```

Fig – 4 : Node and Npm Version Check

1.3 Working with Node Packages :

Working with Node.js packages involves several key concepts and tools that are essential for developing and managing JavaScript applications. Here are the fundamental aspects of working with Node.js packages:

1. **Node Package Manager (NPM)** : NPM is the default package manager for Node.js. It is used to install, manage, and publish packages (also known as modules) for Node.js and JavaScript applications. NPM is included with Node.js, so you don't need to install it separately.
 - To check the installed version of NPM: `" npm -v "`
 - To update NPM to the latest version: `" npm install -g npm "`
2. **Package.json** : The package.json file is a manifest for your Node.js project. It contains metadata about the project and a list of dependencies. You can create a package.json file by running `npm init` and following the prompts.
3. **Installing Packages** : You can install packages using `npm install`. For example, to install a package named "express," you can run `npm install express`. The package will be downloaded and added to your project's `node_modules` directory. You can also specify the `--save` or `-S` flag to add the package to your dependencies in the package.json file.
4. **Global vs. Local Packages** : Some packages can be installed globally using the `-g` flag. These packages are available system-wide and can be used from the command line. Local packages are installed in your project directory.
 - Global installation: `" npm install -g package-name "`
 - Local installation: `" npm install package-name "`
5. **Dependencies** : Packages can have dependencies on other packages. These dependencies are listed in the dependencies section of the package.json file. To install all project dependencies listed in package.json, you can run `" npm install "`.
6. **Dev Dependencies** : Some packages are only needed during development, such as testing frameworks or build tools. These are listed in the dev Dependencies section of package.json. You can install them with `" npm install --save-dev package-name. "`

7. **Versioning and Semver** : Packages can specify version constraints in their dependencies. This is done using semantic versioning (semver). For example, "express": "^4.17.1" means that your project can use any version of Express greater than or equal to 4.17.1 but less than 5.0.0.
8. **Updating Packages** : To update packages to their latest versions, you can use " npm update " or update individual packages with " npm update package-name ". Be cautious with updates, as they may introduce breaking changes.
9. **Uninstalling Packages** : To remove a package from your project, you can use " npm uninstall package-name ". Use the --save or --save-dev flags to remove the package from your package.json file as well.
10. **Publishing Packages** : If you've developed your own Node.js package and want to share it with the community, you can publish it on the npm registry. Use " npm publish " to publish your package, but make sure to follow the npm guidelines and best practices.
11. **Lock Files** : NPM generates a package-lock.json or yarn.lock file to lock the dependency versions to ensure consistency across different environments. You should commit this file to your version control system.

These are the basics of working with Node.js packages and NPM. It's important to understand these concepts to effectively manage dependencies and develop Node.js applications.

1.4 Writing Data To Console :

Open the empty directory through the windows command prompt and then type " npm init -y " where -y means yes for all keys in the package.json file. In JavaScript and Node.js, you can write data to the console using the console object. The console object provides several methods for outputting data, such as text, variables, objects, and more. Here are some common ways to write data to the console:

The **console** object provides various other methods for writing different types of data:

- **console.error()**: Logs an error message.
- **console.warn()**: Logs a warning message.
- **console.info()**: Logs an informational message.
- **console.debug()**: Logs a message for debugging purposes (not available in all environments).
- **console.trace()**: Prints a stack trace of the current call.

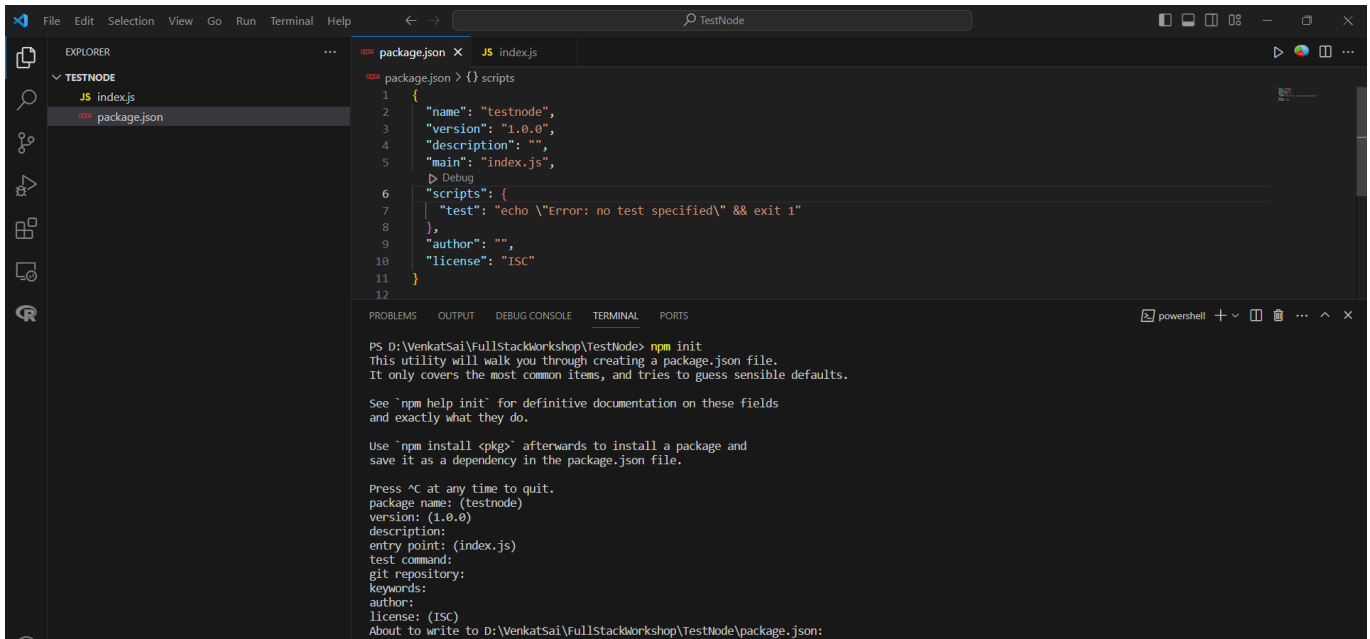


Fig – 5 : package json creation

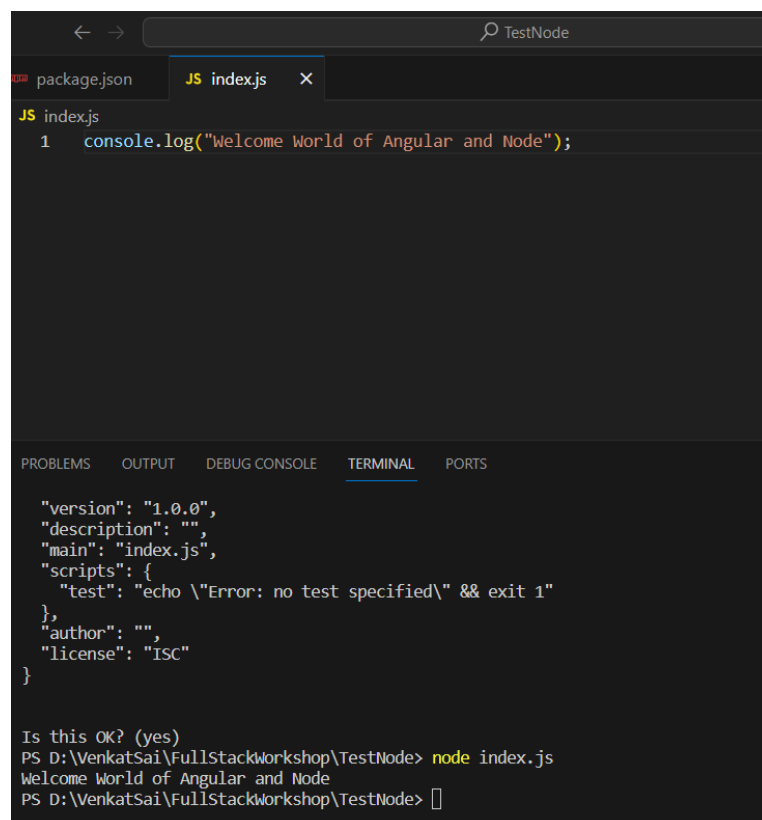


Fig – 6 : Writing data to console

1.5 Events and Event Models :

Events and event models are fundamental concepts in software development, particularly in the context of event-driven programming and user interfaces. Events represent interactions or occurrences, and event models describe how events are handled within a system. Here's an overview of events and event models:

Events : Events are occurrences or interactions that happen within a software system. They can include user actions (like clicking a button or pressing a key), system notifications, data changes, or other significant moments in the software's lifecycle. Events are crucial for creating responsive and interactive applications.

Common types of events include:

1. **User Interface (UI) Events :**

- Click events
- Mouse events (e.g., mouseover, mouseout)
- Keyboard events (e.g., keydown, keyup)
- Touch events (on touch-enabled devices)
- Form events (e.g., submit, change)

2. **Custom Events :** Developers can define and dispatch custom events to represent specific occurrences within their application.

3. **System Events :** Events generated by the operating system or underlying platform, such as network connectivity changes, file system events, or hardware events.

Event Models : Event models define how events are captured, processed, and responded to within a software system. They provide a structured way to handle events, ensuring that the application can respond to user interactions and system changes effectively. There are different event models depending on the technology or platform you're working with:

1. **Event Loop (JavaScript/Node.js) :** In JavaScript and Node.js, the event model is often based on an event loop. The event loop continuously checks for events and executes callback functions associated with those events. This allows for non-blocking, asynchronous event handling. JavaScript libraries like Node.js use this model for handling I/O operations, timers, and more.
2. **DOM Event Model (Web Browsers) :** In web development, the Document Object Model (DOM) event model is used to handle events associated with web pages. Web browsers provide APIs to interact with the DOM and attach event listeners to HTML elements. Developers can use JavaScript to respond to user actions like clicks or form submissions.

3. **Publisher-Subscriber Model** : The publisher-subscriber (pub-sub) model involves the creation of custom events and listeners. Components of a system can publish events, and other components subscribe to those events to respond appropriately. This model is often used in building modular and loosely coupled systems.
4. **Observer Pattern** : The observer pattern is a design pattern that allows an object (the subject) to maintain a list of its dependents (observers) and notify them of any state changes. This is commonly used in GUI frameworks, where UI elements can observe and react to changes in underlying data.
5. **Event-Driven Architecture** : In event-driven architecture, systems are designed around the concept of events. Different components communicate by emitting and listening for events, which promotes decoupling and scalability. This architecture is commonly used in distributed systems and microservices.

Understanding events and choosing the appropriate event model is crucial for building responsive, interactive, and maintainable software systems. The choice of event model may depend on the technology stack, the specific requirements of your application, and your development platform.

1.6 Event Queues :

Event queues are an essential part of event-driven programming and play a critical role in managing asynchronous operations and event handling within software systems. Event queues are data structures that hold events or tasks, and they are processed sequentially in the order they are added. Here's an overview of event queues:

Event Queue Basics :

1. An event queue is a mechanism for managing and processing events or tasks in an orderly fashion.
2. Events or tasks are added to the queue, typically in the order they occur or are scheduled.
3. Events in the queue are processed one at a time in a first-in, first-out (FIFO) order.
4. Event queues are commonly used in event-driven programming, multi-threading, and asynchronous programming to handle concurrency and manage non-blocking operations.

Types of Event Queues :

There are different types of event queues, depending on the programming environment :

1. **Message Queue** : Often used in operating systems, where processes or threads communicate through message-passing. A process or thread sends messages to a message queue, and other processes or threads can retrieve and process these messages.

2. **Task Queue** : Common in JavaScript and web development, a task queue (or job queue) manages asynchronous tasks. Functions are added to the task queue and executed in the order they were added once the main execution thread is idle.
3. **Event Loop Queue** : In the context of JavaScript and Node.js, the event loop queue is where events, callbacks, and promises are processed in a non-blocking and asynchronous manner.
4. **Scheduler Queue** : Used for scheduling tasks or events at specific times or with specific delays. It's commonly used in real-time systems or for managing time-sensitive operations.

Use Cases for Event Queues :

Event queues are used in various scenarios:

1. **User Interface (UI) Event Queue** : Handles user interactions like mouse clicks and keyboard events in graphical user interfaces. The UI framework (e.g., web browsers) processes these events in the order they occur.
2. **Asynchronous Operations** : When working with asynchronous code, event queues are used to manage callbacks, promises, and other asynchronous tasks to ensure non-blocking execution.
3. **Event-Driven Systems** : In event-driven architectures, components or modules communicate through event queues, allowing for loose coupling and scalability.
4. **Real-time Systems** : Event queues are essential in real-time systems, where events must be processed within strict time constraints.

Event Loop and Event Queue in JavaScript :

In JavaScript, the event loop is closely associated with the event queue. JavaScript is single-threaded and non-blocking, and it relies on the event loop to manage asynchronous operations. Tasks are added to the event queue, and the event loop continually checks for tasks to execute when the call stack is empty. Key components in the JavaScript event loop include the callback queue (used for asynchronous operations like `setTimeout`) and the microtask queue (used for promises and other microtasks). Understanding event queues is crucial for building responsive and efficient software,

especially in environments where asynchronous programming and event-driven architectures are prevalent. Properly managing event queues ensures that events and tasks are processed in a well-ordered and efficient manner.

1.7 Call Back Implementations :

A callback is a function that is passed as an argument to another function and is typically executed after the completion of that function. Callbacks are often used in programming to allow for asynchronous operations, event handling, and custom behavior in response to certain events or conditions. Here's a simple example to help illustrate callback implementations in JavaScript.

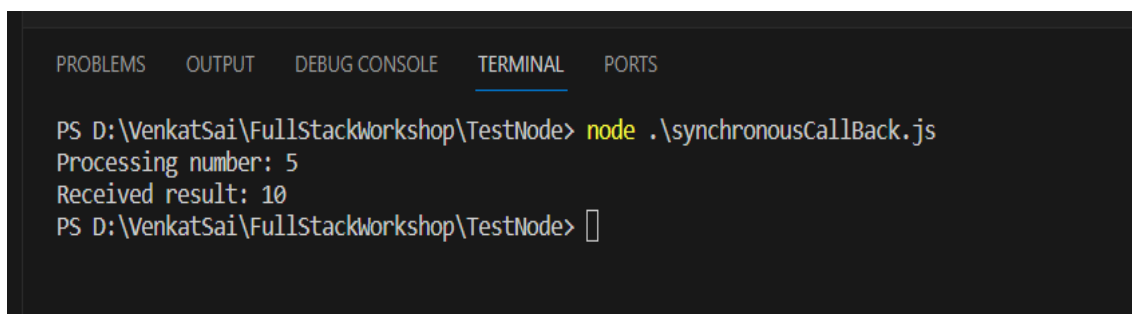
Synchronous Callback Example :

```
function processNumber(number, callback)
{
    console.log(`Processing number: ${number}`);
    callback(number * 2);
}

function handleResult(result)
{
    console.log(`Received result: ${result}`);
}

processNumber(5, handleResult);
```

Output :

A screenshot of a terminal window with a dark background. At the top, there are tabs labeled 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is active and underlined), and 'PORTS'. The terminal shows the command 'PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\synchronousCallback.js' followed by the output 'Processing number: 5' and 'Received result: 10'. The prompt 'PS D:\VenkatSai\FullStackWorkshop\TestNode>' is shown again at the end.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\synchronousCallback.js
Processing number: 5
Received result: 10
PS D:\VenkatSai\FullStackWorkshop\TestNode> 
```

Fig – 7 : Synchronous Output

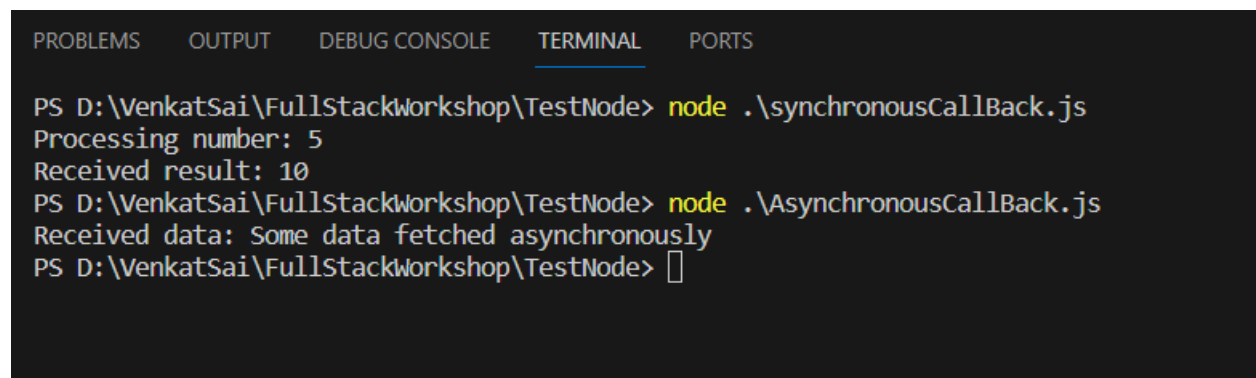
Asynchronous Callback Example :

```
function fetchData(callback)
{
    setTimeout(() => {
        const data = "Some data fetched asynchronously";
        callback(data);
    }, 2000);
}

function handleData(data)
{
    console.log(`Received data: ${data}`);
}

fetchData(handleData);
```

Output :



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\synchronousCallback.js
Processing number: 5
Received result: 10
PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\AsynchronousCallback.js
Received data: Some data fetched asynchronously
PS D:\VenkatSai\FullStackWorkshop\TestNode> █
```

Fig – 8 : Asynchronous Output

Callbacks are essential for handling asynchronous operations in JavaScript, such as making HTTP requests, reading files, and handling user input. They allow you to define custom behavior to be executed when an asynchronous task is completed, making your code more flexible and responsive.

CHAPTER – 2 : DATA HANDLING

2.1 Working With JSON :

Working with JSON (JavaScript Object Notation) in Node.js is quite straightforward because Node.js provides built-in support for JSON. JSON is a lightweight data-interchange format that is easy to read and write for both humans and machines.

Here are some common tasks you might perform when working with JSON in Node.js:

1. **Parsing JSON:** To parse a JSON string and convert it into a JavaScript object, you can use the `JSON.parse()` function.
2. **Stringifying JSON:** To convert a JavaScript object into a JSON string, you can use the `JSON.stringify()` function.
3. **Reading JSON from a File:** To read JSON data from a file in Node.js, you can use the `fs` module to read the file and then parse it as JSON. To install the “`fs` module” use “`npm install fs`” command.
4. **Writing JSON to a File:** To write JSON data to a file, you can use the `fs` module to create or overwrite a file with JSON data in string form.

Example :

```
// Parsing Json Data
const jsonString = '{"name": "John", "age": 30}';
const jsonObject = JSON.parse(jsonString);
console.log("Name of person is : ", jsonObject.name, "\n parsed object is : ", jsonObject);

// Stringfying The json
const jsonObject2 = { name: "John", age: 30 };
const jsonString2 = JSON.stringify(jsonObject);
console.log("Json Stringify : ", jsonString2);

// Reading Data From Json File
const fs = require('fs');
fs.readFile('data.json', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
}
```

```

const jsonObject3 = JSON.parse(data);

console.log("\n Json Data : ", jsonObject3);

});

// Writing Data In To Json

const fs2 = require('fs');

const jsonObject4 = { name: "John", language: "telugu", id : "1082", bio : "Human Being", version : 1.678 };

const jsonString4 = JSON.stringify(jsonObject4);

fs2.writeFile('data2.json', jsonString4, (err) => {

    if (err) {

        console.error(err);

    } else {

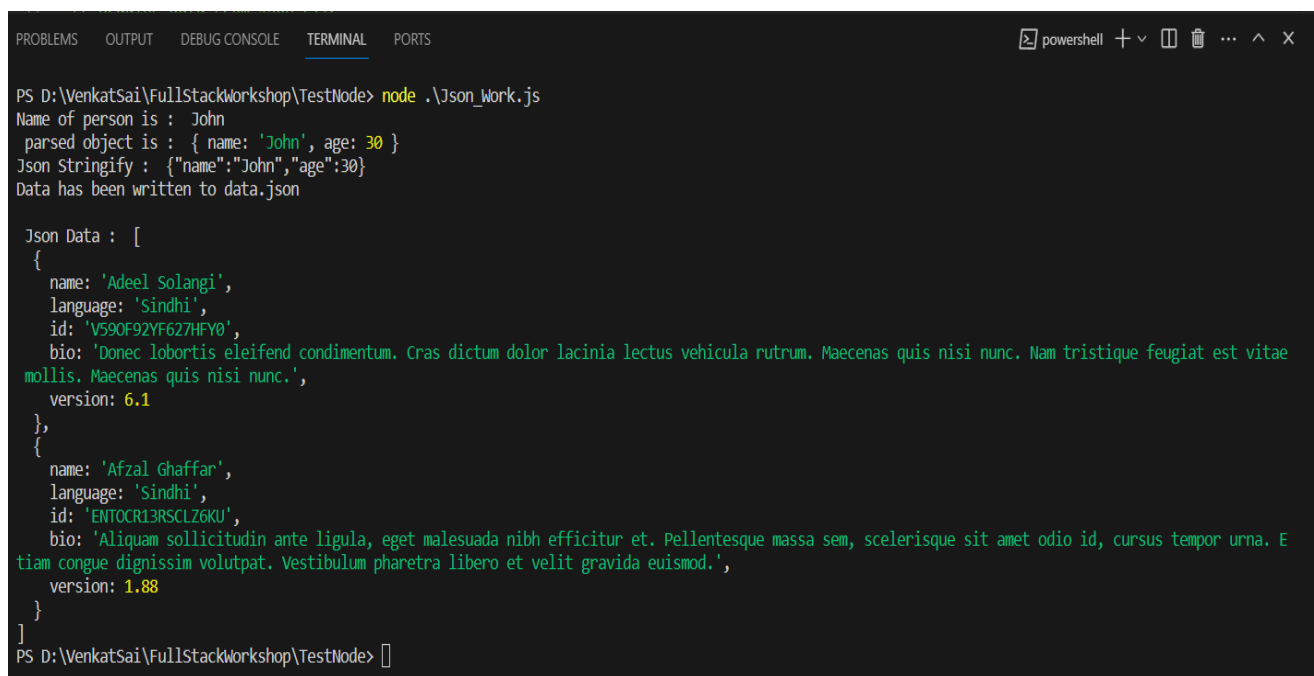
        console.log('Data has been written to data.json');

    }

});

```

Output :



```

PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\Json_Work.js
Name of person is : John
parsed object is : { name: 'John', age: 30 }
Json Stringify : {"name":"John","age":30}
Data has been written to data.json

Json Data : [
  {
    name: 'Adeel Solangi',
    language: 'Sindhi',
    id: 'V590F92YF627HFY0',
    bio: 'Donec lobortis eleifend condimentum. Cras dictum dolor lacinia lectus vehicula rutrum. Maecenas quis nisi nunc. Nam tristique feugiat est vitae mollis. Maecenas quis nisi nunc.',
    version: 6.1
  },
  {
    name: 'Afzal Ghaffar',
    language: 'Sindhi',
    id: 'ENTOCR13RSCLZ6KU',
    bio: 'Aliquam sollicitudin ante ligula, eget malesuada nibh efficitur et. Pellentesque massa sem, scelerisque sit amet odio id, cursus tempor urna. Etiam congue dignissim volutpat. Vestibulum pharetra libero et velit gravida euismod.',
    version: 1.88
  }
]
PS D:\VenkatSai\FullStackWorkshop\TestNode>

```

Fig – 9 : Json Ouput

2.2 Buffer Model To Buffer Data :

In Node.js, you can use buffers to work with binary data and buffer large chunks of data efficiently. Buffers in Node.js are essentially raw memory allocations and are particularly useful when dealing with binary data, file I/O, network operations, or when working with low-level data manipulation. Here's how you can create and work with buffers in Node.js:

1. **Creating a Buffer:** You can create a buffer in Node.js using the **Buffer** class. There are several ways to create a buffer:

- a. Allocate an empty buffer with a specific size :

```
const emptyBuffer = Buffer.alloc(10); // Creates a buffer of 10 bytes
```

- b. Create a buffer from an array of integers:

```
const bufferFromArray = Buffer.from([1, 2, 3, 4, 5]);
```

- c. Convert a string to a buffer:

```
const stringBuffer = Buffer.from('Hello, World!', 'utf8'); // Encoding is optional
```

2. **Reading and Writing to Buffers:**

You can read and write data to buffers using the `.readUInt8()`, `.readUInt16LE()`, `.readUInt16BE()`, `.writeUInt8()`, `.writeUInt16LE()`, and `.writeUInt16BE()` methods, among others. Example :

```
const buffer = Buffer.alloc(4);
buffer.writeUInt8(42, 0); // Write the value 42 at offset 0
const value = buffer.readUInt8(0); // Read the value at offset 0
console.log(value); // Output: 42
```

3. **Slicing Buffers:** You can create a new buffer that is a slice of an existing buffer using the `.slice()` method.

```
const originalBuffer = Buffer.from([1, 2, 3, 4, 5]);
const slicedBuffer = originalBuffer.slice(1, 4); // Creates a new buffer containing [2, 3, 4]
```

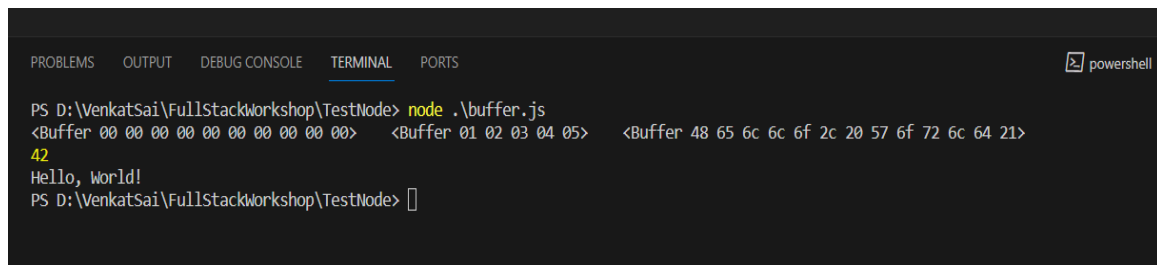
4. **Concatenating Buffers:** To concatenate multiple buffers, you can use the `Buffer.concat()` method.

```
const buffer1 = Buffer.from([1, 2, 3]);
const buffer2 = Buffer.from([4, 5, 6]);
const concatenatedBuffer = Buffer.concat([buffer1, buffer2]); // Results in a buffer [1, 2, 3, 4, 5, 6]
```

5. **Converting Buffers to Strings:** You can convert a buffer to a string using the `.toString()` method. Specify the character encoding when converting.

```
const buffer = Buffer.from('Hello, World!', 'utf8');
```

```
const str = buffer.toString('utf8');  
console.log(str); // Output: Hello, World!
```



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell  
PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\buffer.js  
<Buffer 00 00 00 00 00 00 00 00 00> <Buffer 01 02 03 04 05> <Buffer 48 65 6c 6c 6f 20 57 6f 72 6c 64 21>  
42  
Hello, World!  
PS D:\VenkatSai\FullStackWorkshop\TestNode> []
```

Fig – 10 : Buffer Output

Buffers are often used when working with streams, handling binary data, or when you need to efficiently manage large amounts of data in Node.js. Keep in mind that Node.js is gradually moving towards using TypedArray objects, which are more in line with modern JavaScript practices, especially for working with binary data. However, buffers are still widely used and supported.

2.3 Stream Module To Stream Data :

In Node.js, you can stream data using the built-in stream module. Streams provide a powerful and efficient way to work with data, especially when dealing with large datasets. There are several types of streams available in Node.js, including Readable, Writable, and Transform streams. Below, I'll provide an overview of how to use these stream types to stream data in Node.js.

1. **Readable Streams:** Readable streams are used to read data from a source, such as a file, an HTTP request, or any other source of data. You can create a Readable stream by using the fs module for file reading, or by extending the Readable class.
2. **Writable Streams:** Writable streams are used to write data to a destination, such as a file or an HTTP response. You can create a Writable stream by using the fs module for file writing, or by extending the Writable class.
3. **Transform Streams:** Transform streams are a type of duplex (both readable and writable) stream that allows you to modify or transform data as it passes through the stream. You can create a Transform stream by extending the Transform class.

Example For Streams :

```
import { createReadStream, createWriteStream } from 'fs';  
  
import { Transform } from 'stream';
```

```
// Readable Streams
```

```
const readableStream = createReadStream('example.txt');
```

```
readableStream.on('data', (chunk) => {
```

```
  console.log("Chuck Data : ", chunk, "\n");
```

```
});
```

```
readableStream.on('end', () => {
```

```
  console.log('Reading completed.');
```

```
});
```

```
// Writable Streams
```

```
const writableStream = createWriteStream('output.txt');
```

```
writableStream.write('Writable streams are used to write data to a destination, such as a file or an  
HTTP response. \n');
```

```
writableStream.end();
```

```
writableStream.on('finish', () => {
```

```
  console.log('Writing completed.');
```

```
});
```

```
// Transform Stream
```

```
const upperCaseTransform = new Transform({
```

```
  transform(chunk, encoding, callback) {
```

```
    const upperCaseText = chunk.toString().toUpperCase();
```

```
    this.push(upperCaseText);
```

```
    callback();
```

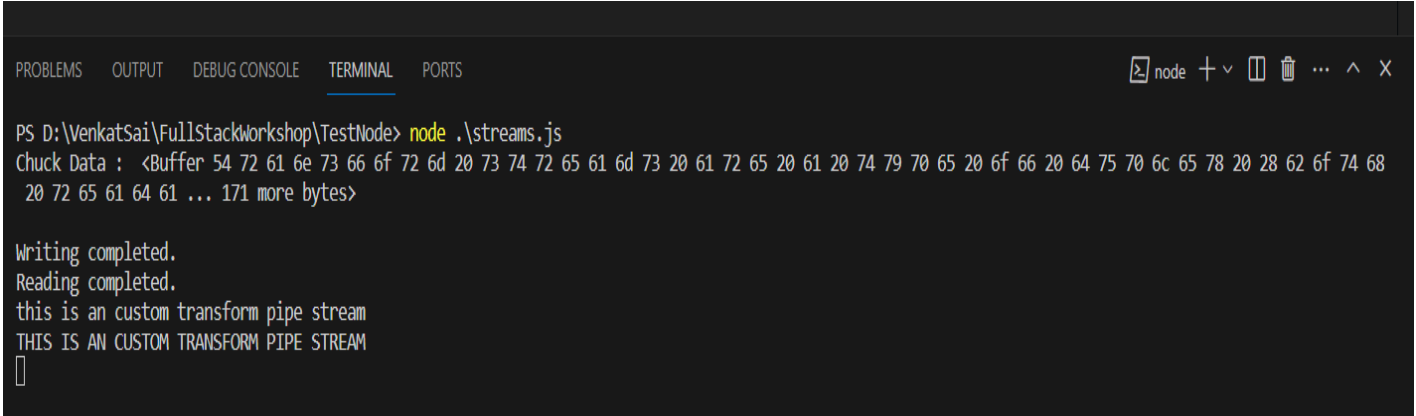
```
  }
```

```
});
```

```
// Pipe the standard input (stdin) to the uppercase transform stream
```

```
process.stdin.pipe(upperCaseTransform).pipe(process.stdout);
```

Output :



```
PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\streams.js
Chuck Data : <Buffer 54 72 61 6e 73 66 6f 72 6d 20 73 74 72 65 61 6d 73 20 61 72 65 20 61 20 74 79 70 65 20 6f 66 20 64 75 70 6c 65 78 20 28 62 6f 74 68
20 72 65 61 64 61 ... 171 more bytes>

Writing completed.
Reading completed.
this is an custom transform pipe stream
THIS IS AN CUSTOM TRANSFORM PIPE STREAM
█
```

Fig – 11 : Streams Output

These are the basic concepts of using the stream module in Node.js. You can combine these streams to build complex data processing pipelines. Streams are especially useful for handling large files and streaming data over the network because they allow you to process data in smaller, more manageable chunks rather than loading everything into memory at once.

CHAPTER – 3 : FILE SYSTEM ACCESS

3.1 Synchronous and Asynchronous File Calls :

Synchronous System File Call :

Synchronous file system calls in Node.js are a way to perform file I/O operations that block the execution of your program until the operation is completed. These methods are simple to use but can have some drawbacks, particularly in Node.js, where the event-driven, non-blocking nature of the platform is a core feature for building scalable and responsive applications.

Here's an explanation of synchronous file system calls in Node.js:

1. **Blocking Nature:** When you use synchronous file system methods, your program will wait for the file operation to complete before continuing with other tasks. This means that if you have multiple synchronous file system calls, they will execute one after the other, effectively blocking the event loop and making your application less responsive.
2. **Method Naming:** Synchronous file system methods in Node.js typically have the same names as their asynchronous counterparts, with "Sync" appended to the method name. For example, the synchronous version of `fs.readFile` is `fs.readFileSync`, and the synchronous version of `fs.writeFile` is `fs.writeFileSync`.
3. **Blocking Behavior:** Synchronous file system calls can block your application's event loop, making it less responsive to other incoming requests or tasks. In high-concurrency scenarios, this can lead to poor performance.
4. **Use Cases:** Synchronous file system calls are not recommended for I/O-bound operations in applications with high levels of concurrency, such as web servers. They may be suitable for simple scripts or command-line tools where the blocking behavior won't negatively impact the user experience.

Example :

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

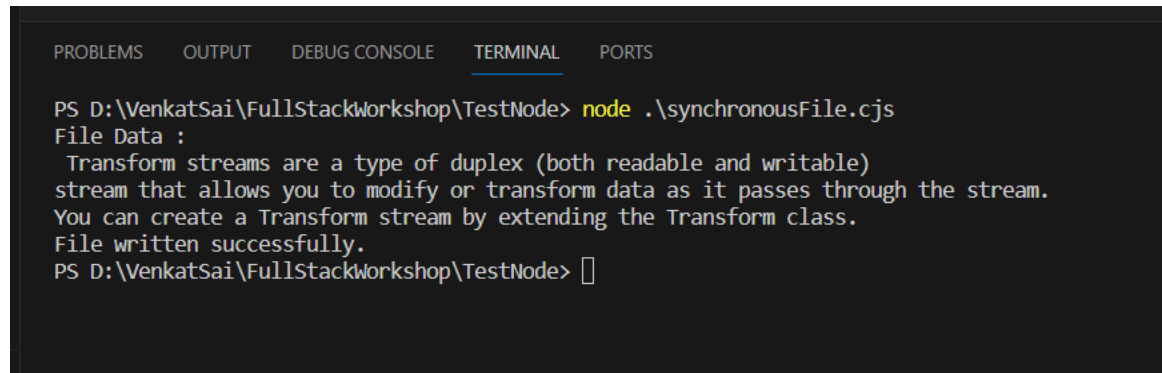
// Reading a file synchronously
try {
  const data = fs.readFileSync('example.txt', 'utf8');
  console.log("File Data : \n", data);
} catch (err) {
  console.error('Error:', err);
}

// Writing to a file synchronously
try {
```

```
fs.writeFileSync('newfile.txt', 'Hello, world!', 'utf8');  
  
console.log('File written successfully.');
```

```
} catch (err) {  
  
    console.error('Error:', err);  
  
}
```

Output :



```
PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\synchronousFile.cjs  
File Data :  
  Transform streams are a type of duplex (both readable and writable)  
  stream that allows you to modify or transform data as it passes through the stream.  
  You can create a Transform stream by extending the Transform class.  
  File written successfully.  
PS D:\VenkatSai\FullStackWorkshop\TestNode> 
```

Fig – 12 : Synchronous File

While synchronous file system calls have their place in certain situations, it's generally advisable to use asynchronous methods when developing Node.js applications. Asynchronous methods allow your application to remain responsive and handle multiple I/O operations concurrently without blocking the event loop. This is especially important in applications that need to handle multiple clients or requests simultaneously.

Asynchronous System File Call :

Asynchronous file system calls in Node.js are a way to perform file I/O operations that do not block the execution of your program. Instead, they operate asynchronously and use callback functions to handle the results of the operation when it completes. Asynchronous file system calls are the recommended approach in Node.js for handling file I/O operations, as they allow your application to remain responsive and handle multiple I/O operations concurrently without blocking the event loop.

Here's an explanation of asynchronous file system calls in Node.js:

1. **Non-blocking Nature:** When you use asynchronous file system methods, your program does not wait for the file operation to complete. Instead, it continues executing other tasks and registers a callback function to be called when the operation is finished. This non-blocking behavior is essential for keeping your Node.js application responsive, especially in scenarios with many concurrent operations.

2. **Method Naming:** Asynchronous file system methods in Node.js typically have the same names as their synchronous counterparts, but they do not have "Sync" in their names. For example, the asynchronous version of `fs.readFile` is `fs.readFile`, and the asynchronous version of `fs.writeFile` is `fs.writeFile`.
3. **Callback Function:** Asynchronous file system methods require you to provide a callback function that will be invoked when the operation completes. The callback function takes two arguments: an error object (if an error occurred during the operation) and the result of the operation (e.g., the data read from a file). You should handle errors and process the results within the callback function.
4. **Non-blocking Behavior:** Asynchronous file system calls do not block your application's event loop, which allows your application to remain responsive and handle multiple I/O operations concurrently. This is crucial for building scalable and efficient applications, such as web servers.
5. **Use Cases:** Asynchronous file system calls are the recommended choice for most Node.js applications, especially those that need to handle concurrent requests, such as web servers, REST APIs, and microservices. They are also suitable for I/O-bound operations, database interactions, and other tasks where non-blocking behavior is essential for maintaining responsiveness.

Example :

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

// Reading a file asynchronously

fs.readFile('example.txt', 'utf8', (err, data) => {

  if (err) {

    console.error('Error:', err);

  } else {

    console.log("File Data : \n ", data);

  }

});

// Writing to a file asynchronously

fs.writeFile('newfile2.txt', 'Hello, world!', 'utf8', (err) => {

  if (err) {
```

```
    console.error('Error:', err);

  } else {

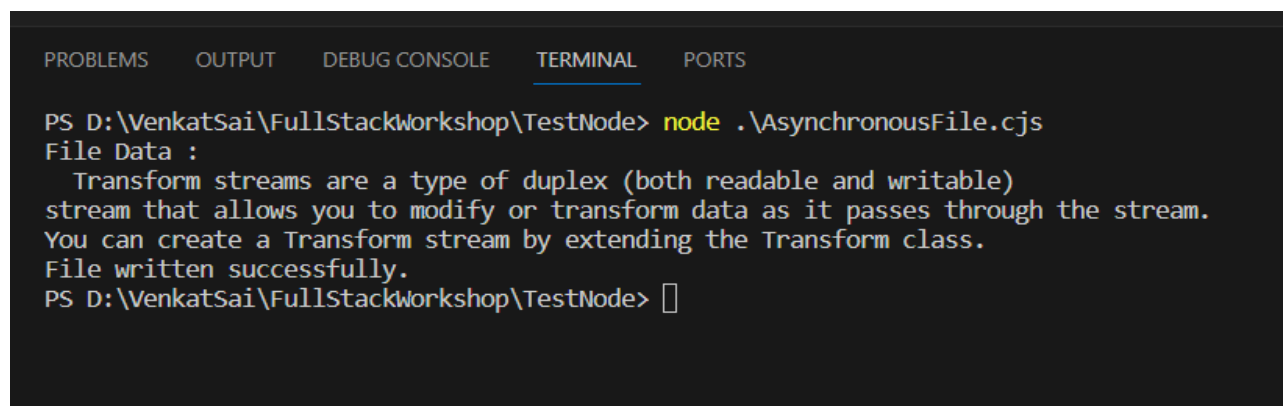
    console.log('File written successfully.');
```



```
  }

});
```

Output :



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\AsynchronousFile.cjs
File Data :
  Transform streams are a type of duplex (both readable and writable)
  stream that allows you to modify or transform data as it passes through the stream.
  You can create a Transform stream by extending the Transform class.
  File written successfully.
PS D:\VenkatSai\FullStackWorkshop\TestNode> 
```

Fig – 13 : Asynchronous Output

In summary, asynchronous file system calls in Node.js are crucial for building responsive and scalable applications. They allow you to efficiently handle I/O operations without blocking the event loop, making Node.js a powerful platform for handling a large number of concurrent requests and tasks.

3.2 Writing and reading Into Files :

File Writing :

File writing in Node.js involves using the built-in **fs** (File System) module to create, open, and write data to a file. Here's a more detailed explanation of how to write files in Node.js:

1. Import the fs module:

To use file system operations, you need to require the **fs** module at the beginning of your JavaScript file.

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

2. Writing Data to a File:

Node.js provides various methods to write data to a file. The two most common methods are `fs.writeFile` and `fs.createWriteStream`.

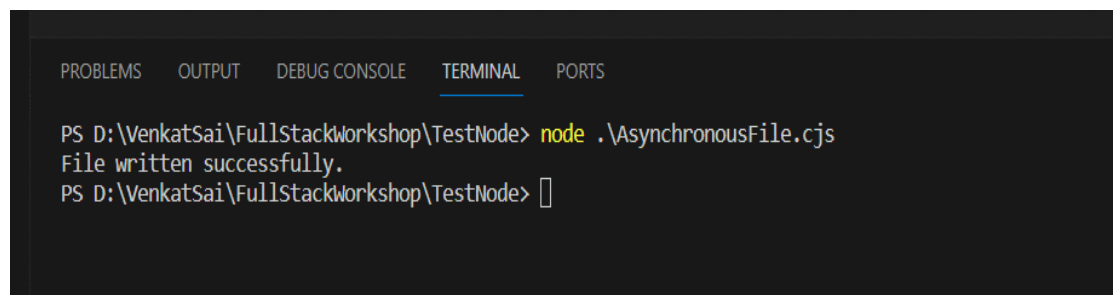
Using `fs.writeFile` (Asynchronous):

This method is asynchronous and is recommended for most use cases. It allows you to write data to a file without blocking the event loop, making it more efficient.

Example :

```
// Writing to a file asynchronously
fs.writeFile('newfile2.txt', 'Hello, world!', 'utf8', (err) => {
  if (err) {
    console.error('Error:', err);
  } else {
    console.log('File written successfully.');
```

Output :



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\AsynchronousFile.cjs
File written successfully.
PS D:\VenkatSai\FullStackWorkshop\TestNode> █
```

Fig – 14 : Writing File

In this example, you provide the file path ('example.txt'), the data you want to write (hello World !), and a callback function that will be executed when the writing operation is complete.

Using `fs.createWriteStream` (Stream-Based):

This method is suitable when you need to write large amounts of data to a file, as it allows you to write data in smaller chunks using streams.

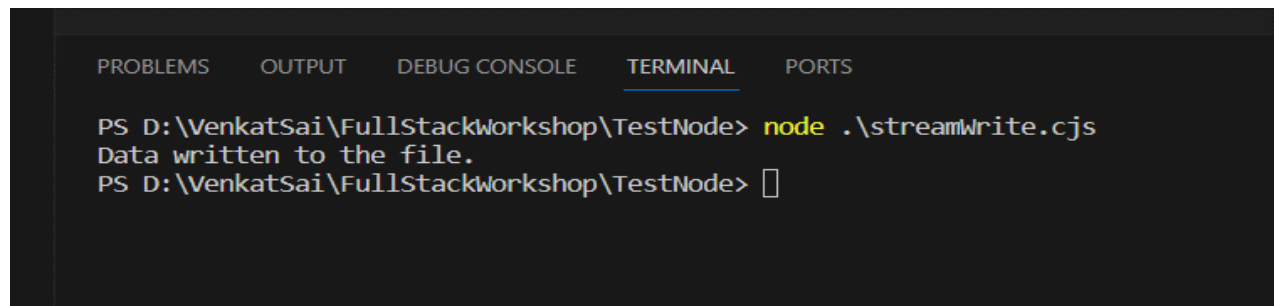
Example :

```
const fs = require('fs');
const writeStream = fs.createWriteStream('example2.txt');
writeStream.write('Hello, Node.js!\n');
writeStream.write('This is a new line of text.\n');
writeStream.end();
```

```
writeStream.on('finish', () => {  
  console.log('Data written to the file.');
```

```
});  
writeStream.on('error', (err) => {  
  console.error('Error writing to the file:', err);  
});
```

Output :

A screenshot of a VS Code terminal window. The terminal has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is active), and PORTS. The command prompt shows the user running 'node .\streamWrite.cjs' in the directory 'D:\VenkatSai\FullStackWorkshop\TestNode'. The output of the script is 'Data written to the file.' followed by a new line and a cursor.

```
PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\streamWrite.cjs  
Data written to the file.  
PS D:\VenkatSai\FullStackWorkshop\TestNode> █
```

Fig – 15 : Stream write

With `fs.createWriteStream`, you create a write stream to the file and use the `write` method to add data in chunks. The `end` method signals that you've finished writing. You can listen for the `'finish'` and `'error'` events to handle the writing process.

3. Handling Errors:

It's crucial to handle errors when writing files to ensure your application is robust. Both `fs.writeFile` and `fs.createWriteStream` methods provide error callbacks or events that you can use to handle potential issues during file writing.

4. Close the File:

When using `fs.createWriteStream`, it's important to wait for the `'finish'` event or the `'error'` event to ensure the file writing process is complete before closing the file or moving on to other tasks.

Remember that file I/O operations can be slow, so it's generally better to perform file writes asynchronously to avoid blocking your application's event loop. The `fs.writeFile` method is a good choice for most use cases, while `fs.createWriteStream` is more suitable for scenarios involving large files or streams of data.

File Reading :

File reading in Node.js involves using the built-in **fs** (File System) module to read data from a file. Here's a detailed explanation of how to read files in Node.js:

1. Import the fs module:

Start by requiring the **fs** module at the beginning of your JavaScript file:

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

2. Reading Data from a File:

Node.js provides several methods to read data from a file. The two most commonly used methods are `fs.readFile` and `fs.createReadStream`.

Using `fs.readFile` (Asynchronous):

This method is asynchronous and is recommended for most use cases. It allows you to read data from a file without blocking the event loop, making it more efficient.

Example :

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

In this example, you provide the file path ('example.txt'), the encoding ('utf8' in this case), and a callback function that will be executed when the reading operation is complete. The file content is passed to the callback as a string.

Using `fs.createReadStream` (Stream-Based):

This method is suitable when you need to read large files or stream data from a file. It allows you to read the file in smaller chunks using streams.

Example :

```
const readStream = fs.createReadStream('example.txt', 'utf8');  
readStream.on('data', (chunk) => {  
  console.log('Chunk of data:', chunk);  
});  
readStream.on('end', () => {  
  console.log('File reading is complete.');});  
readStream.on('error', (err) => {  
  console.error('Error reading the file:', err);  
});
```

With `fs.createReadStream`, you create a read stream from the file and listen for the 'data' event to receive data chunks. The 'end' and 'error' events are used to handle the reading process.

3. Handling Errors:

Just like when writing files, it's crucial to handle errors when reading files to ensure your application is robust. Both `fs.readFile` and `fs.createReadStream` methods provide error callbacks or events that you can use to handle potential issues during file reading.

4. Close the File:

When using `fs.createReadStream`, it's important to listen for the 'end' event or the 'error' event to ensure the file reading process is complete before closing the file or moving on to other tasks.

Reading files in Node.js is an essential part of many applications, and understanding how to efficiently read files is crucial for building reliable and performant software. The choice between `fs.readFile` and `fs.createReadStream` depends on the specific requirements of your application, with `fs.readFile` being more suitable for most use cases and `fs.createReadStream` being appropriate for reading large files or streams of data

CHAPTER – 4 : HTTP SERVICES

4.1 Processing URL's :

In Node.js, processing URLs typically involves parsing and manipulating Uniform Resource Locators (URLs) to extract or modify various components like the protocol, host, path, query parameters, and fragments. To work with URLs in Node.js, you can use the built-in `url` module, which provides methods for parsing and formatting URLs.

Here's a basic overview of how to process URLs in Node.js:

1. **Import the url module:** To use the `url` module, you need to import it at the beginning of your Node.js script.
2. **Parse a URL:** You can parse a URL string into an object to access its various components. The `url.parse()` method is commonly used for this purpose. The second argument `true` passed to `url.parse()` tells it to parse query parameters into an object.
3. **Modify a URL:** You can create or modify URLs by using the `url.format()` method. Here's an example of how to change the protocol and add a fragment to a URL.
4. **Resolve relative URLs:** If you need to resolve relative URLs with respect to a base URL, you can use the `url.resolve()` method.

Example :

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

// Parsing Url

const urlString =
'https://mjnvsai.github.io/Codegnan_MERN/RegistrationForm.html?fn=sai&mob=1234567892&email=sai.doc45%40gmail.com&us=123-456-7890&pass=dS1456&dob=12%2F12%2F2023';

const parsedUrl = url.parse(urlString, true);

console.log("Url Protocol : ", parsedUrl.protocol, "\n");

console.log("url Host : ", parsedUrl.host, "\n");

console.log("Sub Path : ", parsedUrl.path, "\n");

console.log("Value Query : ", parsedUrl.query);
```

```
// Modifying URL
```

```
const urlString2 = 'https://mjnvsai.github.io/Codegnan_MERN/Biography.html#gift';
```

```
const parsedUrl2 = url.parse(urlString2, true);
```

```
parsedUrl2.protocol = 'http:';
```

```
parsedUrl2.hash = '#pics';
```

```
const modifyurl = url.format(parsedUrl2)
```

```
console.log("Modifyied Url : ", modifyurl)
```

```
// Url resolve
```

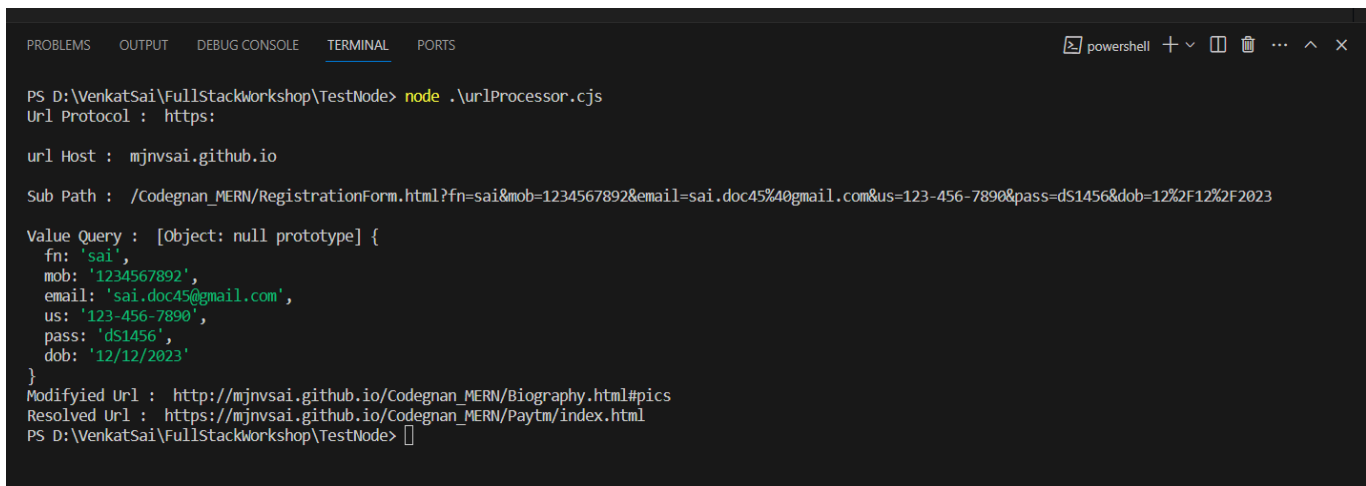
```
const baseUrl = 'https://mjnvsai.github.io/';
```

```
const relativeUrl = 'Codegnan_MERN/Paytm/index.html';
```

```
const resolvedUrl = url.resolve(baseUrl, relativeUrl);
```

```
console.log("Resolved Url : ", resolvedUrl);
```

Output :



```
PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\urlProcessor.cjs
Url Protocol : https:

url Host : mjnvsai.github.io

Sub Path : /Codegnan_MERN/RegistrationForm.html?fn=sai&mob=1234567892&email=sai.doc45@gmail.com&us=123-456-7890&pass=dS1456&dob=12%2F12%2F2023

Value Query : [Object: null prototype] {
  fn: 'sai',
  mob: '1234567892',
  email: 'sai.doc45@gmail.com',
  us: '123-456-7890',
  pass: 'dS1456',
  dob: '12/12/2023'
}
Modifyied Url : http://mjnvsai.github.io/Codegnan_MERN/Biography.html#pics
Resolved Url : https://mjnvsai.github.io/Codegnan_MERN/Paytm/index.html
PS D:\VenkatSai\FullStackWorkshop\TestNode>
```

Fig – 16 : Url Formatting

4.2 Understanding Request :

In Node.js, handling HTTP requests is a common task when building web applications and APIs. Node.js provides built-in modules to help you handle incoming HTTP requests and process them. To understand how to handle requests in Node, you need to become familiar with the http and express modules.

1. Using the http Module:

Node.js has a built-in **http** module that allows you to create an HTTP server and handle incoming requests. Here's a basic example of how to create an HTTP server and handle requests :

Code :

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

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

  if (req.method === 'GET' && req.url === '/') {

    res.writeHead(200, { 'Content-Type': 'text/plain' });

    res.end('You Are In Google Browser\n');

  } else {

    res.writeHead(404, { 'Content-Type': 'text/plain' });

    res.end('Not Found\n');

  }

});

const port = 3000;

server.listen(port, () => {

  console.log(`Server is running on port ${port}`);

});
```

Output :

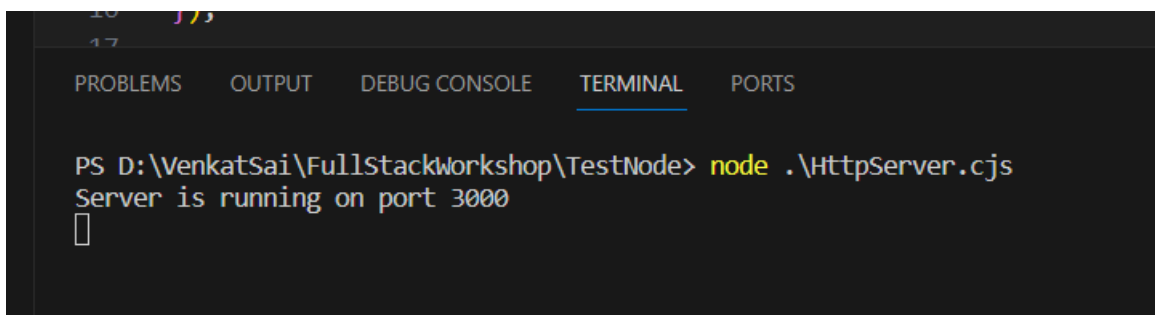
A screenshot of a Visual Studio Code terminal window. The terminal has tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is active), and 'PORTS'. The command prompt shows 'PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\HttpServer.cjs'. The output of the command is 'Server is running on port 3000' followed by a cursor. The background of the terminal is dark with light-colored text.

Fig – 17 : Http Server

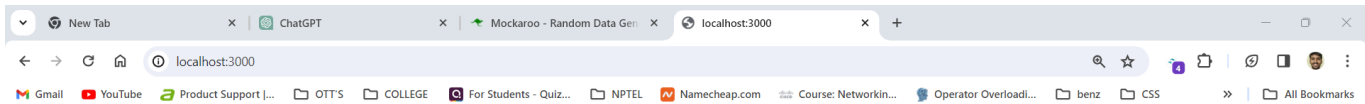


Fig – 18 : Server Output

2. Handling Request Parameters and Data:

In both the http and Express examples, you can access request parameters and data using the req object. For example, to access query parameters, you can use req.query (in Express) or url.parse(req.url, true).query (in the http module). To access request body data, you can use middleware like body-parser in Express or read the request stream directly in the http module.

3. Routing:

In more complex applications, you'll often define routes to handle different parts of your API or web application. Express provides a robust routing system, making it easy to handle various URLs and HTTP methods with different functions.

These are the basics of handling HTTP requests in Node.js. Depending on your specific use case, you may need to dive deeper into topics like middleware, authentication, and error handling, which are essential when building real-world applications.

4.3 Response and Server objects :

In Node.js, the Response and Server objects are not built-in objects, but they are typically associated with building web servers using the Node.js core modules like http or external libraries like Express.js. These objects are essential for handling HTTP requests and responses in a Node.js web server.

1. **Response Object:** The Response object, often referred to as res, represents the HTTP response that a server sends back to the client in response to an HTTP request. It provides methods and properties that allow you to control and manipulate the response sent to the client. Here are some of the common methods and properties of the Response object:

1. **res.write():** This method is used to send data to the client's browser in chunks.

2. **res.end()**: This method is used to end the response and send any remaining data to the client.
3. **res.setHeader(name, value)**: Sets an HTTP header on the response.
4. **res.statusCode**: Represents the HTTP status code to be sent with the response (e.g., 200 for a successful request, 404 for not found, etc.).
5. **res.statusMessage**: Represents the status message associated with the status code.
6. **res.send()**: A common method in Express.js for sending a response with automatic content type and status code.

Example of creating an HTTP server using Node's built-in http module and sending a simple response:

Code :

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

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

  if (req.method === 'GET' && req.url === '/') {

    res.writeHead(200, { 'Content-Type': 'text/plain' });

    res.end('You Are In Google Browser\n');

  } else {

    res.writeHead(404, { 'Content-Type': 'text/plain' });

    res.end('Not Found\n');

  }

});

const port = 3000;

server.listen(port, () => {

  console.log(`Server is running on port ${port}`);

});
```

2. Server Object:

The Server object is not a standard object in Node.js, but it is typically a reference to the HTTP server instance created when you use the http module or when you create a web server with a framework like Express.js. The server object allows you to start the server, listen on a specific port, and handle incoming HTTP requests.

In the example above, server is the Server object created with `http.createServer()`. You can call methods like `server.listen()` to specify the port and host where the server should listen for incoming requests.

3. The Server object can also have event listeners to handle various server-related events, such as 'listening', 'request', 'close', and more. For example, you can add an event listener to handle server listening events:

Code :

```
server.on('listening', () => {  
  console.log('Server is now listening on port 3000');  
});
```

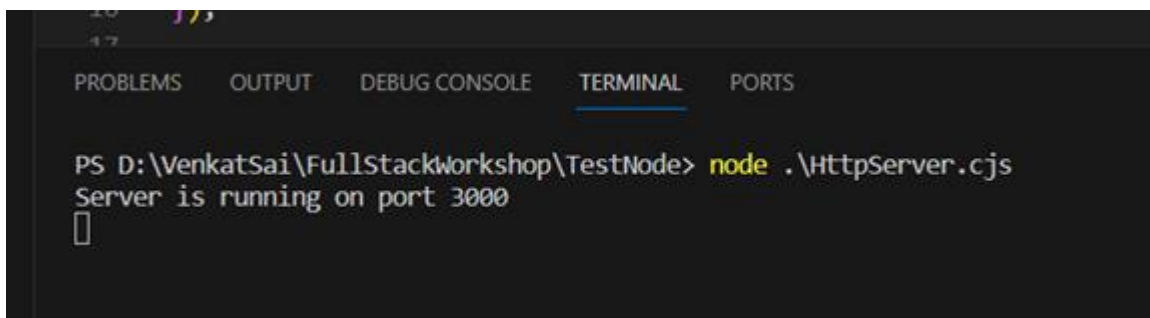


Fig – 19 : Request



Fig – 20 : Response

The Response and Server objects are fundamental components when building web servers in Node.js, and they are essential for handling incoming HTTP requests and sending appropriate responses to

clients. Depending on your use case, you may also use external libraries like Express.js to simplify the process of handling requests and responses in a more structured manner.

4.4 Implementing Http Clients and Servers in Node :

Creating HTTP clients and servers in Node.js is a common task. Node.js provides built-in modules for handling HTTP operations, making it relatively straightforward to implement both HTTP clients and servers. Below are examples of how to create HTTP clients and servers in Node.js.

HTTP CLIENT :

To create an HTTP client in Node.js, you can use the `http` or `https` modules. Here's an example of an HTTP client using the `http` module to make a GET request to a remote server and print the response :

Example :

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

const options = {
  hostname: 'localhost',
  port: 3000, // Port on which your local HTTP server is running
  path: '/',
  method: 'GET',
};

const req = http.request(options, (res) => {
  let data = '';
  res.on('data', (chunk) => {
    data += chunk;
  });
  res.on('end', () => {
    console.log(data);
  });
});
```

```
});  
  
req.on('error', (error) => {  
  
  console.error('Request error:', error);  
  
});  
  
req.end();
```

In this example, we use the `http.request()` method to create an HTTP GET request to a remote server. You can adjust the options object to customize the request as needed.

HTTP SERVER :

To create an HTTP server in Node.js, you can use the `http` module as well. Here's an example of a simple HTTP server.

Example :

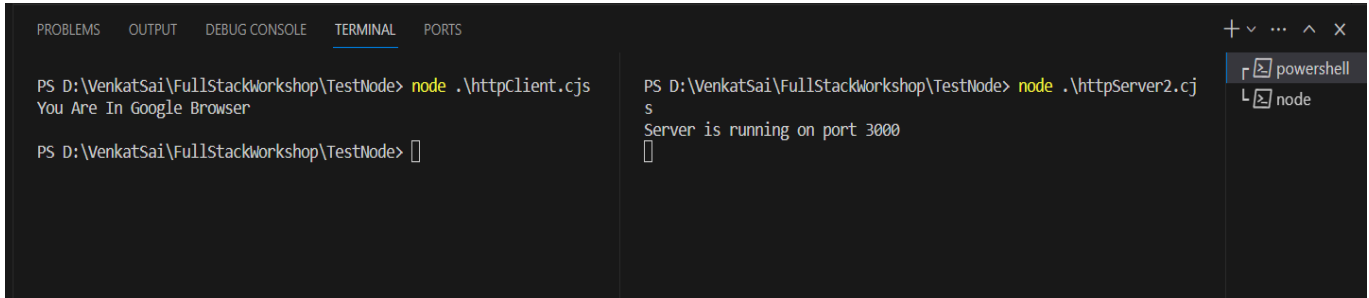
```
const http = require('http');  
  
const server = http.createServer((req, res) => {  
  
  res.writeHead(200, { 'Content-Type': 'text/plain' });  
  
  res.end('Hello, World!\n');  
  
});  
  
const PORT = 3000;  
  
server.listen(PORT, () => {  
  
  console.log(`Server is running on port ${PORT}`);  
  
});
```

In this example, we create an HTTP server using the `http.createServer()` method. When a request is received, the server responds with "Hello, World!" and sets the response header to indicate that the content is plain text. You can adjust the response logic and headers to suit your application's needs.

To run these examples, save the code to a file (e.g., `http-client.js` and `http-server.js`) and execute them with Node.js.

The HTTP client will make a request to the specified server, and the HTTP server will start and listen on the specified port. You can access the server by opening a web browser or using a tool like **curl** to make a request to `http://localhost:3000` in the case of the HTTP server example.

Outputs :



```
PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\httpServer2.js
Server is running on port 3000

PS D:\VenkatSai\FullStackWorkshop\TestNode> node .\httpClient.js
You Are In Google Browser

PS D:\VenkatSai\FullStackWorkshop\TestNode>
```

Fig – 21 : Client and Server



Fig – 22 : Server Output

The output you're seeing when running the HTTP client code indicates that you made a request to "example.com" and received the HTML content of the example domain's homepage. This is expected, as the code I provided in the HTTP client example sends a GET request to "example.com" and logs the response.

In your case, you received the HTML content of the example domain's homepage, which is just a sample web page that often used for illustrative purposes.

If you want to make requests to a different server or obtain specific data from a remote server, you should modify the options object to target the desired server and endpoint, and then parse the response as needed. The example I provided is a basic demonstration of how to perform an HTTP GET request in Node.js, and it can be customized to fit your specific requirements.

Example :

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

const fs = require('fs');

const options = {

  key: fs.readFileSync('key.pem'),

  cert: fs.readFileSync('cert.pem')

};

const server = https.createServer(options, (req, res) => {

  res.writeHead(200);

  res.end('You Web Browser is Working On HTTPS\n');

});

server.listen(443, () => {

  console.log('Server is listening on port 443');

});
```

This code creates an HTTPS server using the self-signed certificate and responds with a simple message.

You also need to install the fs and https module packages using below command.

Command : npm install fs https

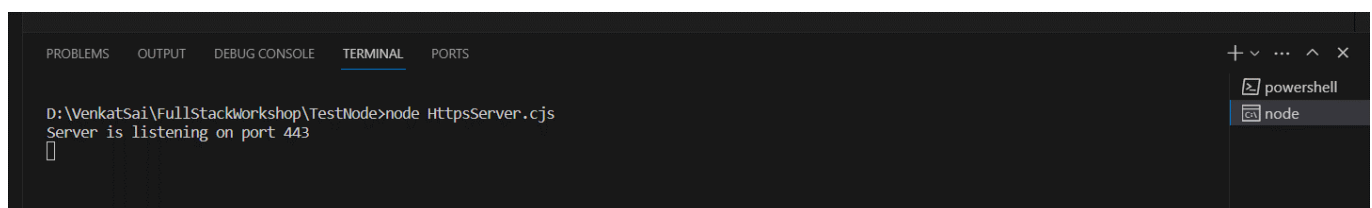
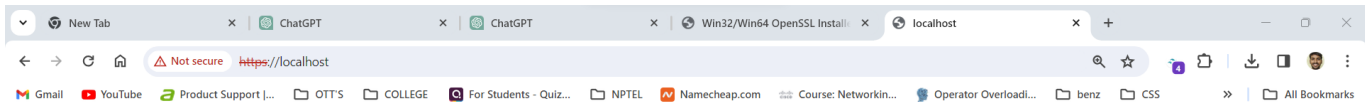
Output :

Fig – 24 : Https server



You Web Browser is Working On HTTPS

Fig – 25 : Https output

HTTPS CLIENT IMPLEMENTATION :

1. **Importing the https Module:** Start by importing the https module in your Node.js application using `const https = require('https');`.
2. **Creating an HTTPS Request:** Use the `https.request()` method to create an HTTPS request. Provide an options object that includes the target server's hostname, port (usually 443 for HTTPS), the path to the specific endpoint you want to access, and the HTTP method (e.g., GET, POST).
3. **Handling the Response:** The `https.request()` method takes a callback function that is invoked when the server responds. You can accumulate and process the response data as it arrives, typically in chunks, using event listeners like `data` and `end` on the response object.
4. **Error Handling:** Implement error handling by adding an error event listener to the request object. This ensures that you can handle and log any errors that might occur during the request.
5. **Sending the Request:** After setting up the request object, you need to call `req.end()` to actually send the request to the server.
6. **Completing the Example:** A complete example combines these steps. Replace the placeholders with your actual server details and endpoints to make secure HTTPS requests.
7. **Secure Communication:** The https module ensures secure communication over the HTTPS protocol by establishing an encrypted connection using SSL/TLS.
8. **Authentication and Certificates:** In production environments, you'd typically handle SSL/TLS certificates and validate server certificates to ensure the authenticity of the server.
9. **Handling Responses:** Depending on your use case, you might parse and process the response data, which can be HTML, JSON, XML, or any other format.

10. Scalability and Error Handling: When building applications, consider error handling, scalability, and best practices, like connection pooling, to efficiently manage and scale your HTTPS client for various tasks.

This client allows you to interact with secure HTTPS services and retrieve data from remote servers, making it essential for applications requiring secure communication over the web.

Example :

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

const options = {

  hostname: 'localhost', // Replace with your target server's hostname

  port: 443,

  path: '/', // Replace with the specific endpoint

  method: 'GET',

  agent: new https.Agent({

    rejectUnauthorized: false // Ignore self-signed certificates

  })

};

const req = https.request(options, (res) => {

  let data = '';

  res.on('data', (chunk) => {

    data += chunk;

  });

  res.on('end', () => {

    console.log(data);

  });

});

req.on('error', (error) => {
```

```
console.error(error);  
  
});  
  
req.end();
```

Output :

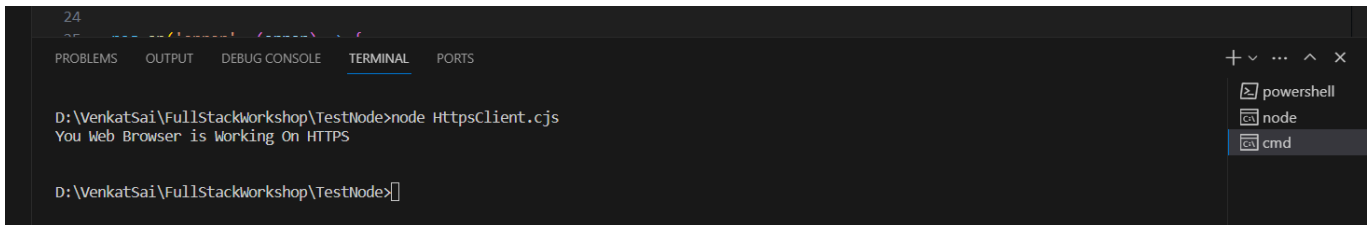


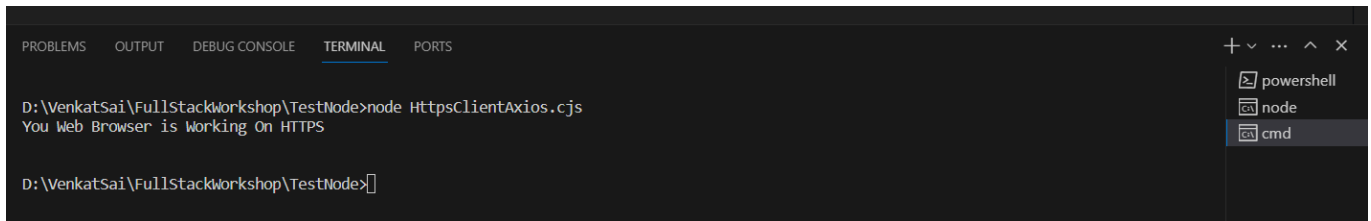
Fig – 26 : https client

Implementing HTTPS Client Using Axios :

Example :

```
const axios = require('axios');  
  
const https = require('https');  
  
const agent = new https.Agent({  
  
  rejectUnauthorized: false, // Ignore self-signed certificates  
  
});  
  
const axiosInstance = axios.create({  
  
  httpsAgent: agent,  
  
});  
  
axiosInstance.get('https://localhost:443')  
  
  .then((response) => {  
  
    console.log(response.data);  
  
  })  
  
  .catch((error) => {  
  
    console.error(error); });
```

Output :



The screenshot shows a VS Code terminal window with the 'TERMINAL' tab selected. The terminal displays the following output:

```
D:\VenkatSai\FullStackWorkshop\TestNode>node HttpClientAxios.cjs
You Web Browser is Working On HTTPS

D:\VenkatSai\FullStackWorkshop\TestNode>
```

On the right side of the terminal, there is a list of shell options: powershell, node, and cmd. The 'cmd' option is currently selected.

Fig – 27 : Axios Client

CHAPTER – 5 : EXPRESS JS

5.1 Implementing Express In Node Js :

Express.js is a popular web application framework for Node.js that simplifies the process of building web applications and APIs. It comes with various features and allows you to use middleware to extend and customize its functionality.

Express Features :

1. **Routing:** Express simplifies routing by allowing you to define routes and associate them with specific HTTP methods and URL patterns. This makes it easy to handle different types of requests and organize your application's code.
2. **Middleware:** Express uses middleware functions to process and modify incoming requests and outgoing responses. Middleware can be used for tasks like request parsing, authentication, logging, error handling, and more.
3. **HTTP Utility Methods:** Express provides a set of utility methods for dealing with HTTP, such as sending responses, setting headers, and redirecting to other URLs.
4. **Template Engines:** Although Express itself doesn't have a built-in template engine, it can be easily integrated with popular template engines like EJS, Handlebars, Pug, and Mustache for rendering dynamic HTML pages.
5. **Static File Serving:** You can use Express to serve static files (e.g., CSS, JavaScript, images) from a specified directory, making it easier to include these assets in your web pages.
6. **Database Integration:** Express can be used with various databases, both SQL and NoSQL, to create dynamic applications with database interactions. It's database-agnostic and provides flexibility in choosing your data storage solutions.
7. **Session Management:** Express provides mechanisms for handling user sessions, such as managing user authentication, storing session data, and handling cookies.
8. **Error Handling:** You can define error-handling middleware to capture and respond to errors in a structured way, making it easier to handle exceptions and maintain application stability.
9. **Security:** While Express itself doesn't handle all security concerns, it provides a foundation for implementing security features like data validation, authentication, and authorization.

Express Middleware :

Middleware in Express are functions that have access to the request and response objects as well as the next middleware function in the application's request-response cycle. Middleware functions can be added in the request-response pipeline and are executed in the order they are defined. They play a crucial role in extending and customizing the behavior of an Express application.

Here's how middleware works and some common use cases:

- **Request Parsing:** Middleware can be used to parse and process incoming data in various formats (e.g., JSON, URL-encoded) before it reaches your route handlers.
- **Authentication and Authorization:** You can use middleware to implement user authentication and authorization checks. For example, you can verify user credentials, check user roles, or manage access control.
- **Logging:** Middleware can log incoming requests, responses, and errors for debugging and monitoring purposes. This helps in tracking the application's behavior.
- **Validation:** You can use middleware to validate incoming data, such as form submissions or API payloads, to ensure it meets your application's requirements.
- **CORS (Cross-Origin Resource Sharing) Handling:** Middleware can handle CORS headers to control which domains are allowed to access resources on your server.
- **Error Handling:** Middleware can capture and handle errors, providing a consistent way to deal with unexpected issues in your application.
- **Security Headers:** Middleware can add security-related HTTP headers to enhance your application's security, such as setting Content Security Policy (CSP) or HTTP Strict Transport Security (HSTS) headers.
- **Compression:** Middleware like compression can be used to compress response data to reduce bandwidth usage and improve performance.
- **Session Management:** Middleware can handle session data, managing user sessions and maintaining session state.
- **Custom Middleware:** You can create custom middleware functions to address specific needs unique to your application.

The order in which middleware is defined matters. Middleware functions defined earlier in the code will be executed first in the request-response cycle. You can choose when and where to apply specific middleware functions to control the flow of your application and add custom functionality as needed.

Implementation :

1. **Set up a Node.js Project:** If you don't already have a Node.js project, you'll need to create one. You can do this by creating a new directory for your project and running “ npm init -y ” to set up a package.json file with project dependencies.
2. **Install Express.js:** Install Express.js as a dependency in your project by running the following command. **Command :** “ npm install express ”
3. **Create a Basic Express App:** Create a JavaScript file (e.g., app.js) in your project directory to start building your Express application. In this file, you'll set up a basic Express app.

Example Code :

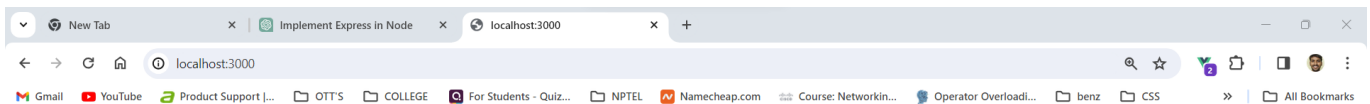
```
const express = require('express');
const app = express();
const port = 3000; // Set your desired port number
// Define a route
app.get('/', (req, res) => {
  res.send('Server is Running on Express and Node !');
});
// Define a route with a URL parameter
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`User ID: ${userId}`);
});
// Example middleware function
app.use((req, res, next) => {
  console.log('Middleware function is executed.');
```

next(); // Move to the next middleware or route handler

```
});
// Start the server
app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

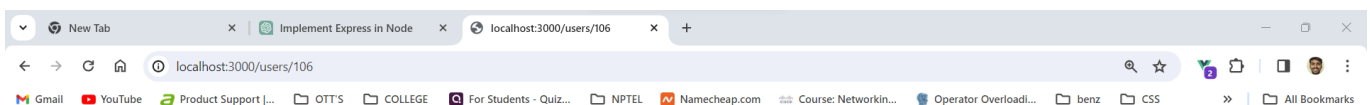
4. Your Express application should now be running, and you can access it in your web browser at **http://localhost:3000** (or the port you specified).
5. **Create More Routes and Middleware:** Express allows you to define more routes and use middleware to handle various aspects of your application. For example, you can add routes for handling different HTTP methods and URL paths, and you can use middleware functions to perform tasks like parsing request data, authentication, and error handling.

Output :



Server is Running on Express and Node !

Fig – 28 : Express and Node



User ID: 106

Fig – 29 : Routes

5.2 Configuring Express Settings :

Configuring settings in an Express.js application is an essential part of building a Node.js web server. Express allows you to set various configurations to control how your application behaves. You typically configure these settings at the application level or for specific routes and middleware. Here's how you can configure settings in an Express application:

1. **Create an Express Application:** First, create an Express application by requiring the Express framework:

Code :

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

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

2. **Set Application-level Settings:** You can configure settings at the application level using `app.set()`. Common application-level settings include the port number, view engine, and other global configurations.

Code :

```
app.set('port', 3000); // Set the port number
```

```
app.set('view engine', 'ejs'); // Set the view engine (if using templates)
```

3. **Access Application-level Settings:** You can access these settings later in your application using `app.get()` :

Code :

```
const port = app.get('port');
```

```
const viewEngine = app.get('view engine');
```

4. **Configure Middleware:** Some middleware may have their own configuration options. For example, the `body-parser` middleware can be configured like this:

Code :

```
const bodyParser = require('body-parser');
```

```
app.use(bodyParser.urlencoded({ extended: false }));
```

5. **Set Environment-specific Settings:** You can set environment-specific settings for development, production, or other custom environments using `app.configure()` (deprecated in Express 4) or by directly setting properties on `app.locals` or `app.settings`.

Code :

```
if (process.env.NODE_ENV === 'development') {
```

```
  app.set('debug', true);
```

```
}
```

6. **Custom Middleware Configuration:** You can also configure your own custom middleware to perform specific actions based on settings you define. For example:

```
app.use((req, res, next) => {
```

```
  if (app.get('debug')) {
```

```
    console.log('Debug mode is enabled');
```

```
  }
```

```
  next();
```

```
});
```


7. **Route-specific Settings:** You can set route-specific settings by using router-level middleware and configuring settings on the router object.

Code :

```
const router = express.Router();
router.get('/special', (req, res) => {
  res.send('This is a special route');
});
app.use('/myroute', router);
```

8. **Error Handling:** You can configure error handling settings using middleware like `app.use((err, req, res, next) => {})` for error handling, and customize error responses based on settings.

Application-level settings: `app.set('key', 'value')` and `app.get('key')`.

Middleware-specific settings: Configuration options vary by middleware and are often documented in the middleware's documentation.

Environment-specific settings: Use `process.env.NODE_ENV` and conditional statements.

Route-specific settings: Configure settings on router objects for specific routes.

Remember that some settings are specific to certain middleware or features you use in your Express application. Always refer to the Express and middleware documentation for detailed information on configuration options.

5.3 Starting Express Server :

Starting an Express server involves a series of steps that you can follow to create and run a basic server. Express is a popular web application framework for Node.js. Here's a simple outline of the steps :

1. **Initialize your Node.js project** If you haven't already set up a Node.js project, you can do so by running the following commands in your project's root directory.

Command : “ `npm init -y` ”

2. **Install Express:** You'll need to install the Express package. Run the following command.

Command : “ `npm install express` ”

3. **Create your Express application:** Create a JavaScript file (e.g., `app.js` or `server.js`) to build your Express application. You can use a code editor to create and edit this file.
4. **Import Express and create an instance:** In your JavaScript file, import Express and create an Express application:

Code :

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

5. **Define Routes:** Define the routes and their handlers. For example, you can create a simple "Hello, World!" route.

Code :

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

6. **Start the Server:** Start the Express server by binding it to a specific port. Typically, port 3000 is used for development.

Code :

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

7. **Run the Server:** In your terminal, run your Express application using Node.js.

Command : “ node app.js “

8. **Access Your Server:** You can now access your server by opening a web browser and navigating to **http://localhost:3000** (or the port you specified in your code).

That's it! You've successfully created and started an Express server. You can continue to define more routes, middleware, and add functionality to your server as needed for your web application.

5.4 Configuring Routes :

Configuring routes in Express is a fundamental part of building web applications using Node.js. Express is a popular web framework for Node.js that makes it easy to define routes and handle HTTP requests. Here are the basic steps to configure routes in an Express application:

1. **Setup Express:** First, you need to set up an Express application. You can do this by installing Express using npm or yarn and requiring it in your Node.js file. Here's an example:

Code :

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

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

```
const port = 3000;
```

2. **Define Routes:** Express uses the `app` object to define routes for different HTTP methods (GET, POST, PUT, DELETE, etc.). You can define routes using the following format:

Code :

```
app.HTTP_METHOD('route', callback);
```

Where `HTTP_METHOD` is the HTTP method for the route (e.g., `get`, `post`, `put`, `delete`), `route` is the URL path for the route, and `callback` is a function that gets executed when a request matches the specified route. Here's an example of defining a simple GET route:

Code :

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

3. **Route Parameters:** You can use route parameters to capture values from the URL. Parameters are defined with a colon (:) and can be accessed in the callback function using `req.params`. For example:

Code :

```
app.get('/user/:id', (req, res) => {  
  const userId = req.params.id;  
  res.send(`User ID: ${userId}`);  
});
```

4. **Route Handlers:** The callback function provided to the route can handle the incoming request and send a response. You can render HTML, send JSON data, or perform any other operations required for your application.

5. **Middleware:** You can use middleware functions in Express to perform tasks like authentication, logging, or modifying the request or response objects before they reach the route handler. Middleware functions can be applied globally or to specific routes. Here's an example of defining and using middleware:

Code :

```
function myMiddleware(req, res, next) {  
  
  // Do something before reaching the route handler  
  
  next(); // Call next to move to the next middleware or route handler  
  
}  
  
app.use(myMiddleware); // Apply middleware globally  
  
app.get('/myroute', myMiddleware, (req, res) => {  
  
  // Route handler  
  
});
```

6. **Start the Server:** Finally, you need to start the Express server to listen for incoming requests on a specified port.

Code :

```
app.listen(port, () => {  
  
  console.log(`Server is running on port ${port}`);  
  
});
```

With these basic steps, you can configure routes in Express and build a web application that responds to various HTTP requests. Express provides a flexible and powerful way to define routes and handle web traffic in your Node.js applications.

5.4 Using Request Objects :

In Express.js, a request object, often denoted as `req`, is an important part of handling HTTP requests in your web application. It contains information about the incoming HTTP request, such as the request method, URL, headers, and any data sent in the request body. Understanding and using the `req` object is crucial for building web applications with Express. Here's a guide on how to work with request objects in Express.

1. **Accessing req Object:** In an Express route handler function, you can access the `req` object as one of the function parameters.

2. **Request Method:** You can access the HTTP request method (e.g., GET, POST, PUT, DELETE) using `req.method`.
3. **Request URL:** You can access the requested URL using `req.url`.
4. **Request Headers:** To access request headers, you can use `req.headers`.
5. **Request Parameters:** If your route contains URL parameters (e.g., `/users/:id`), you can access them through `req.params`.
6. **Query Parameters:** Query parameters (e.g., `/search?q=keyword`) can be accessed using `req.query`.
7. **Request Body:** If you're handling POST or PUT requests with a request body (e.g., JSON or form data), you can access the request body using middleware like `express.json()` or `express.urlencoded()`. The parsed body is available in `req.body`.
8. **Cookies and Sessions:** You can access cookies and sessions using `req.cookies` and `req.session`, respectively. Make sure to use appropriate middleware for session management and cookie parsing.

The `req` object is a central part of Express and provides all the information you need to handle and respond to incoming HTTP requests effectively. Understanding its properties and how to access them is fundamental for building web applications using Express.

Example :

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

const app = express();

const port = 3000;

// Middleware to parse JSON request bodies
app.use(express.json());

// Route to handle a GET request
app.get('/example', (req, res) => {

  // Access the req object properties

  const method = req.method;

  const url = req.url;

  const userAgent = req.headers['user-agent'];

  // Send a response
```

```
res.send(`Request Method: ${method}, Request URL: ${url}, User-Agent: ${userAgent}`);  
  
});  
  
// Route with URL parameters  
  
app.get('/users/:id', (req, res) => {  
  
  const userId = req.params.id;  
  
  res.send(`User ID: ${userId}`);  
  
})  
  
// Route with query parameters  
  
app.get('/search', (req, res) => {  
  
  const query = req.query.q;  
  
  res.send(`Search Query: ${query}`);  
  
});  
  
// Route to handle a POST request  
  
app.post('/data', (req, res) => {  
  
  // Access the request body  
  
  const data = req.body;  
  
  res.send(`Received data: ${JSON.stringify(data)}`);  
  
});  
  
// Start the Express server  
  
app.listen(port, () => {  
  
  console.log(`Server is running on http://localhost:${port}`);  
  
});
```

Output :

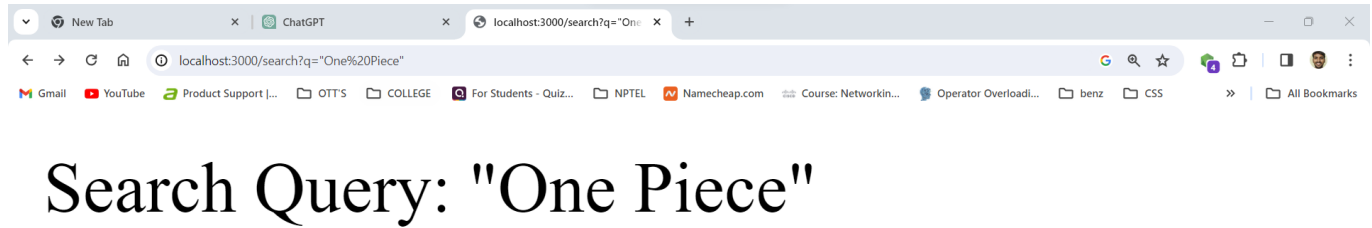


Fig – 30 : Search Route

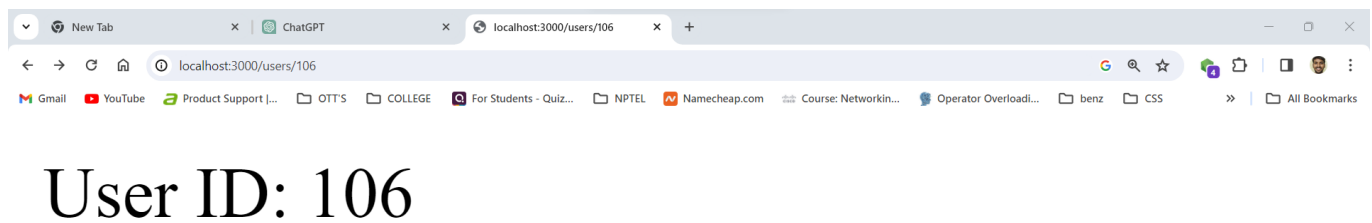


Fig – 31 : User Route

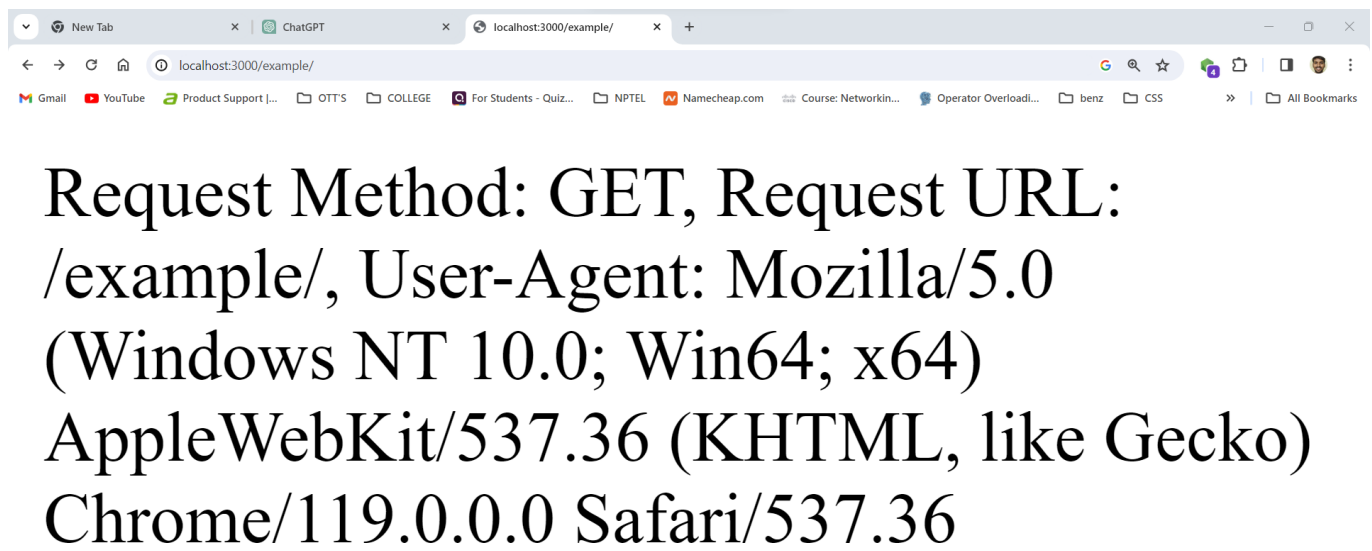


Fig – 32 : Example Route

5.5 Using Response Objects :

Response objects in Express, a popular web application framework for Node.js, play a crucial role in handling HTTP responses efficiently. They are essential for several reasons:

1. **Data Representation:** Response objects allow you to format and send data in various formats, such as JSON, HTML, or plain text, providing flexibility in how your server communicates with clients.
2. **Status Codes:** They enable setting appropriate HTTP status codes (e.g., 200 for success, 404 for not found, 500 for server errors) to convey the outcome of the request to clients.
3. **Headers:** Response objects allow you to set response headers, which are essential for controlling caching, security, and other response behaviors.
4. **Middleware Compatibility:** Express middleware can manipulate and enhance response objects, making it easier to add functionalities like compression, authentication, and error handling.
5. **Content-Type:** You can specify the content type, which informs clients how to interpret the response data, ensuring proper rendering and processing.
6. **Redirects:** Response objects facilitate redirection, helping route clients to different endpoints or URLs as needed.
7. **Send Methods:** They provide convenient methods like **res.send()** to simplify sending data, reducing boilerplate code.
8. **Template Rendering:** For server-side rendering, response objects help in rendering dynamic templates with data.
9. **File Downloads:** They allow you to send files for download by setting appropriate headers and content disposition.
10. **Cross-Origin Resource Sharing (CORS):** Response objects enable configuring CORS headers to control which origins can access resources, enhancing security.
11. **Compression:** Response objects can be used with compression middleware to reduce data transfer size and improve performance.
12. **Custom Responses:** You can craft custom responses tailored to your application's requirements, making your API or website more user-friendly.

13. **Error Handling:** Express response objects make it easy to send error responses with meaningful messages and status codes.
14. **Authentication:** They can be used to send authentication challenges or responses, ensuring secure access to resources.
15. **Testing:** Response objects are vital for testing routes and endpoints, allowing you to verify the expected behavior of your API or web application.
16. **Monitoring and Logging:** Response objects can be instrumented to gather statistics, track usage, and generate logs for analysis and troubleshooting.
17. **Request-Response Cycle:** They are the bridge between client requests and server responses, making them a fundamental component in the request-response cycle.
18. **Performance Optimization:** Properly configured response objects can improve website speed and performance by setting cache headers and enabling content compression.
19. **Consistency:** They help maintain a consistent and standardized API or website behavior for all client interactions.
20. **Compliance:** Response objects enable compliance with web standards, making your application compatible with various web technologies and clients.

In Express, the response object, often denoted as `res`, is a crucial component for handling and sending HTTP responses to clients. It allows developers to control the data and metadata sent back to the client in response to a request. Through the `res` object, you can set HTTP status codes to indicate the outcome of the request, specify response headers for content type, caching, and security, send data in various formats (like JSON or HTML), and even redirect clients to different URLs. It simplifies the process of crafting responses, enhancing the efficiency and functionality of web applications and APIs built with Express. Additionally, middleware can be used to further manipulate the `res` object, enabling features like compression, authentication, and error handling to be seamlessly integrated into the response process.

CHAPTER – 6 : ANGULAR

6.1 Understanding Angular :

Angular is a popular open-source front-end web application framework developed and maintained by Google and a community of individual developers and corporations. It's designed to help developers build dynamic, single-page web applications with a focus on modularity and testability. Here are some key concepts to help you understand Angular :

1. **Components:** In Angular, applications are built using components. A component is a self-contained, reusable piece of the user interface that consists of an HTML template, TypeScript code, and styles. Components are organized in a hierarchical structure and can communicate with each other.
2. **Modules:** Angular applications are typically organized into modules. A module is a container for a group of related components, services, and other code. Modules help you organize your code and provide a way to encapsulate functionality.
3. **Templates:** Templates define the structure and layout of your application's user interface. Angular templates are written in HTML with additional Angular-specific syntax for data binding and structural directives.
4. **Data Binding:** Data binding is a core concept in Angular that allows you to bind data between your components and the DOM. You can use one-way data binding (interpolation), two-way data binding (ngModel), and event binding to update and respond to changes in your application's data.
5. **Directives:** Angular provides a set of directives that allow you to extend HTML with new behavior. For example, the ngFor directive is used for rendering lists, and the ngIf directive is used for conditional rendering.
6. **Services:** Services are used to encapsulate and share common functionality, such as making HTTP requests, data storage, and other business logic. Services are typically injected into components to provide them with the required functionality.
7. **Dependency Injection:** Angular uses a dependency injection system to manage the creation and sharing of application components and services. This promotes modularity, testability, and maintainability.
8. **Routing:** Angular has a powerful routing system that allows you to create a single-page application with multiple views. You can define routes and associate components with those routes, enabling navigation within your app.

9. **Observables:** Angular often uses Observables to handle asynchronous operations, such as HTTP requests and event handling. Observables are part of the RxJS library and provide a way to work with asynchronous data streams.
10. **Testing:** Angular has a strong focus on testing, and it provides tools and libraries for writing unit tests and end-to-end tests for your application. This promotes the development of robust and maintainable code.
11. **CLI (Command Line Interface):** Angular CLI is a powerful tool for generating code, managing dependencies, and building and deploying Angular applications. It simplifies many common development tasks.
12. **Angular Ecosystem:** Angular is supported by a rich ecosystem of libraries, extensions, and tools. Some of the notable ones include Angular Material for UI components, NgRx for state management, and Angular Fire for Firebase integration.

To get started with Angular, you'll need to install Angular CLI, create an Angular project, and start building components, templates, and services. Angular has a learning curve, but it offers powerful features for building complex web applications. You can find official documentation, tutorials, and community resources to help you on your Angular journey.

6.2 Separation Of Responsibilities :

In Angular, Separation of Responsibilities (SoR) is a key principle for organizing and structuring your application's code to make it more maintainable and scalable. SoR aims to ensure that each part of your application has a clear and distinct responsibility, reducing the coupling between different components and improving code reusability. Angular provides a framework that encourages and facilitates SoR through various features and patterns. Here are the main areas where you can apply SoR in an Angular application:

1. **Component-Based Architecture:** Angular applications are typically organized into components, which are the building blocks of the user interface. Each component should have a well-defined responsibility, such as rendering a specific part of the UI or handling a specific feature.
2. **Template Views:** Keep the presentation logic in the component's template (HTML). Templates should focus on displaying data and user interactions and should not contain business logic. This separation makes it easier to maintain and test your UI.

3. **Component Class:** The component class should contain the application's business logic, data processing, and interaction with services. It should be responsible for managing the state of the component and interacting with the template.
4. **Services:** Angular services are used to encapsulate data and logic that can be shared among multiple components. Services are responsible for making API requests, handling data, and managing state. Separating these concerns into services promotes reusability and maintainability.
5. **Routing:** The Angular Router allows you to define routes and route handlers for different parts of your application. By separating routing concerns, you can ensure that different components handle different views and responsibilities.
6. **Dependency Injection:** Angular's dependency injection system is a key feature that helps separate concerns by providing a way to inject services and other dependencies into components and services. This enables components to focus on their specific tasks without worrying about how to obtain their dependencies.
7. **Modules:** Angular modules help organize and encapsulate different parts of your application. By structuring your app into feature modules, you can further separate concerns and make it easier to manage and maintain.
8. **Pipes and Directives:** Angular provides pipes for transforming and formatting data in templates and directives for extending the behavior of HTML elements. These can be used to separate concerns related to data manipulation and custom UI behavior.
9. **State Management:** For more complex applications, you may consider using external state management libraries like NgRx or Akita to centralize and manage application state, further enhancing separation of concerns related to data management.

By following these practices and principles, you can achieve a high level of Separation of Responsibilities in your Angular application, making it more maintainable, testable, and scalable. This separation also helps developers work more efficiently and collaborate effectively on larger projects.

6.3 Adding Angular To The Environment :

To add Angular to your development environment, you'll need to follow a series of steps. Angular is a popular web application framework developed by Google that uses TypeScript. Here's a general guide on how to set up Angular in your environment:

1. **Prerequisites:** Before you start, make sure you have the following prerequisites installed on your system:

Node.js: Angular requires Node.js. You can download it from the official website:

<https://nodejs.org/>

npm (Node Package Manager): This comes with Node.js and is used to manage packages.

2. **Install Angular CLI:** The Angular CLI (Command Line Interface) is a powerful tool for creating and managing Angular projects. To install it, open your terminal or command prompt and run the following command :

Command : `npm install -g @angular/cli`

3. **Verify Installation:** After installation, you can verify that Angular CLI is installed correctly by running : **Command :** `ng -- version`

4. **Create a New Angular Project:** To create a new Angular project, use the following command :

Command : `ng new your-project-name`

5. **Navigate to Your Project Directory:** Change your working directory to the newly created project :

Command : `cd your-project-name`

6. **Serve the Application:** To run your Angular application locally, use the following command :

Command : `ng serve`

7. This will start a development server, and you can access your application in a web browser at “
<http://localhost:4200/>. “

8. **Code and Development:** You can now start building your Angular application. The code is typically located in the **src** folder. You can use your favorite code editor or Integrated Development Environment (IDE) to work on your project. You can configure your project by modifying the **angular.json** file, and you can install additional libraries and packages using npm.

Remember that this is a basic setup guide. Depending on your project's requirements, you may need to configure additional features like routing, forms, HTTP client, and state management. You can refer to the official Angular documentation for in-depth information and tutorials: <https://angular.io/docs>

That's it! You now have Angular set up in your development environment and can start building web applications using this framework.

6.4 Angular CLI :

The Angular CLI (Command Line Interface) is a powerful tool for developing and managing Angular applications. It provides a set of commands and utilities that streamline the development process and make it easier to create, build, test, and deploy Angular applications. Here are some key aspects of the Angular CLI:

1. **Project Generation:** You can use the Angular CLI to create a new Angular project with a basic directory structure and configuration files in a matter of seconds. You can create a new project by running the **ng new** command and specifying various options like project name, styling framework (e.g., CSS, SCSS), and more.
2. **Development Server:** The Angular CLI comes with a built-in development server. You can start the server using the **ng serve** command, which will compile your application and serve it locally. The development server automatically watches for changes in your source code and refreshes the application in your browser, making the development workflow very efficient.
3. **Code Generation:** The CLI provides commands to generate different parts of your Angular application, including components, services, modules, and more. For example, you can create a new component with the **ng generate component** command, and the CLI will generate the necessary files and update your application's module to include the new component.
4. **Build and Optimization:** When you are ready to deploy your Angular application, the CLI can help you build and optimize it for production. The **ng build** command generates a production-ready bundle of your application, minifying and optimizing the code for better performance.
5. **Testing:** The Angular CLI also includes commands for running unit tests and end-to-end (E2E) tests. You can use the **ng test** and **ng e2e** commands to execute your test suites using tools like Jasmine and Protractor.
6. **Configuration:** The CLI provides configuration files, such as **angular.json**, where you can define various settings for your project, including build options, development server settings, and third-party library configurations.
7. **Plugins and Schematics:** The Angular CLI is extensible, allowing you to add and use third-party libraries and tools easily. You can also create your own custom schematics to automate project-specific tasks.

8. **Integration with Angular Features:** The Angular CLI is designed to work seamlessly with the core features of Angular, like the Angular Router and Angular Forms. It can generate code that is compliant with Angular best practices and coding standards.
9. **Continuous Integration (CI) Support:** The CLI's build and test commands are often used in CI/CD pipelines to automate the testing and deployment of Angular applications.

Overall, the Angular CLI is a valuable tool for Angular developers, helping them to follow best practices, streamline the development process, and manage the project effectively. It reduces the overhead of manual setup and configuration, allowing developers to focus on writing code and delivering high-quality Angular applications.

6.5 Creating Basic Angular Application ;

Creating a basic Angular application involves several steps. Angular is a popular JavaScript framework for building web applications. Before you start, make sure you have Node.js and npm (Node Package Manager) installed on your computer. Here's a step-by-step guide to create a basic Angular application :

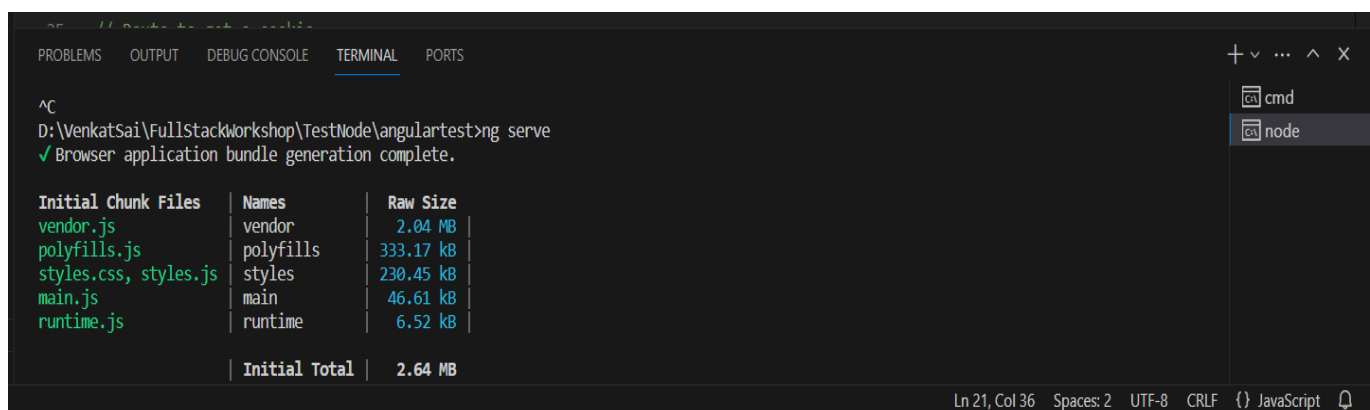
1. **Install Node.js and npm:** Make sure you have Node.js and npm (Node Package Manager) installed on your computer. You can download and install them from the [official website](#).
2. **Install Angular CLI:** Open your terminal or command prompt and run the following command to install the Angular CLI globally **Command :** " npm install -g @angular/cli ".
3. **Create a New Angular Project:** Create a new Angular project by running the following command : **Command :** " ng new my-angular-app ".
4. **Configure Project Options:** The Angular CLI will prompt you to configure project options, such as routing and stylesheet format. Choose the options that suit your project needs.
5. **Navigate to the Project Directory:** Change your working directory to the newly created Angular project : **Command :** " cd my-angular-app "
6. **Serve the Application:** Start a development server to see your application in action : **Command :** " ng serve "
7. The default URL for the development server is <http://localhost:4200>.
8. **Create Components:** Create new components using the Angular CLI. For example : **Command :** " ng generate component my-component "
9. **Edit HTML Templates and Styles:** Modify the HTML templates and styles of your components in their respective files located in the **src/app** directory.

10. **Define Routes (Optional)** : If you chose to include routing during project setup, define routes in the `src/app/app-routing.module.ts` file.
11. **Create Services (Optional)** : Create Angular services to handle data and business logic. You can use the Angular CLI to generate services. **Command** : “ `ng generate service my-service` ”.
12. **Add Data Binding**: Use Angular's data binding techniques like interpolation, property binding, and event binding to interact with your components.
13. **Implement Forms (Optional)**: If your application requires forms, use Angular's built-in forms module to create and handle forms.
14. **Handle HTTP Requests (Optional)**: Use Angular's **HttpClient** module to make HTTP requests to APIs or servers. You may need to create services for this purpose.
15. **Build for Production**: When your application is ready for production, build it using the following command **Command** : “ `ng build --prod` ”
16. **Deploy Your Application**: Deploy your production build to a web server or hosting platform of your choice to make it accessible to users.
17. **Learn and Explore**: Continue learning Angular by referring to the official [Angular documentation](#) and exploring tutorials, courses, and resources available online. Building more complex features and improving your application over time is a great way to gain experience.

That's it! You've created a basic Angular application. From here, you can continue to develop and expand your application by adding components, services, and other features to meet your project requirements.

After running the all the commands the output you will be see in the browser with the below url.

Outputs :



```
^C
D:\VenkatSai\FullStackWorkshop\TestNode\angulartest>ng serve
✓ Browser application bundle generation complete.
```

Initial Chunk Files	Names	Raw Size
vendor.js	vendor	2.04 MB
polyfills.js	polyfills	333.17 kB
styles.css, styles.js	styles	230.45 kB
main.js	main	46.61 kB
runtime.js	runtime	6.52 kB
Initial Total		2.64 MB

Ln 21, Col 36 Spaces: 2 UTF-8 CRLF {} JavaScript

Fig – 33 : Ng Serve

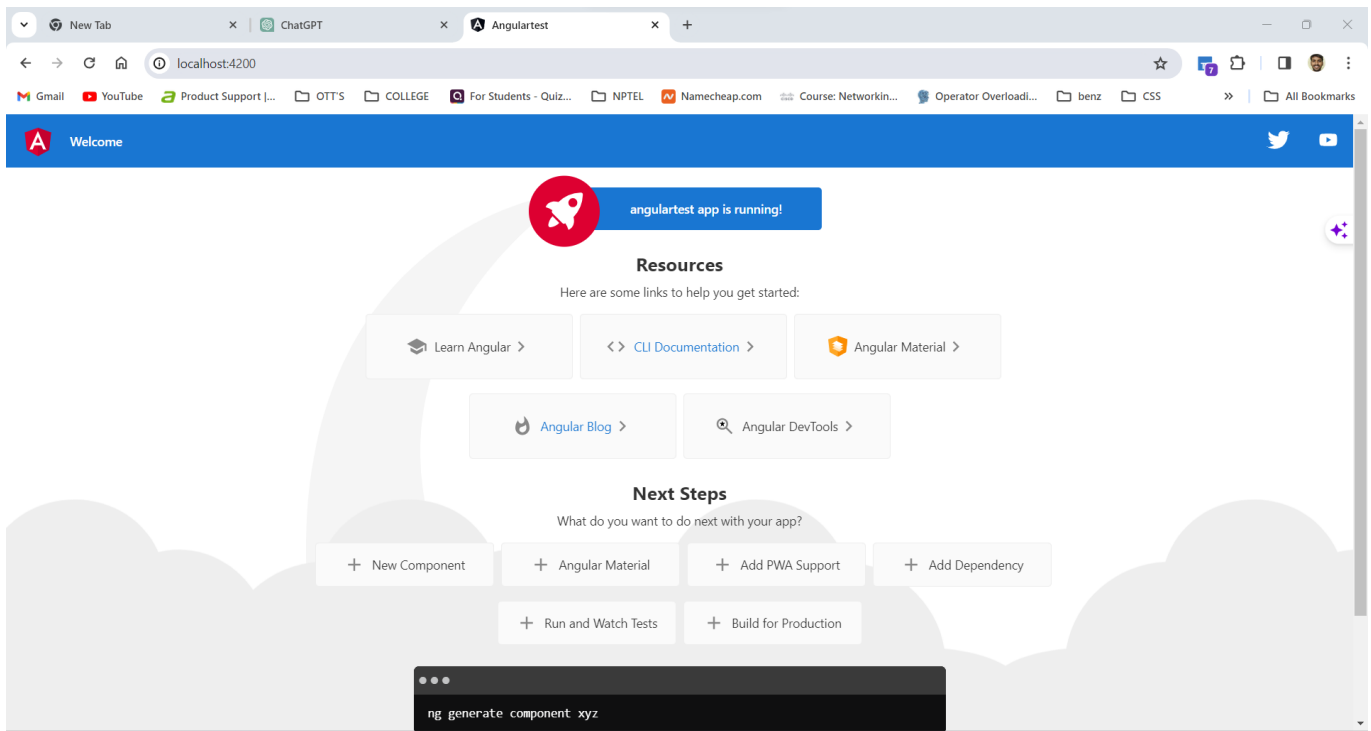


Fig – 34 : Angular

6.6 Angular Components :

Angular is a popular open-source JavaScript framework for building web applications. In Angular, components are one of the core building blocks of an application. They are responsible for encapsulating the user interface and its associated logic into reusable and self-contained units. Components are a fundamental part of the Angular architecture and are used to create dynamic, interactive, and modular web applications.

Here's an overview of Angular components :

1. **Component Structure:** A component in Angular typically consists of three main parts:
 - **Template:** This defines the structure of the component's view, often written in HTML with Angular-specific syntax for data binding and directives.
 - **Class:** The component class contains the logic, data, and methods that define the component's behavior. It is written in TypeScript.
 - **Metadata:** Angular uses decorators, such as **@Component**, to define metadata about the component, including its selector, template, and styles.

2. **Reusability:** Components are designed to be reusable, which means you can use the same component in different parts of your application or even in different projects. This promotes code modularity and maintainability.
3. **Data Binding:** Angular components support data binding, which allows you to bind data from the component's class to the template and vice versa. There are different types of data binding, including one-way binding (interpolation, property binding) and two-way binding, which makes it easy to keep the view and data in sync.
4. **Event Handling:** Components can respond to user interactions through event handling. You can define event handlers in the component class to respond to user actions, such as button clicks or form submissions.
5. **Lifecycle Hooks:** Angular components have a lifecycle, and you can hook into various stages of that lifecycle using lifecycle hooks. These hooks allow you to perform actions like initialization, cleanup, and more at specific points in a component's life.
6. **Dependency Injection:** Angular provides a powerful dependency injection system that allows you to inject services and other dependencies into your components. This promotes separation of concerns and makes it easier to test and maintain your code.
7. **Encapsulation:** Components are designed to encapsulate their functionality and state, which helps prevent conflicts and unintended interactions between different parts of your application.
8. **Hierarchical Structure:** Angular applications are typically organized in a hierarchical structure of components. You have a root component that contains child components, forming a tree-like structure. This structure is essential for building complex applications with clear separation of concerns.
9. **Routing:** Angular also offers a router module that allows you to navigate between different components and views within your application. You can configure routes and associate components with specific routes to create a multi-page application experience.

Angular components are the building blocks of an Angular application, encapsulating the user interface and behavior of different parts of the application. They promote modularity, reusability, and maintainability, making it easier to develop large and complex web applications.

6.7 Angular Expressions :

Angular Expressions are a key aspect of the AngularJS framework, which was popular for building dynamic web applications but has been largely replaced by Angular (also known as Angular 2+). Angular Expressions are used to bind data to HTML elements and templates, allowing you to dynamically display and manipulate data in your web application.

Here's a breakdown of key points about Angular Expressions :

1. **Interpolation:** Angular Expressions are primarily used for data binding. They allow you to interpolate values into the HTML template, meaning you can embed dynamic content within the HTML. This is done by using double curly braces `{{ }}` in the HTML markup. For example, if you have a variable named **name** in your AngularJS controller, you can display its value in the HTML template like this :

Example : “ `<p>Hello, {{ name }}!</p>` “

2. **Expression Syntax:** Angular Expressions support a wide range of JavaScript expressions. You can use variables, literals, operators, and function calls within the double curly braces to create complex expressions. For example, you can do arithmetic operations, string concatenation, and even call functions in your expressions.

Example : “ `<p>Total: {{ price * quantity | currency }}</p>` “

3. In the above example, **price** and **quantity** are variables, and the **currency** filter is applied to format the result as a currency.
4. **Filters:** Angular Expressions can be enhanced with filters. Filters allow you to format, transform, or filter the data before it's displayed in the template. In the example above, the **currency** filter formats the result as a currency.
5. **Context:** Expressions have access to the scope, which is a JavaScript object that contains the model data. The scope is managed by Angular and is the bridge between your JavaScript controller and the HTML template. The expressions can access properties and functions from the scope.
6. **Security:** Angular Expressions are designed to be safe and prevent code execution. They are a subset of JavaScript and do not allow any potentially harmful code. This helps protect against cross-site scripting (XSS) attacks.
7. **Two-Way Data Binding:** While Angular Expressions are often used for one-way data binding (displaying data), Angular also supports two-way data binding. This means that changes in the HTML can update the model, and changes in the model can update the HTML. This is typically done using directives like **ng-model** for form elements. One of the key strengths of Angular Expressions in AngularJS is their ability to establish two-way data binding. Two-way data binding means that

changes in the model are automatically reflected in the view, and changes in the view are propagated back to the model. Angular achieves this through the **ng-model** directive, which is commonly used with form elements like input fields and checkboxes. For example, if you have an input field in your HTML template like this :

Example : “ `<input type="text" ng-model="userName">` “

When a user types into the input field, the **userName** variable in your controller is automatically updated to match the input value. Conversely, if you change the value of **userName** in your controller, the input field in the view will be updated.

8. **Conditional Rendering:** Angular Expressions can be used to conditionally display or hide elements in your HTML. You can use Angular's **ng-if**, **ng-show**, and **ng-hide** directives to control the visibility of elements based on the evaluation of an expression. For example, you can conditionally show a message only if a certain condition is met.

Example : “ `<div ng-if="showMessage">This is a message.</div>` “

In this case, the message will be displayed if the **showMessage** variable in your controller is truthy.

9. **Iteration and Loops:** Angular Expressions also allow for iterating over arrays and collections. The **ng-repeat** directive can be used to create repetitive HTML elements based on the content of an array in your controller. **Example :** “ ` <li ng-repeat="item in items">{{ item.name }} ` “
This code will generate a list with an item for each object in the **items** array in your controller, displaying the **name** property of each object.

10. **Event Handling:** Angular Expressions can be used to handle user events such as clicks, form submissions, and more. Angular provides directives like **ng-click**, **ng-submit**, and **ng-change** that allow you to specify functions or expressions to run when an event occurs.

Example : “ `<button ng-click="doSomething()">Click me</button>` “

In this example, when the button is clicked, the **doSomething** function in your controller will be executed.

Angular Expressions in AngularJS play a pivotal role in creating dynamic, interactive, and data-driven web applications. They provide a powerful way to connect the model and the view, allowing for responsive and real-time updates without requiring manual DOM manipulation. While AngularJS has been succeeded by Angular (Angular 2+), many of the concepts related to expressions and data binding have been retained and enhanced in the newer versions of the framework.

6.8 Angular Bindings :

Angular is a popular JavaScript framework for building web applications, and it includes a feature called data binding, which allows you to establish a connection between your application's data and the user interface. Angular provides several types of data binding, including :

1. **Interpolation ({{ expression }}):** Interpolation is a one-way data binding technique that allows you to display data from your component's TypeScript code in your HTML templates. You enclose an expression within double curly braces, and Angular will replace it with the evaluated value.

Example : “ `<p>{{ message }}</p>` ” In this example, if you have a property named **message** in your component, its value will be displayed in the **<p>** element.

2. **Property Binding ([property]="expression"):** Property binding allows you to set the value of an HTML element's property (e.g., **src**, **value**, **disabled**) based on the component's data. It is one-way binding from the component to the DOM. For example, you can set an image's source attribute dynamically .

Example : “ `` ” In your component, you can change the **imageUrl** property to update the image source.

3. **Event Binding ((event)="expression"):** Event binding allows you to listen for and respond to events triggered by HTML elements, such as button clicks or input field changes. You bind a method from your component to an event in your template. For example, to handle a button click event.

Example : “ `<button (click)="onButtonClick()">Click me</button>` ”

In your component, you define the **onButtonClick()** method to handle the click event.

4. **Two-Way Data Binding ([ngModel]):** Two-way data binding allows you to synchronize data between the component and the view in both directions. It's often used with form elements like input fields and checkboxes. You use **[[ngModel]]** to bind a property to an input element and automatically update the view when the property changes and update the property when the user interacts with the input.

Example : “ `<input [(ngModel)]="name">` ”

In this example, changes to the input element update the **name** property in the component, and changes to the **name** property in the component update the input's value.

Angular's data binding simplifies the development of dynamic and responsive web applications by providing a clear and structured way to manage the interaction between your application's data and the user interface. Depending on your requirements, you can choose the appropriate type of binding to achieve the desired behaviour in your Angular application.

6.9 Directives – Structural and Attributes :

In Angular, directives are a fundamental building block of the framework that allows you to extend the functionality of HTML elements. There are two main types of directives: structural directives and attribute directives.

Structural Directives: Structural directives are responsible for manipulating the DOM's structure by adding, removing, or replacing elements. They are typically prefixed with an asterisk (*) in the HTML template. Some common structural directives in Angular are:

- ***ngIf:** The *ngIf directive conditionally adds or removes elements from the DOM based on a provided expression. If the expression evaluates to **true**, the element is added; if it evaluates to **false**, the element is removed.

Example : “ `<div *ngIf="showElement">This element will be shown or hidden</div>` “

- ***ngFor:** The *ngFor directive is used for iterating over arrays and collections, rendering elements for each item in the collection. Example :

```
<ul>
```

```
<li *ngFor="let item of items">{{ item }}</li>
```

```
</ul>
```

- ***ngSwitch:** The *ngSwitch directive is used to conditionally render elements based on multiple values. Example :

```
<div [ngSwitch]="value">
```

```
<div *ngSwitchCase="'A'">Render for A</div>
```

```
<div *ngSwitchCase="'B'">Render for B</div>
```

```
<div *ngSwitchDefault>Render for everything else</div>
```

```
</div>
```

Attribute Directives: Attribute directives are used to change the appearance or behavior of an element. They are applied to elements as attributes and do not alter the DOM structure. Angular provides some built-in attribute directives, and you can also create your custom attribute directives. Some common built-in attribute directives are:

- **ngClass:** The ngClass directive allows you to conditionally apply one or more CSS classes to an element based on a given expression.

Example : “ `<div [ngClass]='{'active': isActive, 'inactive': !isActive}'>This element's class depends on the value of isActive</div>` “

- **ngStyle:** The ngStyle directive enables you to set inline CSS styles for an element based on a provided expression.

Example : “ `<div [ngStyle]='{'color': textColor, 'font-size': fontSize + 'px'}'>This element's style depends on textColor and fontSize</div>` “

- **ngModel:** The ngModel directive is used for two-way data binding in forms, allowing you to bind the value of form controls to a component property.

Example : “ `<input [(ngModel)]="username" />` “

You can also create custom attribute directives to encapsulate and reuse specific behaviour in your Angular application.

Both structural and attribute directives play a crucial role in enhancing the dynamic and interactive nature of Angular applications by enabling you to modify the DOM and manage the behaviour and appearance of elements.

6.10 Events and Change Detection :

In Angular, events and change detection are essential concepts that enable the framework to react to user interactions and update the user interface when the application's data changes. Let's break down these concepts :

Events in Angular: Events are user interactions or system-generated triggers that can be captured and handled in an Angular application. Common examples of events include user clicks, keyboard input, form submissions, and HTTP requests. Angular provides a mechanism to listen to and respond to these events. To work with events in Angular, you typically use event binding. Event binding allows you to associate a method or expression in your component class with an event in the HTML template. When the event occurs, the associated method is executed.

Example of event binding in Angular:

`<button (click)="handleButtonClick()">Click me</button>`

In the above code, when the button is clicked, the **handleButtonClick** method in the component class will be executed.

Change Detection in Angular: Change detection is the process by which Angular detects changes in the application's data model and updates the view accordingly. Angular's change detection system is automatic and efficient, ensuring that the UI stays in sync with the underlying data without manual intervention.

Angular uses a unidirectional data flow, where data flows from the component class to the template. When a component's properties change, Angular automatically triggers the change detection process to update the view.

Change detection can be triggered by various events, including:

- User interactions (e.g., clicking a button that updates a variable).
- Asynchronous operations (e.g., HTTP requests or timers).
- Angular lifecycle hooks (e.g., **ngOnInit**, **ngOnChanges**).

Developers usually don't need to manage change detection explicitly, as Angular takes care of it. However, you can optimize change detection performance by following best practices like using the **OnPush** change detection strategy, minimizing the number of bindings and using the **async** pipe for asynchronous data.

The **ngZone** service in Angular is used to manage and control the change detection process. It allows you to explicitly run code inside or outside the Angular zone, affecting when and how change detection is triggered.

Example of data binding and automatic change detection :

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-example',
  template: `
    <p>{{ message }}</p>
    <button (click)="updateMessage()">Update Message</button>
  `,
})
```



```

}))

export class ExampleComponent {

  message = 'Hello, Angular!';

  updateMessage() {

    this.message = 'New message';

  }

}

```

In the code above, when the button is clicked, the **updateMessage** method changes the value of the **message** property, and Angular's change detection automatically updates the displayed message in the view.

Understanding events and change detection is crucial for building dynamic and responsive Angular applications, as they enable the application to react to user input and keep the UI synchronized with the data model.

6.11 Using Observables :

In Angular, observables are a key part of the framework and are commonly used to handle asynchronous operations, such as handling user input, making HTTP requests, and managing state changes. Observables are part of the RxJS library, which is included with Angular by default. Observables allow you to work with asynchronous data streams and provide a way to subscribe to and react to changes in those streams. Here's how you can use observables in Angular:

1. **Import RxJS:** First, make sure to import RxJS operators and other necessary functions. You can do this by importing them in your Angular component or service file. For

Code :

```

import { Observable, of } from 'rxjs';
import { map, catchError } from 'rxjs/operators';

```

2. **Create an Observable:** You can create an observable using the Observable class provided by RxJS. Here's a simple example of creating an observable that emits values over time :

Code :

```

const myObservable = new Observable<number>((observer) => {

```

```
observer.next(1);
observer.next(2);
observer.next(3);
observer.complete();
});
```

3. **Subscribe to the Observable:** To listen for values emitted by an observable, you need to subscribe to it. You can do this in an Angular component.

Code :

```
myObservable.subscribe(
  (value) => {
    console.log('Received value: ', value);
  },
  (error) => {
    console.error('Error:', error);
  },
  () => {
    console.log('Observable completed. ');
  }
);
```

In the code above, you pass three functions to the **subscribe** method: one for handling emitted values, one for handling errors, and one for handling the completion of the observable.

4. **Use Observable Operators:** RxJS provides a wide range of operators that you can use to transform, filter, or manipulate the data emitted by an observable. You can chain these operators together to create complex data flows.

Code :

```
myObservable
  .pipe(
    map((value) => value * 2),
    filter((value) => value > 2)
  )
  .subscribe((value) => {
    console.log('Transformed value: ', value);
  });
```

In this code, we use the **map** and **filter** operators to transform and filter the emitted values.

5. **Handling HTTP Requests with Observables:** In Angular, you often use observables to make HTTP requests to a server. The Angular **HttpClient** module returns observables for handling HTTP responses. Here's an example of making an HTTP GET request.

Code :

```
import { HttpClient } from '@angular/common/http';  
  
// Inject the HttpClient service into your component or service  
constructor(private http: HttpClient) {}  
  
fetchData() {  
  this.http.get('https://api.example.com/data')  
    .subscribe((data) => {  
      console.log('Received data:', data);  
    });  
}
```

6. **Clean Up: Unsubscribe Observables** It's essential to unsubscribe from observables when they are no longer needed to prevent memory leaks. You can unsubscribe in the component's **OnDestroy** lifecycle hook.

Code :

```
import { Component, OnDestroy } from '@angular/core';  
import { Subscription } from 'rxjs';  
  
export class MyComponent implements OnDestroy {  
  private subscription: Subscription;  
  
  constructor() {  
    this.subscription = myObservable.subscribe((value) => {  
      console.log('Received value: ', value);  
    });  
  }  
  
  ngOnDestroy() {  
    this.subscription.unsubscribe();  
  }  
}
```

Observables are a powerful way to handle asynchronous data in Angular. They are used extensively in many aspects of Angular development, such as handling user input, managing component state, and making HTTP requests. Understanding how to use and work with observables is essential for building robust and responsive Angular applications.

CHAPTER – 7 : Mongo DB and Node JS

7.1 Adding Mongo DB Driver to Node JS :

MongoDB is a popular NoSQL database that is designed for storing and managing large amounts of data. It's known for its flexibility and scalability, making it a great choice for a wide range of applications. To use MongoDB with Node.js, you need to install the MongoDB Node.js driver, which allows your Node.js application to communicate with the MongoDB database.

Here are the steps involved in adding the MongoDB driver to your Node.js project :

1. **Initialize Your Node.js Project:** If you don't already have a Node.js project, you should start by creating one. You can do this by running the following command in your terminal “ npm init -y “
2. **Install the MongoDB Node.js Driver:** You can install the MongoDB Node.js driver using npm, which is the package manager for Node.js. Run the following command in your project's directory.

Command : “ npm install mongodb “

3. **Import the MongoDB Driver:** In your Node.js code, you'll need to require the MongoDB driver to use it. Here's an example of how to import the driver.

Code : “ const { MongoClient } = require('mongodb'); “

4. **Connect to Your MongoDB Database:** You need to establish a connection to your MongoDB server to perform any database operations. You'll typically provide a connection URL with your MongoDB server's information. Here's an example of how to connect.

Code :

```
Const url =
```

```
'mongodb+srv://saidoc45:saidoc45@testmongo.nqtudyw.mongodb.net/?retryWrites=true&w=majority
```

```
'; // Replace with your MongoDB server URL
```

```
const client = new MongoClient(url);
```

```
async function connectToDatabase() {try {
```

```
await client.connect();
```

```
console.log('Connected to MongoDB');
```

```
} catch (err) {console.error('Error connecting to MongoDB:', err); }} connectToDatabase();
```

5. **Perform Database Operations:** With the MongoDB driver connected to your database, you can now perform various database operations, such as inserting, querying, updating, and deleting documents. Here's a simple example of inserting a document into a collection.

Code :

```
const db = client.db('Collection1'); // Replace with your database name
const collection = db.collection('mycollection'); // Replace with your collection name
const document = { name: 'John', age: 30 };
const result = await collection.insertOne(document);
console.log('Inserted document:', result.ops[0]);
```

6. **Close the Connection:** It's essential to close the MongoDB connection when you're done using it to release resources. You can do this with the **client.close()** method : “ client.close() “.

These are the basic steps involved in adding the MongoDB driver to a Node.js project and performing database operations. Keep in mind that you should handle errors and implement appropriate error handling in your code for real-world applications.

7.2 Connecting To Mongo DB From Node JS :

To connect to a MongoDB database from a Node.js application, you'll need to follow a series of steps. MongoDB is a NoSQL database, and you can use the official MongoDB Node.js driver to interact with it. Here's a step-by-step guide on how to connect to MongoDB from a Node.js application:

1. **Create a Node.js Project:**

Create a new directory for your Node.js project if you haven't already.

Open a terminal or command prompt in the project directory.

2. **Install the MongoDB Node.js Driver:** You can install the MongoDB Node.js driver using npm, which is the package manager for Node.js. Run the following command in your project's directory.

Command : “ npm install mongodb “

3. **Import the MongoDB Driver:** In your Node.js code, you'll need to require the MongoDB driver to use it. Here's an example of how to import the driver.

Code : “ const { MongoClient } = require('mongodb'); “

4. **Connect to the MongoDB Server:** Define a connection string that includes the URL of your MongoDB server. The connection string should have the following format :

- a. **Format :** mongodb://username:password@host:port/database

Use the **MongoClient** to connect to the database using the connection string :

- b. **Code :**

```
const url = 'mongodb://localhost:27017/mydb'; // Replace with your connection string
```

```
MongoClient.connect(url, { useNewUrlParser: true, useUnifiedTopology: true }, (err, client)
=> {
  if (err) {
    console.error('Error connecting to MongoDB:', err);
    return; }
  client.close();
});
```

5. **Perform Database Operations:** Inside the connection callback, you can perform various database operations using the **client** object. For example, you can insert, update, delete, or query data.
6. **Close the Connection:** It's essential to close the MongoDB connection when you're done with your database operations to release resources. This is done by calling **client.close()** as shown in the example.

This is a basic example of connecting to a MongoDB database from a Node.js application. You can expand on this foundation to build more complex database interactions for your specific use case.

7.3 Understanding The Objects :

MongoDB is a NoSQL database that stores data in a flexible, document-oriented format called BSON (Binary JSON). Understanding the key objects in MongoDB is essential for effective data management:

1. **Database:** The top-level container for data in MongoDB. Each database is a separate logical entity, allowing you to group related data.
2. **Collection:** A collection is a group of documents within a database. It's roughly equivalent to a table in a relational database.
3. **Document:** Documents are the fundamental unit of data in MongoDB. They are JSON-like objects, composed of key-value pairs, and are stored in collections. Documents are flexible and can have varying structures.
4. **Field:** A field is a key-value pair within a document, representing a specific piece of data. Fields can be of various data types.
5. **Index:** MongoDB uses indexes to improve query performance. They allow for efficient data retrieval by creating a data structure that speeds up searches.

6. **Cursor:** A cursor is a pointer to the result set of a query. It allows you to iterate through documents in a collection, making it useful for paging and processing large datasets.
7. **Aggregation Pipeline:** A powerful feature for transforming and processing data in MongoDB. It allows you to create complex data transformations using a series of stages.
8. **Query:** MongoDB provides a rich set of query operators to filter and retrieve specific documents from a collection. Queries can be simple or complex, supporting various conditions.
9. **Update:** You can modify documents in a collection using update operations. Updates can replace entire documents or specific fields.
10. **Insert:** To add new data to a collection, you use the insert operation. You can insert single documents or multiple documents at once.
11. **Delete:** Documents can be removed from a collection using the delete operation. You can specify criteria to target specific documents.
12. **Write Concern:** This controls the acknowledgment level for write operations, ensuring data durability based on your needs.
13. **Primary Key (_id):** Each document has a unique identifier, the _id field, which is automatically generated if not provided. It's crucial for document retrieval and updates.
14. **Replica Set:** MongoDB supports high availability and data redundancy through replica sets, which consist of multiple database servers with synchronized data.
15. **Sharding:** For horizontal scaling, sharding distributes data across multiple servers to handle large datasets and high traffic.
16. **GridFS:** GridFS is a specification for storing large files in MongoDB. It allows you to split large files into smaller chunks for efficient storage and retrieval.
17. **Geospatial Data:** MongoDB supports geospatial indexes and queries, making it suitable for location-based applications.
18. **Text Search:** Full-text search is possible in MongoDB, allowing for efficient querying of textual content.
19. **Authentication and Authorization:** MongoDB provides security mechanisms to control access to databases, collections, and documents.

20. **Transactions:** MongoDB supports multi-document transactions, ensuring data consistency in complex operations.

Understanding these fundamental objects in MongoDB is essential for designing efficient database schemas, optimizing queries, and building robust applications that harness the power of this versatile NoSQL database system.

7.4 Accessing and Manipulating The Database :

Accessing and manipulating a MongoDB database involves interacting with the database using a programming language, such as JavaScript (Node.js), Python, Java, or others. MongoDB is a NoSQL database that stores data in JSON-like BSON (Binary JSON) documents within collections. Here's a basic overview of how to access and manipulate a MongoDB database:

1. Install MongoDB on your system and start the MongoDB server.
2. Use a MongoDB client, such as the MongoDB shell or a programming language-specific driver (e.g., pymongo for Python).
3. Connect to the MongoDB server using the appropriate connection string or configuration.
4. Create a collection or use an existing one: **db.createCollection("mycollection")** (MongoDB shell) or **collection = db["mycollection"]** (Python).
5. Insert documents into the collection: **db.mycollection.insertOne({ name: "John", age: 30 })** (MongoDB shell) or **collection.insert_one({"name": "John", "age": 30})** (Python).
6. Query data using the **find** method: **db.mycollection.find()** (MongoDB shell) or **collection.find({"name": "John"})** (Python).
7. Update documents with the **update** or **updateOne** method: **db.mycollection.updateOne({ name: "John" }, { \$set: { age: 31 } })** (MongoDB shell) or **collection.update_one({"name": "John"}, {"\$set": {"age": 31}})** (Python).
8. Delete documents with the **deleteOne** or **deleteMany** method: **db.mycollection.deleteOne({ name: "John" })** (MongoDB shell) or **collection.delete_one({"name": "John"})** (Python).
9. Create indexes for efficient querying: **db.mycollection.createIndex({ name: 1 })** (MongoDB shell) or **collection.create_index([("name", 1)])** (Python).

10. Aggregate data using the **aggregate** method: **db.mycollection.aggregate([...])** (MongoDB shell) or **collection.aggregate([...])** (Python).
11. Monitor the database using the **db.stats()** (MongoDB shell) or administrative commands.
12. Secure your MongoDB instance with authentication and authorization.
13. Backup and restore data using MongoDB tools or services.
14. Disconnect from the MongoDB server when you're done: **db.logout()** (MongoDB shell) or close the connection in your programming language.

These steps provide a basic outline for working with MongoDB, but the specific commands and methods may vary depending on your use case and programming language.

```
1  const { MongoClient } = require('mongodb');
2  const uri = 'mongodb+srv://saidoc45:saidoc45@testmongo.nqtudyw.mongodb.net/Collection1?retryWrites=true&w=majority';
3  async function connectToMongoDB() {
4      const client = new MongoClient(uri, { useNewUrlParser: true, useUnifiedTopology: true }); try {
5          await client.connect();
6          console.log('Connected to MongoDB');
7          const db = client.db();
8          const collection = db.collection('mycollection');
9          const document1 = { name: "John Doe", age: 30 };
10         const document2 = { name: "Prabhas", age: 50 };
11         const result = await collection.insertOne(document1);
12         const result1 = await collection.insertOne(document2);
13         console.log("Inserted document with ID: " + result.insertedId);
14         const query = { name: "John Doe" };
15         const update = { $set: { age: 31 } };
16         const result2 = await collection.updateOne(query, update);
17         console.log("Updated " + result2.modifiedCount + " document(s)");
18         const query1 = { name: "John Doe" };
19         const result3 = await collection.deleteOne(query1);
20         console.log("Deleted " + result3.deletedCount + " document(s)");
21         const cursor = collection.find();
22         const documents = await cursor.toArray();
23         documents.forEach(document => { console.log(document)});
24     } catch (err) { console.error('Error connecting to MongoDB:', err); } finally { await client.close(); } }
25 connectToMongoDB();
26
```

Fig – 34 : Mongo DB

7.5 Introducing The DataSet :

Understanding a dataset in the context of MongoDB involves a comprehensive exploration of the data stored within the NoSQL database. This process typically begins with establishing a connection to the MongoDB server and selecting the appropriate database. To gain insight into the dataset, one must examine the collections within the database, which serve as containers for related data. Queries are then employed to retrieve specific documents from these collections, allowing for the examination of individual data points. It's essential to assess the structure of the data, including field names and data types, as well as any existing indexes that optimize query performance. Further analysis may include data aggregation,

examining field values for distribution and uniqueness, estimating data size and storage distribution, and understanding document relationships, whether through references or embedded documents. This thorough exploration is critical for effectively working with MongoDB datasets, enabling informed decision-making and the creation of efficient applications and systems.

1. **Connect to MongoDB:** Use a MongoDB client to establish a connection to your database.
2. **Select the Database:** Choose the specific database where your dataset is stored using the **use** command.
3. **List Collections:** List the collections in your database using the **show collections** command to see what data is available.
4. **Query Documents:** Use the **find** method to query documents within a collection to examine the data.
5. **View Data Structure:** Explore the data's structure, including fields and data types, to understand its format.
6. **Index Information:** Check for any existing indexes on the collection to optimize query performance.
7. **Aggregate Data:** Use the aggregation framework to perform complex data transformations and analysis.
8. **Analyse Field Values:** Understand the values of specific fields, including their distribution and uniqueness.
9. **Data Size and Storage:** Estimate the size of the dataset and how it's distributed across shards or servers.
10. **Document Relationships:** Investigate relationships between collections and how data is related, possibly using references or embedded documents.

By following these steps, you can gain a solid understanding of your dataset in MongoDB, enabling you to work with it effectively.

7.6 Understanding The Query Objects and Options :

In MongoDB, the **find** method is used to query data from a collection. When you use the **find** method, you can specify a query object and optionally a query options object to filter and customize the results. Let's take a closer look at both the query object and query options object:

Query Object:

The query object is used to specify the criteria for selecting documents from a collection. It defines which documents should be returned based on their field values. The query object is passed as an argument to the **find** method.

The query object is a JSON-like document that contains key-value pairs. The keys represent field names, and the values represent the criteria that documents must meet to be included in the result set.

For example, if you have a collection of documents representing people and you want to find all documents where the "age" field is greater than or equal to 30, your query object might look like this : “ { "age": { \$gte: 30 } } ” and Here, the **\$gte** operator stands for "greater than or equal to."

Query Options Object :

The query options object is an optional argument that allows you to customize the behavior of the find operation. It's typically used to specify options like sorting, limiting the number of results, and skipping documents. The query options object is also a JSON-like document that can contain various options. Some commonly used options include:

Sort : Specifies the sorting order of the results. For example, to sort documents by the "name" field in ascending order, you would use the following query options object.

Code : “ { "name": 1 } ” and Here, **1** represents ascending order, and **-1** would represent descending order.

limit: Limits the number of documents returned in the result set. For instance, to retrieve only the first 10 documents, you can use : “ { "limit": 10 } ”

skip: Allows you to skip a specific number of documents from the beginning of the result set. For example, to skip the first 5 documents, you can use : “ { "skip": 5 } ”

These are just a few examples of query options, and there are more available for various purposes, such as projection, hinting, and collation.

Here's an example of using both a query object and a query options object with the **find** method in MongoDB :

Code : “ db.collection('people').find({ "age": { \$gte: 30 } }).sort({ "name": 1 }).limit(10).skip(5) “

This query will retrieve documents from the "people" collection where the "age" is greater than or equal to 30, sort them by the "name" field in ascending order, limit the result set to 10 documents, and skip the first 5 documents that match the criteria.

7.7 Finding Specific Sets Of Documents :

To find specific sets of documents in a MongoDB collection, you can use the **find()** method with a query that specifies the criteria for selecting documents. MongoDB's query language is quite flexible and allows you to filter documents based on various conditions. Here's how you can perform such queries :

1. **Basic Query:** To find documents that match a specific field with a certain value, you can use the basic query format: “ db.collection_name.find({ field_name: value }) “
2. **Query Operators:** You can use query operators to specify more complex conditions. For example:

Greater than (**\$gt**) or less than (**\$lt**) a value : “ db.sales.find({ amount: { \$gt: 1000 } }) “

Logical **AND** and **OR** conditions : “ db.orders.find({ \$or: [{ status: "shipped" }, { status: "processing" }] }) “
3. **Nested Fields :** If your documents have nested fields, you can query them using dot notation. For example, to find all books published in a specific year : “ db.books.find({ "publication.year": 2020 }) “
4. **Array Fields :** If you want to find documents with specific elements in an array, you can use array operators. For example, to find all users who like a particular genre : “ db.users.find({ likedGenres: "Science Fiction" }) “
5. **Projection :** You can specify which fields you want to retrieve using projection. This can help you limit the amount of data returned in the results : “ db.products.find({ category: "Electronics" }, { name: 1, price: 1, _id: 0 }) “
6. **Sorting:** You can also sort the results using the **sort()** method : “ db.orders.find().sort({ orderDate: -1 }) “

This is just a basic overview of querying in MongoDB. MongoDB's query language is powerful and allows for complex queries. Be sure to refer to the official MongoDB documentation for more details and specific use cases: [MongoDB Query Operators](#).

7.8 Counting Documents :

Counting documents in MongoDB is a fundamental operation in database management. MongoDB, a NoSQL database, stores data in collections, and knowing the number of documents within a collection is crucial for various purposes, including data analysis, monitoring, and maintaining data integrity. To achieve this, developers can use the official MongoDB drivers for their programming language of choice, such as Node.js, to connect to the database and perform a `countDocuments` operation. This operation is efficient, and it provides an accurate count of the documents in a collection, enabling users to make informed decisions and gain insights into their data. Whether for reporting or real-time analytics, counting documents in MongoDB is a simple yet essential task for effective database management.

1. Create a JavaScript file (e.g., **countDocuments.js**) for your code.
2. Use the following code to count documents in a MongoDB collection :



```
1  const { MongoClient } = require('mongodb');
2  const url = 'mongodb+srv://saidoc45:saidoc45@testmongo.nqtudyw.mongodb.net/Collection1?retryWrites=true&w=majority';
3  const dbName = 'Collection1';
4  const collectionName = 'mycollection';
5  async function countDocuments() { try {
6      const client = new MongoClient(url, { useUnifiedTopology: true });
7      await client.connect();
8      const db = client.db(dbName);
9      const collection = db.collection(collectionName);
10     const count = await collection.countDocuments();
11     console.log(`Total documents in ${collectionName}: ${count}`);
12     await client.close();
13 } catch (error) { console.error('Error:', error); } }
14 countDocuments();
```

Fig – 35 : Counting

This code will connect to your MongoDB instance, count the documents in the specified collection, and then print the count to the console.

7.9 Limiting and Sorting Result Sets :

In MongoDB, limiting and sorting result sets are crucial for efficiently managing and extracting data. Sorting allows you to arrange documents in ascending or descending order based on specified fields, making it easier to retrieve data in a meaningful way. Limiting, on the other hand, restricts the number of documents returned in a query result, which is particularly useful when dealing with large datasets to improve performance and reduce unnecessary data transfer. These capabilities are essential for tasks such as displaying paginated data, implementing search functionality, or simply organizing data for analysis. MongoDB provides various methods and options to achieve these goals, enabling developers to fine-tune their queries and optimize database performance.

1. **Install and Set Up MongoDB:** Make sure you have MongoDB installed and running on your system.
2. **Create a Collection:** You'll need a collection to work with. If it doesn't exist, you can create it by inserting some documents. For example :

```
db.myCollection.insertMany([  
  { name: "John", age: 30 },  
  { name: "Jane", age: 25 },  
  { name: "Bob", age: 35 }  
])
```
3. **Find Documents:** To retrieve documents from the collection, you can use the **find()** method. For instance : “ `db.myCollection.find({})` ”
4. **Sort Documents:** You can sort the result set by using the **sort()** method. For example, to sort documents by the "name" field in ascending order : “ `db.myCollection.find({}).sort({ name: 1 })` ”
5. **Limit Results:** If you want to limit the number of results, you can use the **limit()** method. For instance, to limit to 2 results : “ `db.myCollection.find({}).limit(2)` ”
6. **Skip Results:** To skip a specific number of results, you can use the **skip()** method. For example, to skip the first 2 results and then limit to 2 : “ `db.myCollection.find({}).skip(2).limit(2)` ”
7. **Combining Sort, Limit, and Skip:** You can combine these methods to sort, limit, and skip results as needed.
8. **Create an Index:** If you frequently sort on a specific field, consider creating an index for that field to improve query performance. To create an index on the "age" field :
`db.myCollection.createIndex({ age: 1 })` “
9. **Explain the Query:** To understand the query execution plan and performance, you can use the **explain()** method. For example : “ `db.myCollection.find({}).sort({ name: 1 }).explain("executionStats")` ”
10. **Analyse the Query Plan:** Review the output from **explain()** to optimize your query.
11. **Use Query Operators:** MongoDB provides various query operators to filter documents effectively. For example, you can use **\$gt** (greater than) and **\$lt** (less than) for numerical fields.
12. **Use the Aggregation Framework:** For complex sorting and limiting operations, consider using the aggregation framework, which provides more advanced capabilities.
13. **Consider Sharding:** If you have a large dataset, you might need to consider sharding to distribute your data across multiple servers for better scalability.
14. **Test and Optimize:** Finally, test your queries with realistic data and workload, and optimize them as needed to achieve the desired performance.

These steps should help you perform sorting and limiting of result sets in MongoDB effectively, whether for small or large datasets.

REFERENCES

1. <https://nodejs.org/en/learn>
2. <https://angular.io/docs>
3. <https://www.mongodb.com/docs/>
4. <https://www.typescriptlang.org/docs/>
5. <https://expressjs.com/en/starter/installing.html>
6. <https://www.w3schools.com/angular/>
7. <https://www.freecodecamp.org/news/learn-angular-full-course/>
8. <https://www.simplilearn.com/tutorials/angular-tutorial>
9. <https://www.tektutorialshub.com/angular-tutorial/>
10. <https://www.tutorialsteacher.com/angular>
11. <https://www.w3schools.com/typescript/>
12. <https://www.freecodecamp.org/news/learn-typescript-beginners-guide/>
13. <https://www.codecademy.com/learn/learn-typescript>
14. <https://www.tutorialsteacher.com/typescript>
15. <https://learn.microsoft.com/en-us/training/modules/typescript-get-started/>