SRINIVAS UNIVERSITY

COLLEGE OF COMPUTER & INFORMATION SCIENCE

CITY CAMPUS, PANDESHWAR, MANGALORE - 575001

BACKGROUND STUDY MATERIAL

Web Application Framework

III SEMESTER M.C.A

SRINIVAS UNIVERSITY

SAMAGRA GNANA

ESTD: 1988

Compiled by

Faculty

Paper Code: 22MCA31 Theory/Week: 4 Hours Credits: 4	WEB APPLICATION FRAMEWORK	Hours: 40 IA : 50 Exam: 50

Course Objectives:

- Understand Node JS and REPL terminal.
- Experiment with Node JS Modules and Node Package Manager.
- Develop applications to handle events in Node JS
- Make use of Web Server to manage database.
- Demonstrate Express Framework

Course Outcomes:
After the completion of the course, students will be able
CO1: Construct Node File with JavaScript, Accessing a Node.js
CO2: Explain and Develop Node JS Modules
CO3: Build Web Server
CO4: Extend Node.js to work with Database

UNIT – I : 8hrs

Introduction
Features and advantages of Node JS, Traditional Web Server Model, Node.js Process Model, Asynchronous programming with Node.js, Types of applications that can be developed using Node.js
Setup Development Environment
Install Node.js on Windows, working in REPL, Node JS Console, Creating a Node File with JavaScript, Accessing a Node.js File Through the Command Line Interface, Using Node.js in Net- Beans IDE.

Teaching Methodology:
Chalk and Board
Power Point presentation
Activity based Teaching

UNIT – II: 8hrs

Node.JS Basics
Primitive Types, Object Literal, Functions, Buffer, Access Global Scope.
Node.JS Modules
Module, Module Types: Core Modules, Local Modules, Third Party Modules, Module Exports. Using Modules in a Node.js File, Using the Built in HTTP, URL, Query String Module, Creating a Custom Module
Node Package Manager
NPM, Installing Packages Locally, Adding dependency in package.json, Installing packages globally, Updating packages.

Teaching Methodology:
Chalk and Board
Power Point presentation
Activity based Teaching

UNIT – III: 8hrs

Creating Web Server:
Handling HTTP requests, Sending requests

tem Reading, Writing a File, Writing a file asynchronously, Opening a file, deleting a file, Other IO Operations: Append, Rename, Truncate. File System Module with URL Module Create, Read, Remove a Directory **Debugging Node JS Application** Core Node.js debugger, Node Inspector, Built-in debugger in IDEs **Teaching Methodology:** Chalk and Board Power Point presentation Activity based Teaching
UNIT – IV: 8hrs
Events EventEmitter class, Methods and Events of EvenEmitter Class, Returning event emitter, Extend EventEmitter Class, Passing Arguments and 'this' to listeners, Asynchronous and Synchronous call, Handle Events only Once, Error Events. **Database Connectivity** Connection string, Configuring, Working with insert, select command, Updating records, Deleting records, Drop tables, Ordered Result Set **Teaching Methodology:** Chalk and Board Power Point presentation Activity based Teaching
UNIT – V: 8hrs
Express and Node JS Introduction to Express Framework, Express Server Request-Response Routes, Route Parameters, Multiple Route Callback/Handler Functions, Methods of Response Object, Chaining Route Handlers, Send Static Files, Accept User Input, File Upload with Express, Manage Cookies, Send file as a response, Templates and Express. **RestFul API** What is REST, Install PostMan, Get all articles, Post a new article, Delete request, Make a specific Get request, put operation on a specific record, patch request, deleting a specific record **Authentication and Security** Introduction, register and logic, encryption, cookies and session **Teaching Methodology:** Chalk and Board Power Point presentation Activity based Teaching

Reference Books:

1. Dhruti Shah, "Node.JS Guidebook", BPB Publications, 2018.
2. Basarat Ali Syed, Beginning Node.js, A press, 2014

Paper Code: 22MCA36 **Theory/Week: 4 Hours** **Credits: 2**	**WEB APPLICATION FRAMEWORK AND AWS LAB**	**Hours: 40** **IA: 50** **Exam: 50**
Course Objectives: • To overview on developing mobile application using android.		
Course Outcomes: After the completion of the course, students will be able CO1: Develop effective user interfaces that leverage evolving mobile devices CO2: Develop applications using software development kits (SDKs), frameworks and toolkits. CO3: Implement suitable methods to integrate database and server-side technologies		

Teaching Plan

Total Hours : 40 **Internal Marks: 50**

Hours/Week: 4 hours **External Marks: 50**

Course Objective:

- Understand Node JS and REPL terminal.
- Experiment with Node JS Modules and Node Package Manager.
- Develop applications to handle events in Node JS
- Make use of Web Server to manage database.
- Demonstrate Express Framework

Learning Outcome: After the completion of the course, students will be able to

CO1: Construct Node File with JavaScript, Accessing a Node.js

CO2: Explain and Develop Node JS Modules

CO3: Build Web Server

CO4: Extend Node.js to work with Database

UNIT-1

Chapter 1: Introduction **8Hrs**

Session 1: Features and advantages of Node JS,

Session 2: Traditional Web Server Model,

Session 3: Node.js Process Model,

Session 4: Asynchronous programming with Node.js,

Session 5:Types of applications that can be developed using Node.js

Chapter 2: Setup Development Environment

Session 6: Install Node.js on Windows, working in REPL,

Session 7: Node JS Console, Creating a Node File with JavaScript, Accessing a Node.js File Through the Command Line Interface,

Session 8: Using Node.js in Net- Beans IDE.

UNIT-2

Chapter 3: Node.JS Basics **8hrs**

Session 1: Primitive Types, Object Literal

Session 2: Functions, Buffer, Access Global Scope.

Chapter 4: Node.JS Modules

Session 3:Module, Module Types: Core Modules, Local Modules, Third Party Modules

Session 4:, Module Exports. Using Modules in a Node.js File,

Session 5: Using the Built in HTTP, URL,

Session 6: Query String Module, Creating a Custom Module

Chapter 5: Node Package Manager
Session 7: NPM, Installing Packages Locally
Session 8: Adding dependency in package.json, Installing packages globally, Updating packages.

UNIT-3

Chapter 6: Creating Web Server and File System 8hrs

Session 1: Handling HTTP requests, Sending requests

Session 2: Reading, Writing a File, Writing a file asynchronously,

Session 3: Opening a file, deleting a file, Other IO

Session 4: Operations: Append, Rename, Truncate.

Session 5: File System Module with URL Module Create, Read, Remove a Directory

Chapter 7 :Debugging Node JS Application

Session 6: Core Node.js debugger

Session 7: Node Inspector,

Session 8: Built-in debugger in IDEs

UNIT-4

Chapter 8: Events 8hrs

Session 1: EventEmitter class, Methods and Events of EvenEmitter Class,

Session 2: Returning event emitter, Extend EventEmitter Class,

Session 3: Passing Arguments and ‘this’ to listeners,

Session 4: Asynchronous and Synchronous call, Handle Events only Once, Error Events.

Chapter 9:Database Connectivity

Session 5: Connection string, Configuring,

Session 6: Working with insert, select command,

Session 7: Updating records, Deleting records,

Session 8:Drop tables, Ordered Result Set ,

UNIT-5

Chapter 10:Express and Node JS 8hrs

Session 1: Introduction to Express Framework, Express Server Request-Response Routes,Route Parameters, Multiple Route Callback/Handler Functions

Session 2: Methods of Response Object, Chaining Route Handlers, Send Static Files, Accept User Input,

Session 3: File Upload with Express, Manage Cookies, Send file as a response, Templates and Express.

Session 4: What is REST, Install PostMan, Get all articles,

Session 5: Post a new article, Delete request, Make a specific Get request,

Session 6: Put operation on a specific record, patch request, deleting a specific record

Session 7: Introduction, register and logic

Session 8: Encryption, cookies and session

Reference Books

- Dhruti Shah, “Node.JS Guidebook”, BPB Publications, 2018.
- Basarat Ali Syed, Beginning Node.js, A press, 2014

Unit-1

Chapter-1

Introduction to Node.js

Definition of Node.js

Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside of a web browser. It uses the V8 JavaScript engine from Google to compile JavaScript code into native machine code, allowing it to run at near-native speeds.

History and Evolution of Node.js

Node.js was created by Ryan Dahl in 2009. It was inspired by the limitations of traditional web servers, which were not well-suited for handling a large number of concurrent connections. Node.js was designed to be lightweight and efficient, making it ideal for building scalable network applications.

1.1 Features and Advantages of Node.js

Non-blocking, Event-driven Architecture

Node.js uses a non-blocking, event-driven I/O model, which means that it can handle multiple connections simultaneously without blocking the execution of other tasks. This makes Node.js highly efficient for handling I/O-bound operations.

JavaScript Everywhere

Node.js allows developers to use JavaScript for both client-side and server-side development. This reduces the need to switch between languages, making development more streamlined and efficient.

Fast Execution

Node.js uses the V8 JavaScript engine, which compiles JavaScript code to native machine code. This results in fast execution speeds, making Node.js well-suited for building high-performance applications.

Single-threaded, Highly Scalable

Node.js uses a single-threaded event loop model, which allows it to handle a large number of concurrent connections efficiently. This makes Node.js highly scalable and well-suited for building real-time applications.

Large Ecosystem of Modules (npm)

Node.js has a vast ecosystem of third-party modules available through the Node Package Manager (npm). This makes it easy for developers to add functionality to their applications without having to reinvent the wheel.

Good for Real-time Applications

Node.js is well-suited for building real-time applications such as chat applications and online gaming platforms. Its non-blocking, event-driven architecture makes it easy to handle real-time interactions with clients.

Example:

```
// Simple HTTP server using Node.js

const http = require('http');


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

  res.statusCode = 200;

  res.setHeader('Content-Type', 'text/plain');

  res.end('Hello, world!\n');

});


server.listen(3000, '127.0.0.1', () => {

  console.log('Server running at http://127.0.0.1:3000/');

});
```

1.2 Traditional Web Server Model

Overview of the Traditional Web Server Model

In the traditional web server model, each client request is handled by a separate thread or process. This can lead to scalability issues when handling a large number of concurrent connections, as each thread or process consumes system resources.

Issues with the Traditional Model

The traditional web server model can be inefficient and resource-intensive, especially for I/O-bound applications that spend a significant amount of time waiting for I/O operations to complete. This can lead to poor performance and scalability issues.

Comparison with Node.js

Node.js' event-driven, non-blocking I/O model allows it to handle a large number of concurrent connections efficiently. By using a single-threaded event loop, Node.js can perform non-blocking I/O operations without consuming system resources unnecessarily.

Example:

```
// Traditional web server model (pseudo-code)

const http = require('http');


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

  // Handle each request in a new thread

  // This is a simplified example and does not handle concurrency well

  processRequest(req, res);

});


function processRequest(req, res) {

  // Processing logic here

  res.end('Hello, world!\n');

}
```

```
server.listen(3000, '127.0.0.1', () => {
  console.log('Server running at http://127.0.0.1:3000/');
});
```

1.3 Node.js Process Model

Overview of the Node.js Process Model

Node.js is a single-threaded runtime environment, but it uses an event-driven architecture to handle I/O operations asynchronously. This allows Node.js to handle multiple operations concurrently without blocking the execution of other tasks.

Event Loop and Its Role

The event loop is a core component of Node.js that continuously checks for events and executes callback functions in response to these events. This allows Node.js to handle I/O operations asynchronously, improving the performance of I/O-bound applications.

Handling I/O Operations Asynchronously

Node.js uses non-blocking I/O operations, which allows it to perform other tasks while waiting for I/O operations to complete. This improves the efficiency of I/O-bound applications and allows Node.js to handle a large number of concurrent connections.

Node.js Event-driven Architecture

Node.js uses events and event emitters to handle asynchronous operations. For example, the **fs** module in Node.js uses event emitters to handle file I/O operations asynchronously. This allows Node.js to handle I/O-bound operations efficiently without blocking the event loop.

Example:

```
// Node.js event loop example
const fs = require('fs');


// Asynchronous file read operation
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);

    return;

  }

  console.log(data);

});
```

1.4 Asynchronous Programming with Node.js

Understanding Asynchronous Programming

Asynchronous programming in Node.js allows multiple operations to be performed concurrently, improving the performance of I/O-bound applications. Asynchronous programming is achieved using callbacks, promises, and async/await.

Callbacks and Their Role in Node.js

Callbacks are functions that are passed as arguments to other functions and are called when a specific event occurs. In Node.js, callbacks are used to handle the completion of asynchronous operations. For example, the **fs.readFile** method in Node.js takes a callback function as an argument to handle the completion of the file read operation.

Promises and Async/Await in Node.js

Promises are a way to handle asynchronous operations in a more readable and maintainable way. Promises represent a value that may be available in the future, allowing developers to chain asynchronous operations together. Async/await is a syntactic sugar for working with promises, making asynchronous code look more like synchronous code.

Error Handling in Asynchronous Operations

Node.js uses the callback pattern to handle errors in asynchronous operations. Developers can pass an error object as the first argument to the callback function to indicate that an error has occurred. Promises and async/await also provide mechanisms for error handling, allowing developers to catch and handle errors in asynchronous code.

Example:

```
// Asynchronous file read using callback

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

```
fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});

// Asynchronous file read using promises
const util = require('util');
const readFile = util.promisify(fs.readFile);

readFile('example.txt', 'utf8')
  .then(data => console.log(data))
  .catch(err => console.error(err));

// Asynchronous file read using async/await
async function readFileAsync() {
  try {
    const data = await readFile('example.txt', 'utf8');
    console.log(data);
  } catch (err) {
    console.error(err);
  }
}
```

```
readFileAsync();
```

1.5 Types of Applications that can be Developed using Node.js

Web Applications

Node.js is commonly used for building web applications, thanks to its ability to handle a large number of concurrent connections efficiently. For example, LinkedIn uses Node.js for its mobile backend.

Real-time Applications

Node.js is well-suited for building real-time applications such as chat applications and online gaming platforms, thanks to its event-driven, non-blocking I/O model. For example, Slack and Twitch use Node.js for their real-time features.

API Servers

Node.js is often used for building API servers due to its lightweight and scalable nature. For example, PayPal uses Node.js for its API servers.

Command-line Applications

Node.js can be used to build command-line applications, thanks to its ability to interact with the file system and execute shell commands. For example, the npm CLI is built with Node.js.

Internet of Things (IoT) Applications

Node.js can be used to build IoT applications, thanks to its ability to interact with hardware devices. For example, Tessel is a JavaScript-enabled microcontroller that runs Node.js.

Chapter-2

Setup Development Environment for Node.js

2.1 Install Node.js on Windows

1. **Download Node.js:** Visit the official Node.js website (https://nodejs.org) and download the Windows installer.
2. **Run the Installer:** Double-click the downloaded installer to run it. Follow the on-screen instructions in the installation wizard.

3. **Verify Installation:** After the installation is complete, open a command prompt and type **node -v**. This command should print the installed version of Node.js. Similarly, you can check the version of npm (Node Package Manager) by typing **npm -v**.

2.2 Working in REPL (Read-Eval-Print Loop)

1. **Start REPL:** Open a command prompt and type **node** to start the Node.js REPL.
2. **Use the REPL:** You can now enter JavaScript code directly into the REPL. Press Enter to execute the code. For example:

```
> console.log("Hello, Node.js!");
```

1. **Exit REPL:** To exit the REPL, press Ctrl + C twice or type **.exit**.

2.3 Node.js Console

1. **Create a Node.js File:** Create a new file named **console.js** and open it in a text editor.
2. **Write JavaScript Code:** Use the **console.log()** method to print messages to the console. For example:

```
console.log("Hello, Node.js!");
```

1. **Run the Node.js File:** Save the file and run it from the command line using **node console.js**. You should see the message printed to the console.

2.4 Creating a Node File with JavaScript

1. **Create a Node.js File:** Create a new file named **app.js** and open it in a text editor.
2. **Write JavaScript Code:** Write your JavaScript code in the file. For example:

```
function greet(name) {
  return "Hello, " + name + "!";
}


console.log(greet("Alice"));
```

3. **Run the Node.js File:** Save the file and run it from the command line using **node app.js**. You should see the greeting message printed to the console.

2.5 Accessing a Node.js File Through the Command Line Interface

1. **Navigate to File Directory:** Open a command prompt and navigate to the directory containing your Node.js file.
2. **Run the File:** Run the Node.js file using **node <filename>.js**, replacing **<filename>** with the name of your file.

2.6 Using Node.js in NetBeans IDE

1. **Install NetBeans IDE:** Download and install the latest version of NetBeans IDE from the official website (https://netbeans.apache.org/).
2. **Install Node.js Plugin:** Open NetBeans IDE and go to Tools > Plugins. Search for "Node.js" in the Available Plugins tab and install the Node.js plugin.
3. **Create a Node.js Project:** Go to File > New Project. Select "Node.js" from the list of project types and click Next. Follow the wizard to create a new Node.js project.
4. **Write and Run Code:** Write your Node.js code in the IDE. Use the built-in features for editing, debugging, and running Node.js applications.

Answer the following:

1. **Explain the features of Node.js that make it suitable for building scalable web applications.**
2. **Compare the traditional web server model with the Node.js process model.**
3. **Give the advantages of the Node.js model.**
4. **Discuss the concept of asynchronous programming in Node.js and list its advantages**
5. **Describe the steps to install Node.js on a Windows system. Include the process of setting up the development environment and running a simple Node.js file.**
6. **Explain the role of the Node.js Console in development. How can developers use it to debug and interact with Node.js applications?**
7. **Explain how the event loop works in Node.js and its significance in building scalable applications.**

8. **Provide a step-by-step explanation of the event loop's processing of asynchronous tasks.**

10 marks:

9. **How does Node.js leverage non-blocking, event-driven architecture to handle concurrent requests efficiently? Provide a detailed explanation with examples.**

10. **Describe the Node.js module system. How does it help in organizing and managing code in Node.js applications? Provide examples to demonstrate module creation, import, and export in Node.js.**

Unit-2

Chapter-3

Node.js Basics

3.1 Primitive Types

Node.js, like JavaScript, supports several primitive types:

1. **String**: Represents a sequence of characters enclosed within single or double quotes.

```
let name = "John";
```

Number: Represents numeric values, including integers and floating-point numbers.

```
let age = 30;

let price = 19.99;
```

Boolean: Represents a logical value, either **true** or **false**.

```
let isStudent = true;
```

Null: Represents the intentional absence of any value.

```
let car = null;
```

Undefined: Represents a variable that has been declared but not assigned a value.

```
let city;
```

3.2 Object Literal

An object literal is a key-value pair representation enclosed within curly braces **{}**:

```
let person = {

  name: "Alice",

  age: 25,

  isStudent: false,

};
```

3.3 Functions

Functions in Node.js are similar to functions in JavaScript and can be defined using the **function** keyword:

```
function greet(name) {
  return "Hello, " + name + "!";
}

console.log(greet("Alice")); // Output: Hello, Alice!
```

Functions can also be defined using arrow functions (ES6):

```
let greet = (name) => {
  return "Hello, " + name + "!";
};

console.log(greet("Alice")); // Output: Hello, Alice!
```

3.4 Buffer

Buffers are instances of the **Buffer** class that represent fixed-size chunks of memory:

```
let buf = Buffer.from("hello", "utf8");
console.log(buf); // Output: <Buffer 68 65 6c 6c 6f>
```

Buffers are commonly used in Node.js to work with binary data, such as reading from or writing to files, or working with network protocols.

3.5 Access Global Scope

In Node.js, variables declared outside of any function or block have global scope:

```
let globalVar = "I'm a global variable";

function foo() {
  console.log(globalVar); // Output: I'm a global variable
}


foo();
```

To access global variables within Node.js modules, you can use the **global** object:

```
globalVar = "I'm a global variable";
```

Example: Creating a Node.js Module

```
// myModule.js
let message = "Hello from myModule.js";


function greet(name) {
  return "Hello, " + name + "!";
}


module.exports = {
  message: message,
  greet: greet,
};


module.exports = {
  globalVar: globalVar,
};
```

<u>In another file:</u>

```
// index.js
const myModule = require('./myModule');
```

console.log(myModule.message); // Output: Hello from myModule.js

console.log(myModule.greet('Alice')); // Output: Hello, Alice!

Chapter-4

Node.js Modules

4.1 Module

- A module in Node.js is a reusable block of code that encapsulates related functionality.
- Each module in Node.js has its own scope, so the variables and functions defined in a module are private to that module by default.

4.2 Module Types

1. **Core Modules**: These are modules that are included with Node.js and provide basic functionality. Examples include **fs** (file system), **http** (HTTP server), **os** (operating system), etc.
2. **Local Modules**: These are modules that you create in your Node.js application. You can create a local module by defining a JavaScript file and using **module.exports** to export functions or objects from that file.
3. **Third-party Modules**: These are modules created by third-party developers and are not included with Node.js. You can install third-party modules using npm (Node Package Manager) and then use **require()** to include them in your application.

4.3 Module Exports

- To export a function or object from a module, you use the **module.exports** or **exports** object.
- Anything assigned to **module.exports** or **exports** will be available for other modules to require.

4.4 Using Modules in a Node.js File

- To use a core module, you simply require it using **require()**. For example, to use the **fs** module to read a file:
- const fs = require('fs');
- To use a local module, you require it using a relative path. For example, if you have a module named **myModule.js** in the same directory as your current file:

- const myModule = require('./myModule');
- To use a third-party module, you first need to install it using npm. For example, to use the **lodash** module:
- npm install lodash
- Then, you can require it in your code:
- const _ = require('lodash');

Using the Built-in HTTP, URL, Query String Module

- Node.js includes several built-in modules for common tasks. For example, the **http** module can be used to create a simple HTTP server:

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


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

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

  res.end('Hello, World!');

}).listen(3000);
```

The **url** module can be used to parse URLs:

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

let parsedUrl = url.parse('https://www.example.com/path?query=value');

console.log(parsedUrl);
```

The **querystring** module can be used to parse query strings:

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

let parsedQuery = querystring.parse('query=value&name=John');

console.log(parsedQuery);
```

4.5 Creating a Custom Module

- To create a custom module, you simply define your module in a JavaScript file and use **module.exports** to export functions or objects from that file.

- For example, to create a module that calculates the area of a circle:

```
// circle.js
const PI = 3.14159;

function calculateArea(radius) {
  return PI * radius * radius;
}

module.exports = {
  calculateArea: calculateArea
};
```

You can then use this module in another file:

```
const circle = require('./circle');
console.log(circle.calculateArea(5)); // Output: 78.53975
```

Chapter-5

Node Package Manager (npm)

5.1 NPM

- npm is the default package manager for Node.js and is used to install, manage, and publish packages/modules.
- npm comes bundled with Node.js installation.

Installing Packages Locally

- To install a package locally, use the **npm install** command followed by the package name.
- The package will be installed in the **node_modules** directory of your project.
- npm will also automatically update the **package-lock.json** file to reflect the installed package and its dependencies.

5.2 Adding Dependency in package.json

- When you install a package using **npm install**, npm automatically adds it to the **dependencies** section of your **package.json** file.
- This allows you to track the packages your project depends on.
- To install all dependencies listed in **package.json**, use **npm install** without any package name.

5.3 Installing Packages Globally

- To install a package globally (i.e., accessible from any project), use the **-g** flag.
- Global packages are installed in a system directory and can be used in any project.
- However, it's generally recommended to install packages locally within your project whenever possible, as global packages can lead to dependency conflicts.
- npm install -g packageName

5.4 Updating Packages

- To update a package to the latest version, use the **npm update** command followed by the package name.
- To update all packages in your project to their latest versions, use **npm update** without any package name.
- npm will update the **package-lock.json** file to reflect the updated packages and their dependencies.
- npm update packageName

Example

1. Installing a package locally:

   npm install lodash

1. This will install the lodash package in the **node_modules** directory of your project and add it to **dependencies** in **package.json**.
2. Installing a package globally:

   npm install -g nodemon

This will install the nodemon package globally, allowing you to use it as a command-line tool.

Updating a package:

npm update lodash

This will update the lodash package to the latest version.

Answer the following:

1. **Explain the concept of primitive types in Node.js. Provide examples of primitive types and their uses in Node.js.**
2. **What is an object literal in Node.js? How can you create and use object literals in your Node.js applications?**
3. **Describe the role of functions in Node.js. How are functions defined and called in Node.js, and what are their advantages?**
4. **What is a buffer in Node.js? How are buffers used to handle binary data in Node.js applications?**
5. **How can you access the global scope in Node.js? What precautions should be taken when working with the global scope in Node.js applications?**
6. **What is the purpose of the Node Package Manager (npm) in Node.js development? How can you use npm to manage dependencies in your Node.js projects?**
7. **How can you add a dependency to your project's package.json file using npm? Explain the benefits of adding dependencies to the package.json file.**
8. **Explain the difference between installing packages locally and globally using npm. When would you use each approach in your Node.js projects?**

10 marks:

9. **Discuss the concept of modules in Node.js. Explain the different types of modules (core modules, local modules, third-party modules) and how they are used in Node.js applications. Provide examples to illustrate each type of module.**

10. **Explain the process of using modules in a Node.js file. How can you import and use modules in your Node.js applications? Provide examples to demonstrate the import and use of modules.**

Unit-3

Chapter-6

Creating a Web Server in Node.js

6.1 Handling HTTP Requests

- Node.js includes a built-in **http** module that allows you to create a web server.
- To create a basic HTTP server, you can use the **createServer** method from the **http** module. This method takes a callback function that is called whenever a request is received.
- Inside the callback function, you can handle the request and send a response back to the client.

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


http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!');
}).listen(3000);
```

6.2 Sending Requests

- You can also use the **http** module to send HTTP requests to other servers.
- To send a GET request, you can use the **http.get** method. This method takes a URL and a callback function that is called with the response.

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


http.get('http://example.com', (res) => {
  let data = '';


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

  res.on('end', () => {
    console.log(data);
  });
});
```

Receiving Requests

- When a request is received by your HTTP server, the callback function passed to **createServer** is called with two arguments: **req** (the request object) and **res** (the response object).
- You can use the **req** object to access information about the request, such as the URL, method, and headers.
- You can use the **res** object to send a response back to the client.

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

http.createServer((req, res) => {
  console.log(`Received ${req.method} request for ${req.url}`);

  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello, World!');
}).listen(3000);
```

Chapter-7

File System Operations in Node.js

7.1 Reading a File

- Node.js provides the **fs** (File System) module to work with files.
- To read a file, you can use the **fs.readFile** method. This method takes the file path and an optional encoding parameter.
- If no encoding is specified, the raw buffer data is returned.

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


fs.readFile('file.txt', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
});
```

7.2 Writing a File

- To write to a file, you can use the **fs.writeFile** method. This method takes the file path, data to write, and an optional callback function.
- If the file already exists, it will be overwritten. If not, a new file will be created.

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


fs.writeFile('file.txt', 'Hello, World!', (err) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log('File written successfully');
});
```

7.3 Writing a File Asynchronously

- To write to a file asynchronously, you can use the **fs.writeFileSync** method. This method takes the file path, data to write, and an optional encoding parameter.
- Unlike **writeFile**, **writeFileSync** blocks the execution of further code until the file is written

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


try {
  fs.writeFileSync('file.txt', 'Hello, World!');
  console.log('File written successfully');
} catch (err) {
  console.error(err);
}
```

7.4 Opening a File

- To open a file, you can use the **fs.open** method. This method takes the file path and a flag indicating the mode in which to open the file (e.g., read, write, append).
- After opening the file, you can perform further operations using the file descriptor returned by **fs.open**.

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


fs.open('file.txt', 'r', (err, fd) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log('File opened successfully');
  // Further operations using fd
});
```

7.5 Deleting a File

- To delete a file, you can use the **fs.unlink** method. This method takes the file path and a callback function.

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


fs.unlink('file.txt', (err) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log('File deleted successfully');
});
```

7.6 Other IO Operations: Append, Rename, Truncate

- **Append to a File**: Use the **fs.appendFile** method to append data to a file.
- **Rename a File**: Use the **fs.rename** method to rename a file.
- **Truncate a File**: Use the **fs.truncate** method to truncate (empty) a file to a specified length

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


// Append to a file
fs.appendFile('file.txt', 'More data', (err) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log('Data appended to file');
});

// Rename a file
fs.rename('file.txt', 'newfile.txt', (err) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log('File renamed successfully');
});

// Truncate a file
fs.truncate('file.txt', 0, (err) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log('File truncated successfully');
});
```

7.7 File System Module with URL Module

- Node.js also provides the **url** module for working with URLs.
- Combining the **fs** and **url** modules, you can create, read, and remove directories based on URLs.
- Here's an example of creating a directory based on a URL

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

const url = require('url');


const parsedUrl = new URL('http://example.com/path/to/directory');

const directoryPath = parsedUrl.pathname.substring(1);


fs.mkdir(directoryPath, { recursive: true }, (err) => {

  if (err) {

    console.error(err);

    return;

  }

  console.log('Directory created successfully');

});
```

Debugging Node.js Application

7.8 Core Node.js Debugger

- Node.js includes a built-in debugger that allows you to debug your applications using the command line.
- To start the debugger, use the **--inspect** flag when running your Node.js application
- node --inspect app.jsThis will start the debugger and print a message with a URL that you can open in Chrome DevTools to debug your application.

7.9 Node Inspector

- Node Inspector is a debugger interface for Node.js applications that provides a more user-friendly debugging experience than the core debugger.

To use Node Inspector, first install it globally using npm

```
npm install -g node-inspector
```

Then, start your Node.js application with the **--inspect** flag:

```
node-inspector
```

This will start Node Inspector and print a URL that you can open in a web browser to debug your application.

7.10 Built-in Debugger in IDEs

- Many IDEs (Integrated Development Environments) provide built-in debuggers for Node.js applications, making it easier to debug your code.
- Examples of IDEs with built-in Node.js debuggers include Visual Studio Code, WebStorm, and Atom.
- To use the built-in debugger in your IDE, set breakpoints in your code and then start the debugger from within the IDE.

Debugging Steps

1. **Set Breakpoints**: Identify the areas of your code where you want to pause execution to inspect variables and the application state.
2. **Start Debugger**: Use one of the methods mentioned above to start the debugger for your Node.js application.
3. **Debugging Process**: Once the debugger is running, it will pause execution at the breakpoints you've set. You can then use the debugging interface to inspect variables, step through code, and track the flow of your application.
4. **Fix Issues**: Use the information gathered during debugging to fix any issues in your code.
5. **Resume Execution**: Once you've fixed the issues, you can resume execution of your application to continue testing.

Example Usage in Visual Studio Code

1. Install the "Debugger for Chrome" extension in Visual Studio Code.
2. Open your Node.js project in Visual Studio Code.
3. Set breakpoints in your code by clicking on the left margin next to the line numbers.

4. Start your Node.js application with the **--inspect** flag:

```
node --inspect app.js
```

5. In Visual Studio Code, go to the Debug view (Ctrl + Shift + D), select "Attach to Node.js", and click the green play button to start debugging.
6. Visual Studio Code will connect to the debugger and pause execution at your breakpoints. You can then use the debugging tools in Visual Studio Code to inspect variables and debug your code.

<u>Answer the following:</u>

1. **Explain the process of creating a basic web server in Node.js using the http module. Include the steps to create a server, define request handlers, and listen on a port.**
2. **Explain the GET method in HTTP. How is it used to request data from a server? Provide an example of a GET request in a Node.js application.**
3. **Discuss the POST method in HTTP. How is it used to send data to a server? Provide an example of a POST request in a Node.js application.**
4. **How can you read a file synchronously in Node.js? With an example explain of reading a file.**
5. **Explain the difference between reading a file synchronously and asynchronously in Node.js. When would you use each approach in your applications?**
6. **Describe the process of writing a file in Node.js. How can you use the fs module to write content to a file on the file system?**
7. **How can you delete a file in Node.js using the fs module? With an example of explain deleting a file.**
8. **Explain the concept of debugging in Node.js. What tools and techniques can be used to debug Node.js applications?**

<u>10 marks:</u>

9. **Discuss the role of the Node Inspector in debugging Node.js applications. How does it differ from the core Node.js debugger?**
10. **Describe the role of the URL module in Node.js. How can you use this module to parse and manipulate URLs in your applications?**

Unit-4

Chapter-8

Events in Node.js

8.1 EventEmitter Class

- The **EventEmitter** class in Node.js is used to handle events. It provides an implementation of the observer pattern.
- You can create an instance of **EventEmitter** and use it to emit events and register event listeners.

```
const EventEmitter = require('events');

const myEmitter = new EventEmitter();
```

8.2 Methods and Events of EventEmitter Class

- **Methods**:
  - **on(eventName, listener)**: Adds a listener for the specified event.
  - **emit(eventName[, ...args])**: Emits the specified event with optional arguments.
  - **once(eventName, listener)**: Adds a one-time listener for the specified event.
  - **removeListener(eventName, listener)**: Removes the specified listener for the specified event.
  - **removeAllListeners([eventName])**: Removes all listeners for the specified event, or all listeners if no event name is provided.
- **Events**:
  - **newListener**: Emitted when a new listener is added.
  - **removeListener**: Emitted when a listener is removed.

8.3 Returning EventEmitter

- You can return an **EventEmitter** instance from a function to allow users to listen for events on that instance.

- This is useful for creating modules that emit events.

```
function createEmitter() {
  const emitter = new EventEmitter();
  // Add some listeners or emit events
  return emitter;
}

const myEmitter = createEmitter();
```

8.4 Extending EventEmitter Class

- You can extend the **EventEmitter** class to create your own custom event emitter classes.
- This allows you to add custom methods and properties to your event emitter

```
class MyEmitter extends EventEmitter {
  constructor() {
    super();
    // Additional initialization
  }

  // Custom methods
  myMethod() {
    // Do something
  }
}

const myEmitter = new MyEmitter();
```

8.5 Passing Arguments and 'this' to Listeners

- When emitting an event, you can pass arguments to the listeners of that event.
- The **this** keyword inside a listener refers to the **EventEmitter** instance that emitted the event

```
myEmitter.on('event', function(arg1, arg2) {
  console.log(arg1, arg2, this);
});


myEmitter.emit('event', 'arg1', 'arg2');
```

8.6 Asynchronous and Synchronous Call

- By default, event listeners are called synchronously in the order they were added.
- However, you can use the **setImmediate** or **process.nextTick** methods to defer the execution of an event listener to the next iteration of the event loop, making it asynchronous.

```
myEmitter.on('event', (arg1, arg2) => {
  setImmediate(() => {
    console.log('This listener is asynchronous');
  });
});
```

8.7 Handling Events Only Once

- You can use the **once** method to add a one-time listener that is automatically removed after it's called.
- This is useful for handling events that should only be handled once

```
myEmitter.once('event', () => {
  console.log('This listener will only be called once');
});
```

8.8 Error Events

- The **EventEmitter** class emits an **error** event when an error occurs.

- If an **error** event is emitted without an event listener, the Node.js process will exit with an error code.

```
myEmitter.on('error', (err) => {
  console.error('Error:', err);
});


myEmitter.emit('error', new Error('Something went wrong'));
```

Chapter-9

Database Connectivity in Node.js

9.1 Connection String

- A connection string is a string that specifies the information needed to connect to a database.
- It typically includes details such as the database server's address, port number, database name, and authentication credentials.

9.2 Configuring Database Connection

- To configure a database connection in a Node.js application, you can use a database driver/module specific to the database you're using (e.g., **mysql** for MySQL, **mongodb** for MongoDB).
- First, install the appropriate database driver using npm

```
npm install mysql
```

Then, create a connection pool or establish a connection to the database using the driver's API.

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


const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: 'password',
  database: 'mydatabase'
});

connection.connect((err) => {
  if (err) {
    console.error('Error connecting to database:', err);
    return;
  }
  console.log('Connected to database');
});
```

9.3 Working with Insert Command

- To insert records into a database table, you can use the **INSERT INTO** SQL command.
- The following example inserts a new record into a **users** table:

```
connection.query('INSERT INTO users (name, email) VALUES (?, ?)', ['John Doe',
'john@example.com'], (err, result) => {
  if (err) {
    console.error('Error inserting record:', err);
    return;
  }
  console.log('Record inserted successfully');
});
```

9.4 Working with Select Command

- To retrieve records from a database table, you can use the **SELECT** SQL command.
- The following example retrieves all records from the **users** table

```
connection.query('SELECT * FROM users', (err, results) => {
  if (err) {
    console.error('Error selecting records:', err);
    return;
  }
  console.log('Records:', results);
});
```

9.5 Updating Records

- To update records in a database table, you can use the **UPDATE** SQL command.
- The following example updates the email address of a user with ID **1**

```
connection.query('UPDATE users SET email = ? WHERE id = ?', ['newemail@example.com', 1],
(err, result) => {
  if (err) {
    console.error('Error updating record:', err);
    return;
  }
  console.log('Record updated successfully');
});
```

9.6 Deleting Records

- To delete records from a database table, you can use the **DELETE FROM** SQL command.
- The following example deletes a user with ID **1**

```
connection.query('DELETE FROM users WHERE id = ?', [1], (err, result) => {
  if (err) {
    console.error('Error deleting record:', err);
    return;
  }
  console.log('Record deleted successfully');
});
```

9.7 Dropping Tables

- To drop a table from a database, you can use the **DROP TABLE** SQL command.
- **Note:** Be careful when using this command, as it will permanently delete the table and all its data.

```
connection.query('DROP TABLE users', (err, result) => {
  if (err) {
    console.error('Error dropping table:', err);
    return;
  }
  console.log('Table dropped successfully');
});
```

9.8 Ordered Result Set

- To retrieve records from a database table in a specific order, you can use the **ORDER BY** clause in your **SELECT** SQL command.
- The following example retrieves all records from the **users** table ordered by the **name** column:

```
connection.query('SELECT * FROM users ORDER BY name', (err, results) => {
  if (err) {
    console.error('Error selecting records:', err);
    return;
  }
  console.log('Records ordered by name:', results);
});
```

Answer the following:

1. **Explain the EventEmitter class in Node.js. How is it used to handle events in Node.js applications?**

2. **How can you return an event emitter from a function in Node.js? Provide an example to demonstrate the process.**

3. **Describe how you can extend the EventEmitter class to create custom event emitters in Node.js. Provide an example of extending the class and using the custom event emitter.**

4. **Explain the process of passing arguments to event listeners in Node.js. How can you use 'this' to reference the EventEmitter object within a listener?**

5. **How can you ensure that an event handler is only executed once in Node.js? Provide an example to demonstrate this behavior.**

6. **What is a connection string in the context of database connectivity? How can you configure a connection string for connecting to a database in Node.js?**

7. **Describe the process of working with insert and select commands in Node.js for database operations. Provide examples of inserting and selecting records from a database.**

8. **Discuss the common types of errors that can occur during database operations and how you can handle them in your Node.js applications.**

10 marks:

9. **Discuss the methods and events of the EventEmitter class. Provide examples of each method and event and explain their use cases.**

10. **Describe how you can retrieve an ordered result set from a database using Node.js. Discuss the use of the ORDER BY clause and provide examples to demonstrate ordering results.**

Unit-5

Chapter -10

Express and Node.js

10.1 Introduction to Express Framework

- Express is a web application framework for Node.js.
- It provides a set of features for building web applications and APIs.
- Key features include middleware, routing, and templating.

10.2 Express Server

- Use **express()** to create an Express application.
- Start the server with the **listen** method, specifying a port number.
- Example

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

const app = express();

app.listen(3000, () => {

  console.log('Server started on port 3000');

});
```

Request-Response

- Use the **req** object to access the request details.
- Use the **res** object to send a response back to the client.
- Example

```
app.get('/', (req, res) => {

  res.send('Hello, World!');

});
```

Routes

- Define routes for different URLs using **app.get**, **app.post**, etc.
- Example

```
app.get('/about', (req, res) => {
  res.send('About page');
});
```

10.3 Route Parameters

- Use route parameters to capture values from the URL.
- Example

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

10.4 Multiple Route Callback/Handler Functions

- Define multiple callback functions for a single route.
- Example

```
app.get('/example', (req, res, next) => {
  console.log('First callback');
  next();
}, (req, res) => {
  res.send('Second callback');
});
```

10.5 Methods of Response Object

- Use methods like **res.send**, **res.json**, **res.sendFile** to send responses.
- Example

```
app.get('/json', (req, res) => {
  res.json({ message: 'Hello, JSON!' });
});
```

10.6 Chaining Route Handlers

- Chain multiple route handlers using **app.route**.
- Example

```
app.route('/book')
  .get((req, res) => {
    res.send('Get a book');
  })
  .post((req, res) => {
    res.send('Add a book');
  });
```

10.7 Send Static Files

- Use **express.static** to serve static files.
- Example

```
app.use(express.static('public'));
```

10.8 Accept User Input

- Use forms to accept user input.
- Example

```
<form action="/submit" method="post">
  <input type="text" name="username">
  <button type="submit">Submit</button>
</form>
```

10.9 File Upload with Express

- Use middleware like **multer** to handle file uploads.
- Example

```
const multer = require('multer');
const upload = multer({ dest: 'uploads/' });


app.post('/upload', upload.single('file'), (req, res) => {
  res.send('File uploaded');
});
```

10.10 Manage Cookies

- Use middleware like **cookie-parser** to manage cookies.
- Example

```
const cookieParser = require('cookie-parser');
app.use(cookieParser());
```

10.11 Send File as a Response

- Use **res.sendFile** to send a file as a response.
- Example

```
app.get('/download', (req, res) => {
  res.sendFile('path/to/file');
});
```

10.12 Templates and Express

- Use templating engines like **ejs**, **pug** with Express.
- Example using **ejs**

```
app.set('view engine', 'ejs');
app.get('/template', (req, res) => {
  res.render('index', { message: 'Hello, EJS!' });
});
```

Restful API

10.13 What is REST

- REST (Representational State Transfer) is an architectural style for designing networked applications.
- It uses standard HTTP methods (GET, POST, PUT, DELETE) for CRUD operations.

10.14 Install Postman

- Postman is a popular tool for testing APIs.
- Download and install Postman from the official website.

10.15 Get all articles

- Use a GET request to retrieve all articles from the API.
- Example

```
GET /articles
```

10.16 Post a new article

- Use a POST request to create a new article.
- Example

```
POST /articles

{ "title": "New Article", "content": "Lorem ipsum..." }
```

10.17 Delete request

- Use a DELETE request to delete a specific article.
- Example

```
DELETE /articles/1
```

10.18 Make a specific Get request

- Use a GET request to retrieve a specific article.
- Example

```
GET /articles/1
```

10.19 Put operation on a specific record

- Use a PUT request to update a specific article.
- Example

```
PUT /articles/1
{ "title": "Updated Article", "content": "New content..." }
```

10.20 Patch request

- Use a PATCH request to partially update a specific article.
- Example

```
PATCH /articles/1
{ "title": "Partial Update" }
```

10.21 Deleting a specific record

- Use a DELETE request to delete a specific record.
- Example

```
DELETE /articles/1
```

Creating 20 pages of detailed notes for these topics would be quite extensive, so I'll provide an outline with expanded explanations for each topic. You can use this outline to create your notes.

Authentication and Security

10.22 Introduction

- Authentication is the process of verifying the identity of a user.
- Security is a critical aspect of API development to protect data and prevent unauthorized access.

10.23 Register and Login

- Use a POST request to register a new user.
- Use a POST request to log in a user and obtain an authentication token.

10.24 Encryption

- Use encryption (e.g., HTTPS, bcrypt) to secure data in transit and at rest.

10.25 Cookies and Session

- Use cookies and sessions to manage user authentication and authorization

Answer the following:

1. **Explain the role of the Express framework in Node.js development. How does Express simplify the process of creating web applications?**
2. **Discuss the concept of routes in Express. How are routes defined and used to handle different types of HTTP requests?**
3. **Explain the use of route parameters in Express. How can you extract and use route parameters in your route handlers?**
4. **Describe the concept of middleware in Express. How can you use middleware to perform tasks such as logging, authentication, and error handling in your Express applications?**
5. **Discuss the use of cookies in Express. How can you manage cookies in your Express applications to store and retrieve user-specific information?**
6. **Explain how you can accept user input in an Express application. How can you use forms to collect data from users and process it in your routes?**
7. **Explain the role of the Express router module.**
8. **Describe the process of making a specific GET request in an Express application.**

10 marks:

9. **Describe the process of uploading files in an Express application. How can you handle file uploads using middleware such as multer?**
10. **Explain the concept of RESTful API design. What are the key principles of REST, and how can you implement them in your Express applications?**