Introduction & Setup

i. Node.js is a **JavaScript runtime** built on **Chrome's V8 engine**.
ii. It enables JavaScript to run **outside the browser**.
iii. Uses a **non-blocking, event-driven architecture** for handling multiple tasks efficiently.

Install npm

```
npm init -y
```

Practice

1. Install Node.js and check the version:

```
node -v

npm -v
```

2. Write a simple script (app.js):

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

3. Run it in the terminal:

```
node app.js
```

Core Concepts & Global Objects

i. **Global Objects**: Available in all Node.js modules.
ii. setTimeout(): Executes a function after a given delay.
iii. setInterval(): Executes a function repeatedly after a fixed interval.
iv. _dirname: Directory of the current module.
v. _filename: Filename of the current module.

Practice

1. Write a script using global objects (global.js):

```
console.log("Current Directory:", __dirname);
console.log("Current File:", __filename);
setTimeout(() => console.log("Hello after 2 seconds"), 2000);
setInterval(() => console.log("Repeating every 3 seconds"), 3000);
```

2. Run it in the terminal:

```
Node global.js
```

Modules & require()

- Creating and importing modules
- module.exports and require()

Notes

- **Built-in Modules**: fs, http, os, path, etc.
- **Custom Modules**: You can create your own modules.

Practice

1. Create a module (math.js):

```
function add(a, b) {
   return a + b;
}
module.exports = add;
```

2. Use it in another file (app.js):

```
const add = require('./math');
console.log(add(5, 3)); // Output: 8
```

3. Run:

```
node app.js
```

File System (fs module)

- **fs.readFile():** Reads a file asynchronously.
- **fs.writeFile():** Writes data to a file.
- **fs.appendFile():** Appends data to a file.

Practice

1. Read a file (fileReader.js):

```
const fs = require('fs');
fs.readFile('data.txt', 'utf8', (err, data) => {
   if (err) throw err;
   console.log(data);
});
```

2. Write to a file (fileWriter.js):

```
const fs = require('fs');
fs.writeFile('output.txt', 'Hello, Node.js!', (err) => {
   if (err) throw err;
```

```
    console.log("File written successfully");
});
```

HTTP Module (Creating a Server)

- Creating a basic HTTP server
- Handling requests & responses

Notes

- **http.createServer()**: Creates an HTTP server.
- **res.writeHead()**: Sets response headers.
- **res.end()**: Ends the response.

Practice

1. Create a server (server.js):

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

2. Run the server:

```
node server.js
```

3. Open http://localhost:3000 in your browser.

Express.js (Web Framework)

- Installing & setting up Express
- Creating routes

Notes

- Express simplifies building web servers in Node.js.
- **app.get()**: Defines GET routes.
- **app.listen()**: Starts the server.

Practice

1. Install Express:

```
npm install express
```

2. Create a simple Express server (server.js):

```
const express = require('express');
const app = express();
app.get('/', (req, res) => res.send('Hello Express!'));
app.listen(3000, () => console.log('Server running on port 3000'));
```

3. Run and test in the browser.

```
http://localhost:3000/
```

Working with JSON & APIs

- Handling JSON data in Node.js
- Making HTTP requests using fetch() and axios
- Creating a simple REST API

Notes

- JSON (**JavaScript Object Notation**) is a lightweight format for data exchange.
- **Express.js** makes it easy to build REST APIs.
- Use axios for making API requests.

Practice

1. Create a simple REST API (api.js):

```
const express = require('express');
const app = express();
app.use(express.json());

let users = [{ id: 1, name: "Deepak" }];

app.get('/users', (req, res) => res.json(users));
app.post('/users', (req, res) => {
  users.push({ id: users.length + 1, name: req.body.name });
  res.send("User added!");
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

2. Test with Postman or a browser.

Express Router & Middleware

1. Express Router

Express.js provides a built-in router to help you manage routes efficiently, especially in large applications. The router allows you to group related routes and export them as separate modules.

2. Middleware in Express

Middleware functions are functions that execute during the request-response cycle. They have access to the req and res objects and can modify them before sending a response.

Types of Middleware

1. **Application-Level Middleware** – Applies to all requests.
2. **Router-Level Middleware** – Specific to an Express Router.
3. **Built-in Middleware** – Provided by Express (e.g., express.json()).
4. **Third-Party Middleware** – Installed via npm (e.g., cors).

Notes

- Middleware functions **modify requests** before they reach route handlers.
- Routers help in **structuring large applications**.

Practice

1. Create a middleware (middleware.js):

```
function logger(req, res, next) {
   console.log(`${req.method} ${req.url}`);
   next(); // Move to the next middleware
}

module.exports = logger;
```

2. Use it in Express (server.js):

```
const express = require('express');
const logger = require('./middleware');
const app = express();
app.use(logger);
app.get('/', (req, res) => res.send('Middleware in action!'));
app.listen(3000, () => console.log('Server running on port 3000'));
```