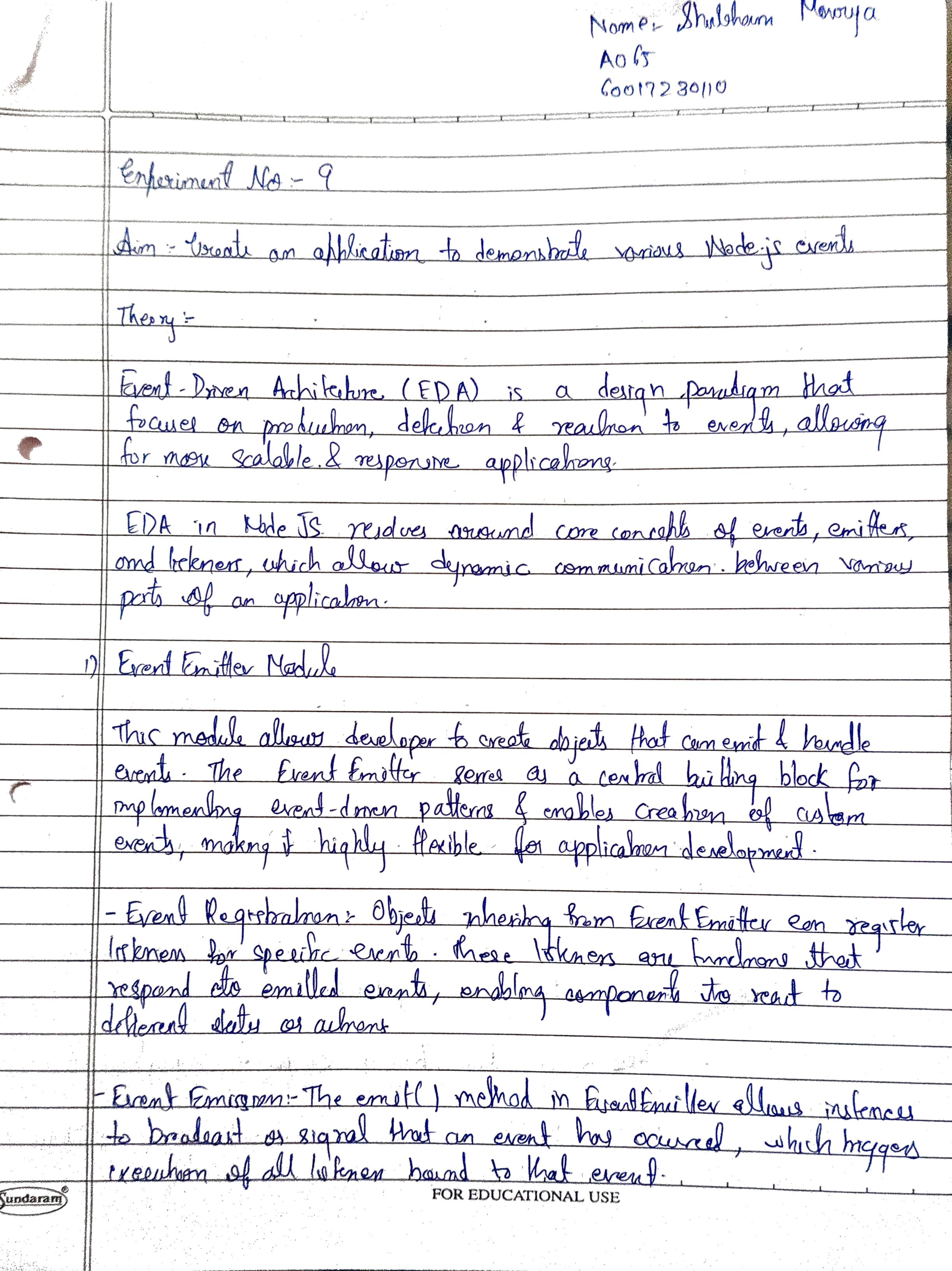
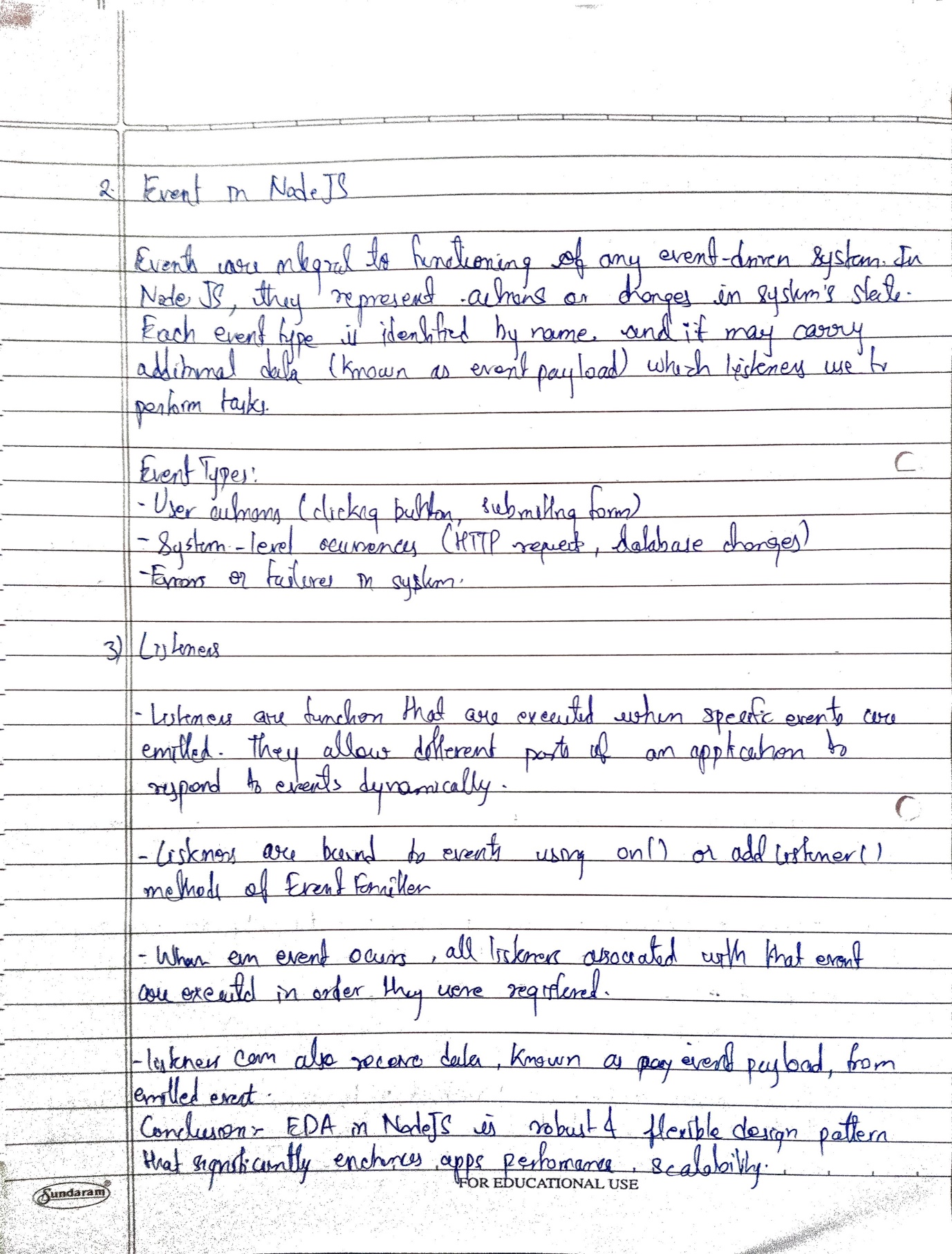
**Experiment 9**

**Name: Shubham Mourya SAP ID: 60017230110**





|  |  |
| --- | --- |
| **Date:** | **08/10/2024 and 10/10/2024** |
| Aim | **Create an application to demonstrate various Node.js Events** |
| Software |  |
| Pre-requisite | Active internet connection |
| Theory | Event-Driven Architecture (EDA) has emerged as a powerful paradigm for building scalable, responsive, and loosely coupled systems. In Node.js, EDA plays a pivotal role, leveraging its asynchronous nature and event-driven capabilities to create efficient and robust applications. Let's delve into the intricacies of Event-Driven Architecture in Node.js exploring its core concepts, benefits, and practical examples. Key Components in Node.js Event-Driven Architecture: **1. EventEmitter Module:** At the heart of Node.js event-driven architecture lies the EventEmitter module, which enables the creation of objects that can emit events and handle them. It serves as a foundational building block for implementing event-driven patterns within applications. Key aspects of the EventEmitter include:   * ***Event Registration:*** Objects that inherit from EventEmitter can register event listeners for specific events they are interested in. This registration involves associating a function (listener) with a particular event name. * **Event Emission:** The emit() method within the EventEmitter allows instances to emit events, signalling that a specific action or state change has occurred. This triggers the invocation of all registered listeners for that particular event. * **Custom Events:** Developers can create custom events in their applications, defining unique event names to represent various actions or occurrences within the system.   const EventEmitter = require('events');  class MyEmitter extends EventEmitter {}  const myEmitter = new MyEmitter();  // Event listener for 'customEvent'  myEmitter.on('customEvent', (arg1, arg2) => {  console.log('Event received with arguments:', arg1, arg2);  });  // Emitting the 'customEvent'  myEmitter.emit('customEvent', 'Hello', 'World');  n this example, a custom MyEmitter class is created, inheriting from EventEmitter. An event listener is added for the event "customEvent", which logs the received arguments when the event is emitted using emit().  **2. Events:** In Node.js, events are fundamental occurrences that are recognized and handled within an application. They encapsulate specific actions or changes in the system's state. Key aspects of events include:   * ***Event Types:*** Events can encompass a wide range of actions or changes, such as data updates, user interactions, system errors, or lifecycle events. * ***Event Naming:*** Events are typically identified by strings that represent their nature or purpose. Well-defined and descriptive event names facilitate better understanding and maintainability within the codebase. * ***Event Payload:*** Events can carry additional data or information, known as the event payload. This data can be passed along when emitting events and can be utilized by listeners to perform specific actions based on the event context.   const http = require('http');  const server = http.createServer((req, res) => {  if (req.url === '/home') {  res.writeHead(200, { 'Content-Type': 'text/plain' });  res.end('Welcome to the home page!');  } else if (req.url === '/about') {  res.writeHead(200, { 'Content-Type': 'text/plain' });  res.end('About us page.\n');  } else {  res.writeHead(404, { 'Content-Type': 'text/plain' });  res.end('Page not found!');  }  });  // Listening for the 'request' event  server.on('request', (req, res) => {  console.log(`Request received for URL: ${req.url}`);  });  server.listen(3000, () => {  console.log('Server running on port 3000');  });  In this example, the HTTP server emits a "request" event each time it receives a request. The on() method is used to listen to this event, enabling logging of the requested URL.  **3. Listeners:** Listeners are functions associated with specific events that are triggered when the corresponding event is emitted. Key aspects of listeners include:   * ***Event Binding:*** Listeners are bound to events using the on() or addListener() method provided by EventEmitter. They are registered to respond to particular events emitted by an emitter. * ***Execution of Listeners:*** When an event is emitted, all registered listeners for that event are executed sequentially, allowing multiple functions to respond to the same event. * ***Listener Parameters:*** Listeners can receive parameters or the event payload when they are invoked, enabling them to access relevant information associated with the emitted event.   const EventEmitter = require('events');  const myEmitter = new EventEmitter();  // Listener 1 for 'eventA'  myEmitter.on('eventA', () => {  console.log('Listener 1 for eventA executed');  });  // Listener 2 for 'eventA'  myEmitter.on('eventA', () => {  console.log('Listener 2 for eventA executed');  });  // Emitting 'eventA'  myEmitter.emit('eventA');  In this example, two listeners are registered for the "eventA". When the event is emitted using emit(), both listeners are executed sequentially in the order they were registered. Benefits of Event-Driven Architecture in Node.js **1. Asynchronous Processing and Non-Blocking IO:** Node.js known for its asynchronous nature complements EDA seamlessly. EDA leverages this by enabling non-blocking event handling. As events occur, Node.js efficiently manages these events concurrently without waiting for each operation to complete. This approach significantly enhances application performance, as the system can handle multiple tasks simultaneously without getting blocked by I/O operations or other tasks.  **2. Loose Coupling and Modularity:** EDA promotes loose coupling between different components of an application. Components communicate through events, reducing direct dependencies among them. This loose coupling allows for greater modularity, as components can operate independently, making the system more maintainable and easier to extend or modify. Changes to one component generally have minimal impact on others, fostering a more adaptable and scalable architecture.  **3. Scalability and Responsiveness:** Node.js event-driven model contributes significantly to the scalability of applications. The ability to distribute events across multiple listeners or subscribers allows for better load distribution and resource utilization. This scalability ensures that the application remains responsive, even under heavy loads, by efficiently handling concurrent events and requests.  **4. Enhanced Error Handling and Resilience:** EDA facilitates robust error handling within Node.js applications. By emitting specific error events, components can communicate failures or exceptional conditions, allowing other parts of the system to respond accordingly. This enhances the application's resilience by providing a structured way to handle errors and recover from unexpected situations.  **5. Real-time Communication and Event-Driven Data Flow:** In scenarios requiring real-time communication or data flow, such as chat applications or IoT systems, EDA in Node.js excels. The event-driven approach allows for seamless communication between different parts of the system in real-time. Events can propagate updates or changes across the system, ensuring that all relevant components are notified and can react promptly.  **6. Flexibility and Extensibility:** EDA fosters a flexible architecture that accommodates future changes and extensions. New functionalities or features can be added by introducing new events or listeners without disrupting the existing components. This extensibility ensures that the system can evolve over time to meet changing requirements without significant architectural overhauls. Examples **1: Real-Time Chat Application** Imagine building a real-time chat application using Node.js and Socket, where multiple users can exchange messages instantly. Here's a simplified demonstration.  const http = require('http');  const express = require('express');  const socketIO = require('socket.io');  const app = express();  const server = http.createServer(app);  const io = socketIo(server);  // Event handler for WebSocket connections  io.on('connection', (socket) => {  // Event handler for incoming messages  socket.on('message', (message) => {  // Broadcasting the received message to all connected clients except the sender  socket.broadcast.emit('message', message);  });  });  server.listen(8080, () => {  console.log('Server running on port 8080');  });  In this example, the SocketIO server (an instance of SocketIO Server) listens for connections. When a client connects, an event is emitted. Subsequently, the server listens for incoming messages from clients, emitting the ‘message' event. The server broadcasts received messages to all connected clients, ensuring real-time communication between multiple users.  **2: Event-Driven File System Monitoring** Consider a scenario where you need to monitor a directory for file changes using Node.js.  const fs = require('fs');  const EventEmitter = require('events');  class FileWatcher extends EventEmitter {  watchDir(directory) {  fs.watch(directory, (eventType, filename) => {  if (eventType === 'change') {  this.emit('fileChanged', filename);  }  });  }  }  In this example, an instance of FileWatcher, which extends EventEmitter, is created. It watches a specified directory for file changes using Node.js' fs.watch() method. When a ‘change' event occurs in the directory, the watcher emits a ‘fileChanged' event. An event listener is set up to handle this event by logging the filename that has been changed.  **3: HTTP Request Handling with Express.js** Let's expand on the HTTP server example using Express.js to handle incoming requests.  const express = require('express');  const app = express();  // Event handler for GET request to the home route  app.get('/', (req, res) => {  res.send('Welcome to the home page!');  });  // Event handler for GET request to other routes  app.get('\*', (req, res) => {  res.status(404).send('Page not found!');  });  // Start the server  const server = app.listen(3000, () => {  console.log('Server running on port 3000');  });  // Event listener for server start event  server.on('listening', () => {  console.log('Server started!');  });const wss = new WebSocket.Server({ port: 8080 });  In this example, Express.js which itself utilizes event-driven patterns is used to define routes and handle incoming HTTP requests. When a GET request is made to the home route (‘/') express emits a ‘request' event. Similarly for other routes, a ‘request' event is emitted. Additionally, the server emits a ‘listening' event when it starts, allowing for event-driven handling of server startup. |
| Code | **App.jsx**  import React, { useState, useEffect, useMemo } from 'react';  import { io } from 'socket.io-client';  const App = () => {    const [message, setMessage] = useState('');    const [receivedMessages, setReceivedMessages] = useState([]);    const [room, setRoom] = useState('');    const [socketID, setSocketID] = useState('');    const [targetSocketID, setTargetSocketID] = useState('');    const [joinedRoom, setJoinedRoom] = useState('');    const socket = useMemo(() => io('http://localhost:3000'), []);    useEffect(() => {      socket.on('connect', () => {        console.log('Connected to server', socket.id);        setSocketID(socket.id);      });      socket.on('receive-message', (data) => {        console.log('Received message:', data);        setReceivedMessages((prevMessages) => [...prevMessages, data]);      });      socket.on('room-joined', (roomName) => {        console.log(`Successfully joined room: ${roomName}`);        setJoinedRoom(roomName);      });      return () => {        socket.disconnect();      };    }, [socket]);    const handleChange = (e) => {      setMessage(e.target.value);    };    const handleRoomChange = (e) => {      setRoom(e.target.value);    };    const handleTargetSocketChange = (e) => {      setTargetSocketID(e.target.value);    };    const handleJoinRoom = (e) => {      e.preventDefault();      socket.emit('join-room', room);      console.log(`Requested to join room: ${room}`);      setRoom('');    };    const handleSubmit = (e) => {      e.preventDefault();      if (targetSocketID) {        socket.emit('message-to-socket', { message, targetSocketID });        console.log(`Sent message to socket: ${targetSocketID}`);      } else if (joinedRoom) {        socket.emit('message-to-room', { message, room: joinedRoom });        console.log(`Sent message to room: ${joinedRoom}`);      } else {        socket.emit('broadcast-message', { message });        console.log('Sent broadcast message');      }      setMessage('');    };    return (      <div className="flex flex-col items-center justify-center min-h-screen">        <h1 className="text-red-600 text-4xl text-center m-4">Learning Socket.io</h1>        <h2 className="text-lg">Your Socket ID: {socketID}</h2>        {joinedRoom && <h3 className="text-lg">Current Room: {joinedRoom}</h3>}        <form onSubmit={handleJoinRoom} className="flex flex-col items-center space-y-4">          <input            type="text"            placeholder="Enter room name..."            value={room}            onChange={handleRoomChange}            className="border border-gray-300 rounded px-4 py-2"          />          <button type="submit" className="bg-green-500 text-white px-4 py-2 rounded">            Join Room          </button>        </form>        <form onSubmit={handleSubmit} className="flex flex-col items-center space-y-4 mt-4">          <input            type="text"            placeholder="Type your message..."            value={message}            onChange={handleChange}            className="border border-gray-300 rounded px-4 py-2"          />          <input            type="text"            placeholder="Enter target socket ID..."            value={targetSocketID}            onChange={handleTargetSocketChange}            className="border border-gray-300 rounded px-4 py-2"          />          <button type="submit" className="bg-blue-500 text-white px-4 py-2 rounded">            Send Message          </button>        </form>        <div className="mt-6">          <h2 className="text-2xl">Messages:</h2>          <ul>            {receivedMessages.map((msg, index) => (              <li key={index} className="mt-2 text-lg">                {msg}              </li>            ))}          </ul>        </div>      </div>    );  };  export default App;  **index.js**  import express from 'express';  import { Server } from 'socket.io';  import { createServer } from 'http';  import cors from 'cors';  const port = 3000;  const app = express();  const server = createServer(app);  app.use(cors({      origin: 'http://localhost:5173',      methods: ['GET', 'POST'],      credentials: true,  }));  const io = new Server(server, {      cors: {          origin: 'http://localhost:5173',          methods: ['GET', 'POST'],      }  });  io.on('connection', (socket) => {      console.log('User connected');      console.log('ID:', socket.id);      socket.emit('user-id', socket.id);      socket.on('join-room', (room) => {          socket.join(room);          console.log(`User ${socket.id} joined room: ${room}`);          socket.emit('room-joined', room);      });      socket.on('message-to-room', ({ message, room }) => {          console.log(`Message from ${socket.id} to room: ${room}`);          io.to(room).emit('receive-message', message);      });      socket.on('message-to-socket', ({ message, targetSocketID }) => {          console.log(`Message from ${socket.id} to ${targetSocketID}: ${message}`);          io.to(targetSocketID).emit('receive-message', message);      });      socket.on('broadcast-message', ({ message }) => {          console.log(`Broadcast message from ${socket.id}: ${message}`);          socket.broadcast.emit('receive-message', message);      });      socket.on('disconnect', () => {          console.log('User disconnected');      });  });  app.get('/', (req, res) => {      res.send('Hello World');  });  server.listen(port, () => {      console.log(`Server is running on port ${port}`);  }); |
| Result |  |
| Conclusion | Event-Driven Architecture in Node.js provides a multitude of benefits that empower developers to create high-performing, scalable and responsive applications. By leveraging asynchronous processing, loose coupling, scalability, and real-time communication, EDA enhances the overall architecture's robustness, flexibility, and ability to handle complex tasks efficiently. |