**Event-Driven Architecture in Node.js**

**1. What is Event-Driven Architecture?**

Event-Driven Architecture (EDA) is a **design pattern** where system components communicate by **emitting** and **listening** for events. Instead of executing code sequentially, the system reacts to events asynchronously.

**2. How Node.js Uses Event-Driven Architecture?**

Node.js is **event-driven** at its core, using the **Event Loop** to handle non-blocking I/O. It relies on the **EventEmitter** class to emit and handle events.

**3. Understanding EventEmitter in Node.js**

The events module in Node.js provides the EventEmitter class, which allows objects to:

* **Emit events** (trigger an event).
* **Listen for events** (execute a function when the event occurs).

**Example 1: Basic EventEmitter Usage**

const EventEmitter = require('events');

// Create an instance of EventEmitter

const eventEmitter = new EventEmitter();

// Define an event listener

eventEmitter.on('greet', (name) => {

    console.log(`Hello, ${name}!`);

});

// Emit the event

eventEmitter.emit('greet', 'Alice');

//  Output:

// Hello, Alice!

✅ The **emit** method triggers the **greet** event, which executes the event listener.

**4. EventEmitter Methods**

| **Method** | **Description** |
| --- | --- |
| on(event, callback) | Registers an event listener. |
| emit(event, args...) | Emits (triggers) an event. |
| once(event, callback) | Registers a listener that runs only once. |
| removeListener(event, callback) | Removes a specific listener. |
| removeAllListeners(event) | Removes all listeners for an event. |

**5. Example 2: once() - Event That Runs Only Once**

const EventEmitter = require('events');

const eventEmitter = new EventEmitter();

eventEmitter.once('welcome', () => {

    console.log('Welcome! This message appears only once.');

});

eventEmitter.emit('welcome');

eventEmitter.emit('welcome');  // This won't trigger again

//  Output:

// Welcome! This message appears only once.

✅ The once() method ensures the listener runs only **one time**.

**6. Example 3: Removing an Event Listener**

const EventEmitter = require('events');

const eventEmitter = new EventEmitter();

const handler = () => console.log('Event triggered!');

eventEmitter.on('test', handler);

eventEmitter.emit('test');  // Runs

eventEmitter.removeListener('test', handler);

eventEmitter.emit('test');  // Won't run

✅ The event listener is removed after the first execution.

**7. Real-World Use Cases**

**1. Event-Driven HTTP Server**

Node.js web servers use event-driven architecture. Example with http:

const http = require('http');

const server = http.createServer();

server.on('request', (req, res) => {

    console.log(`Received request: ${req.method} ${req.url}`);

    res.end('Hello, World!');

});

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

✅ The server **listens for request events** and responds when a request arrives.

**2. Real-Time Chat with WebSockets**

WebSockets (socket.io) rely on event-driven programming for real-time communication:

const io = require('socket.io')(3000);

io.on('connection', (socket) => {

    console.log('New client connected');

    socket.on('message', (msg) => {

        console.log(`Message received: ${msg}`);

        io.emit('message', msg);  // Broadcast to all clients

    });

    socket.on('disconnect', () => {

        console.log('Client disconnected');

    });

});

✅ Messages are sent and received **asynchronously** using events.

**3. Event-Driven Logging System**

const EventEmitter = require('events');

class Logger extends EventEmitter {

    log(message) {

        console.log(`LOG: ${message}`);

        this.emit('log', { message, timestamp: Date.now() });

    }

}

const logger = new Logger();

logger.on('log', (data) => console.log('Log event received:', data));

logger.log('User logged in');

✅ The logging system **emits a log event** whenever a message is logged.

**8. Advantages of Event-Driven Architecture**

✅ **High Performance** – Non-blocking, asynchronous execution.  
✅ **Scalability** – Can handle thousands of concurrent connections.  
✅ **Loose Coupling** – Components are independent and communicate via events.  
✅ **Flexibility** – Easily extend functionalities by adding new event listeners.

**9. Best Practices for Event-Driven Architecture**

✔ **Use once() for events that should run only once** (e.g., connection established).  
✔ **Avoid too many event listeners** to prevent memory leaks (setMaxListeners() can help).  
✔ **Always remove unused listeners** to free up resources (removeListener(), removeAllListeners()).  
✔ **Use event-driven architecture for real-time apps** like chat apps, notifications, or WebSockets.

**What’s Next?**

Would you like to explore:

* **How Event Loop & Event-Driven Architecture Work Together?**
* **Performance Optimization in Event-Driven Apps?**
* **Event-Driven Microservices Architecture?**

Let me know what interests you!