Node.js Event-Driven Architecture and EventEmitter

1. Understanding Node's Event-Driven Framework

Concept

Node.js uses an **event-driven architecture**:

- Your application listens for **events** and runs **callbacks** (also called **event handlers**) when those events occur.
- Non-blocking I/O allows Node to handle many operations concurrently without creating multiple threads.

How it Works

1. You register a **listener** for an event.
2. When the event occurs, Node's **event loop** executes the listener callback.

Why it matters: It enables building highly scalable applications such as chat servers, real-time dashboards, and streaming services.

Example: Simple Event Flow

```
// events-demo.js
const EventEmitter = require('events');
const emitter = new EventEmitter();

// 1. Register an event listener
emitter.on('greet', () => {
  console.log('Hello from Node event-driven framework!');
});

// 2. Emit the event
emitter.emit('greet');
```

Run:

```
node events-demo.js
```

Output:

```
Hello from Node event-driven framework!
```

Explanation:

- `emitter.on('greet', ...)` registers a listener.
- `emitter.emit('greet')` triggers the event.

2. The EventEmitter Class

`EventEmitter` is provided by Node's built-in `events` module. It is the core of the event-driven system.

Common Methods

Method	Purpose
`.on(event, listener)`	Add an event listener
`.emit(event, [...args])`	Trigger an event
`.once(event, listener)`	Add a listener that runs only once
`.removeListener(event, listener)` or `.off(event, listener)`	Remove a specific listener
`.removeAllListeners([event])`	Remove all listeners for an event

Example: Passing Arguments

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

emitter.on('order', (item, qty) => {
  console.log(`Order received: ${qty} x ${item}`);
});

emitter.emit('order', 'Coffee', 2);
```

Output:

```
Order received: 2 x Coffee
```


Use Case: Logging Events in a Web Server

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

const logger = new EventEmitter();

// Log every request
logger.on('log', (message) => {
  console.log('LOG:', message);
});

http.createServer((req, res) => {
  logger.emit('log', `${req.method} ${req.url}`);
  res.end('Request logged!');
}).listen(3000, () => console.log('Server running on http://localhost:3000'));
```

This approach cleanly separates logging logic using events.

3. Events and the Event Loop

What is the Event Loop?

The **event loop** is the heart of Node.js' non-blocking architecture:

- It continuously checks for **tasks**, **callbacks**, and **events**.
- When an I/O operation finishes, its callback is queued for execution.

Key Phases

1. **Timers**: Executes `setTimeout` and `setInterval` callbacks.
2. **Pending Callbacks**: Executes system-related callbacks.
3. **Poll**: Retrieves new I/O events and executes I/O callbacks.
4. **Check**: Executes `setImmediate` callbacks.
5. **Close callbacks**: Executes `close` event callbacks.

In practice: *You rarely manage the loop directly—Node handles it automatically. Understanding it helps when debugging performance or async behavior.*

Example: Order of Execution

```
setTimeout(() => console.log('Timeout callback'), 0);
setImmediate(() => console.log('Immediate callback'));
console.log('Synchronous log');
```

Typical Output:

```
Synchronous log
Immediate callback
Timeout callback
```

(Actual order between Immediate and Timeout may vary slightly, but synchronous logs always run first.)

4. Inheriting Events

You can create **custom classes** that emit events by **extending EventEmitter**.

Example: Custom Class

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

// Custom class
class OrderService extends EventEmitter {
  placeOrder(item) {
    console.log(`Placing order for ${item}`);
    // Emit event when order is placed
    this.emit('orderPlaced', item);
  }
}

// Usage
const service = new OrderService();

// Listener
service.on('orderPlaced', (item) => {
  console.log(`Order confirmed for: ${item}`);
});

// Trigger event via method
service.placeOrder('Pizza');
```

Output:

```
Placing order for Pizza
Order confirmed for: Pizza
```


Use Case: Chat Application

- **Class:** `ChatRoom` inherits EventEmitter.
- **Event:** `message`.
- **When user sends a message:** emit `message`.
- **Other users:** listen for `message` to display it in real-time.

This pattern powers real-time applications like messaging, notifications, and live updates.

5. Summary Table

Concept	Key API/Feature	Practical Use
Event-Driven Framework	`on`, `emit`	Real-time apps, async I/O
EventEmitter Class	`events` module	Create and listen to custom events
Event Loop	Built-in mechanism	Handles callbacks and I/O
Inheriting Events	`class extends EventEmitter`	Build reusable event-driven components

6. Best Practices

- Use `once` for events that must run only one time (e.g., database connection).

- Remove listeners with `.off()` or `.removeListener()` to prevent memory leaks.
- Keep event names descriptive (`'userLoggedIn'` instead of `'u1'`).

12/12

Final Thoughts

Node.js' **event-driven architecture** makes it ideal for applications requiring high concurrency and real-time communication. By understanding the **EventEmitter** class, **event loop**, and **inheritance**, you can build scalable, maintainable applications such as **chat servers, streaming services, and real-time dashboards**.