NODE

Let's go deep into **Node.js** and **Express** at a **senior level** — covering advanced concepts, architecture patterns, performance optimizations, scalability, and anticipated **senior-level interview questions** with strong answers.

🚀 1. Core Node.js Concepts

Node.js is a **runtime environment** that allows JavaScript to run on the server side. It's built on **Chrome's V8 engine** and provides an **event-driven**, **non-blocking** I/O model using the **libuv** library.

✅ a) Event Loop

- The event loop allows Node.js to handle multiple operations asynchronously in a single thread.
- The event loop operates in **phases**:
  1. **Timers Phase** – Executes `setTimeout()` and `setInterval()` callbacks.
  2. **Pending Phase** – Handles I/O callbacks.
  3. **Idle/Prepare Phase** – Internal operations.
  4. **Poll Phase** – Handles new I/O events and executes synchronous callbacks.
  5. **Check Phase** – Executes `setImmediate()` callbacks.
  6. **Close Phase** – Executes close callbacks (e.g., socket close).

👉 Example: Event Loop in Action

```
setTimeout(() => console.log('timeout'), 0);
setImmediate(() => console.log('immediate'));

process.nextTick(() => console.log('nextTick'));
```

Output:

```
nextTick
immediate
```

```
timeout
```

👉 **Why?**

- `process.nextTick()` runs before the event loop continues to the next phase.
- `setImmediate()` runs during the **check phase**.
- `setTimeout()` runs during the **timers phase**.

✅ b) Non-Blocking I/O

- Node.js uses the **libuv** library to offload I/O tasks to a thread pool (in the background).
- Non-blocking I/O allows Node.js to handle thousands of requests without creating new threads.

👉 **Example:**

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

fs.readFile('file.txt', (err, data) => {
  if (err) throw err;
  console.log(data.toString());
});

console.log('Reading file...');
```

Output:

```
Reading file...
<file content>
```

➡️ `fs.readFile()` is asynchronous — it offloads file reading to the thread pool.

✅ c) Streams

- Streams handle data in **chunks** rather than loading entire content into memory.
- Types of streams:
  - **Readable** – Data flows from source to Node.js.

- **Writable** – Data flows from Node.js to destination.
- **Duplex** – Acts as both readable and writable (e.g., sockets).
- **Transform** – Modifies data as it passes through.

👉 **Example: Create a Readable Stream**

```
const fs = require('fs');
const stream = fs.createReadStream('file.txt');

stream.on('data', chunk => {
  console.log(chunk.toString());
});
```

➡️ Efficient for large files since data is processed in chunks.

✅ d) Worker Threads

- Worker Threads enable multi-threading in Node.js.
- Suitable for **CPU-bound** tasks like encryption or data processing.

👉 **Example:**

```
const { Worker } = require('worker_threads');

const worker = new Worker('./worker.js');
worker.on('message', msg => console.log(msg));
```

➡️ Unlike the event loop, Worker Threads execute tasks in **parallel**.

⚙️ 2. Express.js Core Concepts

Express.js is a **lightweight, unopinionated framework** for Node.js used to build REST APIs and web servers.

✅ a) Routing

Express provides a simple way to define routes:

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

app.get('/users', (req, res) => {
  res.send('User list');
});

app.listen(3000);
```

✅ b) Middleware

Middleware functions sit between the request and response lifecycle.

Type	Description
Application-Level	`app.use()` applies to all routes
Router-Level	`router.use()` applies to specific routes
Error-Handling	`next(err)` handles errors
Third-Party	Examples: `cors`, `helmet`, `body-parser`

👉 Example:

```
app.use((req, res, next) => {
  console.log('Middleware triggered');
  next();
});
```

✅ c) Error Handling

Express uses a **centralized error-handling** mechanism:

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

✅ d) Request Handling Lifecycle

1. **Request received**
2. **Middleware execution**
3. **Route handling**
4. **Response sent**

✅ e) Performance Tuning

- Use **gzip compression** (`compression` library).
- Use **Redis or Memcached** for caching.
- Minify response payload using `express-minify`.
- Use **HTTP/2** for better multiplexing.

🏆 3. Advanced Topics

✅ a) Load Balancing

- Use **NGINX** or **HAProxy** for load balancing.
- Use **PM2** to run multiple Node.js processes on a single machine.
- Use **sticky sessions** if session consistency is required.

✅ b) Security

- Use **Helmet** to secure HTTP headers.
- Prevent NoSQL Injection using `mongoose.sanitizeFilter()`.
- Validate input using `express-validator`.
- Protect against CSRF using `csurf`.
- Use OAuth 2.0 + JWT for authentication.

✅ c) Rate Limiting

Use `express-rate-limit` to prevent abuse:

```
const rateLimit = require('express-rate-limit');

const limiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 100
});

app.use(limiter);
```

🔥 4. Anticipated Senior-Level Interview Questions

1. How does the Node.js event loop work?

✅ Explain event loop phases and how async callbacks are handled.

---

2. How would you handle a high CPU-bound task in Node.js?

✅ Use **Worker Threads** for parallel execution.

---

3. How can you prevent memory leaks in a Node.js application?

✅ Use:

- `WeakMap` for caching
- Close open handles
- Monitor event emitter leaks

---

4. What are the advantages and disadvantages of using Node.js for backend development?

✅ Advantages:

- Fast I/O handling
- Large ecosystem
- Single programming language (JavaScript)

✅ Disadvantages:

- Single-threaded model limits CPU-bound tasks
- Callback hell without async/await

---

5. How would you implement caching in an Express app?

✅ Use **Redis** or **Memory Cache** for low-latency responses.

---

6. How would you handle database connection pooling in Node.js?

✅ Use `pg-pool` (PostgreSQL) or `mongoose-pool` for MongoDB.

---

7. How would you improve the performance of an Express API handling 100K+ requests per second?

✅ Use:

- Load balancing
- Redis for caching
- Connection pooling
- Optimize query execution

---

8. How does Node.js handle backpressure?

✅ Use:

- **Streams**
- Throttling requests
- Load shedding strategies

---

9. How would you handle cross-origin requests in Express?

✅ Use `cors`:

```
const cors = require('cors');
app.use(cors());
```

---

10. How would you design a real-time chat application using Node.js and Express?

✅ Use:

- **WebSockets** for real-time communication
- **Redis** for session storage
- **Rate limiting** for abuse protection

🎯 Summary

✅ Mastering **Event Loop**, **Streams**, **Middleware**, **Load Balancing**, **Security**, and **Performance Tuning** is key at a senior level.
✅ Prepare for in-depth architecture and scalability questions.
✅ Focus on **how Node.js works under the hood** and how to **optimize Express** for high throughput.

💪 Ready to dive deeper into any of these? 😎

Below is an **end-to-end** guide on **Express.js**—from beginner concepts all the way to advanced patterns. We'll start with the foundations, gradually build up to more complex topics, and illustrate concepts with examples. Grab your text editor and follow along!

---

1. What is Express.js?

Express.js is a **minimalist**, **unopinionated** web framework for **Node.js**. It simplifies the process of creating and managing:

- **HTTP servers**
- **RESTful APIs**
- **Web applications**

Key Features

1. **Routing** – Easy to define URL endpoints and HTTP methods (GET, POST, etc.).
2. **Middleware** – Powerful pattern for handling and transforming incoming requests/responses.
3. **Extensibility** – Large ecosystem of community-maintained middleware for common tasks like security, sessions, and more.
4. **Unopinionated** – Gives you freedom to structure your application your way.

---

2. Setup and Installation

2.1 Prerequisites

- **Node.js** installed (v14.x or above recommended).
- Basic knowledge of JavaScript fundamentals.

2.2 Initializing a Project

1. **Create a folder** for your project:

```
mkdir express-tutorial
cd express-tutorial
```

2. **Initialize Node.js**:

```
npm init -y
```

This command creates a `package.json` with default settings.

3. **Install Express**:

```
npm install express
```

2.3 Hello World Example

Create a file named `index.js`:

```
const express = require('express');
const app = express();
const PORT = 3000;

// Define a simple GET route
app.get('/', (req, res) => {
  res.send('Hello, World from Express!');
});

app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Run your app:

```
node index.js
```

Open your browser at `http://localhost:3000` to see:

```
Hello, World from Express!
```

---

3. Understanding Routing

Routing is about defining how your application responds to various **HTTP requests** (GET, POST, PUT, DELETE, etc.) at different **paths** (URLs).

3.1 Basic Routes

```
app.get('/users', (req, res) => {
  // handle GET request on /users
  res.send('GET request on /users');
});

app.post('/users', (req, res) => {
  // handle POST request on /users
  res.send('POST request on /users');
});
```

3.2 Route Parameters

Route parameters allow you to capture dynamic values in the path:

```
// :id is the route parameter
app.get('/users/:id', (req, res) => {
  const userId = req.params.id;
  res.send(`User ID requested is ${userId}`);
});
```

If you visit `http://localhost:3000/users/42`, the response will be:

```
User ID requested is 42
```

3.3 Query Parameters

Query parameters are appended to the URL after a `?`:

```
app.get('/search', (req, res) => {
  const { q } = req.query; // e.g., /search?q=javascript
  res.send(`You searched for ${q}`);
});
```

Visiting `http://localhost:3000/search?q=javascript` yields:

```
You searched for javascript
```

3.4 Handling Multiple Params

```
app.get('/books/:bookId/chapters/:chapterId', (req, res) => {
  const { bookId, chapterId } = req.params;
  res.send(`You requested chapter ${chapterId} of book ${bookId}`);
});
```

---

4. Middleware

Middleware functions are the heart of Express.js. They are functions that execute **in sequence** during the request-response cycle.

4.1 What is Middleware?

A middleware function has the signature:

```
function middleware(req, res, next) {
  // do something
  next(); // proceed to next middleware or route handler
}
```

- `req` – The request object.
- `res` – The response object.
- `next()` – Passes control to the next middleware.

4.2 Application-Level Middleware

```
app.use((req, res, next) => {
  console.log(`Incoming request: ${req.method} ${req.url}`);
  next();
});
```

This middleware runs for **every** request, logging the method and URL.

4.3 Router-Level Middleware

You can create an **Express Router** and apply middleware only to specific routes:

```
const userRouter = express.Router();

// router-level middleware
```

```
userRouter.use((req, res, next) => {
  console.log('User router middleware');
  next();
});

userRouter.get('/', (req, res) => {
  res.send('Users home');
});

app.use('/users', userRouter);
```

4.4 Error-Handling Middleware

Error-handling middleware has **four** parameters:

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

- Call `next(err)` inside any route or middleware to jump here.

4.5 Third-Party Middleware

Common libraries:

- `cors` – Enable Cross-Origin Resource Sharing.
- `helmet` – Security headers.
- `morgan` – HTTP request logging.
- `compression` – GZIP compression.

Example:

```
const cors = require('cors');
const helmet = require('helmet');
const morgan = require('morgan');
const compression = require('compression');

app.use(cors());
app.use(helmet());
app.use(morgan('combined'));
app.use(compression());
```

5. Serving Static Files

Express can serve static files (images, CSS, JavaScript) with the built-in `express.static` middleware.

```
app.use(express.static('public'));
// public/index.html will be served at http://localhost:3000/index.html
```

6. Templating Engines

While you can serve static HTML, you might want a **dynamic template**. Popular templating engines:

- **Pug** (formerly Jade)
- **EJS** (Embedded JavaScript templates)
- **Handlebars**

Example with **EJS**:

1. Install EJS:

   ```
   npm install ejs
   ```

2. Set up EJS in Express:

   ```
   app.set('view engine', 'ejs');
   ```

3. Create a file `views/index.ejs`:

   ```
   <h1>Hello, <%= name %>!</h1>
   ```

4. Render the view:

```
app.get('/', (req, res) => {
  res.render('index', { name: 'Express' });
});
```

The response will be:

```
<h1>Hello, Express!</h1>
```

---

7. Working with JSON and Forms

7.1 JSON Parsing

By default, Express does not parse JSON automatically; you need the built-in `express.json()` middleware.

```
app.use(express.json());

app.post('/api/data', (req, res) => {
  console.log(req.body); // parsed JSON data
  res.send('Data received');
});
```

7.2 URL-Encoded Form Data

To handle HTML form submissions, use `express.urlencoded()`:

```
app.use(express.urlencoded({ extended: true }));
```

Now `req.body` will have the form data.

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

```
app.post('/submit', (req, res) => {
  console.log(req.body.username);
  res.send('Form submitted successfully!');
});
```

---

8. Cookies and Sessions

8.1 Cookies

Use the `cookie-parser` middleware to read cookies:

1. Install:

   ```
   npm install cookie-parser
   ```

2. Setup:

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

3. Use:

   ```
   app.get('/set-cookie', (req, res) => {
     res.cookie('username', 'John', { httpOnly: true });
     res.send('Cookie set!');
   });

   app.get('/get-cookie', (req, res) => {
     const { username } = req.cookies;
     res.send(`Cookie: ${username}`);
   });
   ```

8.2 Sessions

For **persistent** login sessions, use the `express-session` library:

1. Install:

   ```
   npm install express-session
   ```

2. Setup:

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

app.use(session({
  secret: 'secretKey', // change this
  resave: false,
  saveUninitialized: true,
  cookie: { secure: false } // set secure to true if using HTTPS
}));
```

3. Use:

```
app.get('/login', (req, res) => {
  req.session.user = { name: 'John' };
  res.send('Logged in!');
});

app.get('/profile', (req, res) => {
  if (req.session.user) {
    res.send(`Welcome, ${req.session.user.name}`);
  } else {
    res.send('Not logged in!');
  }
});
```

---

9. File Uploads

Use `multer` for handling file uploads:

1. Install:

```
npm install multer
```

2. Setup:

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

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

In an HTML form, make sure to set `enctype="multipart/form-data"`:

```
<form action="/upload" method="POST" enctype="multipart/form-data">
  <input type="file" name="file" />
  <button type="submit">Upload</button>
</form>
```

---

10. Error Handling

10.1 Centralized Error Handler

```
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({ error: 'Internal Server Error' });
});
```

10.2 Synchronous Errors

Throw an error in a route:

```
app.get('/sync-error', (req, res) => {
  throw new Error('Something broke!');
});
```

Express automatically catches it with the error handler.

10.3 Asynchronous Errors

Use `next(err)` in asynchronous operations:

```
app.get('/async-error', async (req, res, next) => {
  try {
    // some async code
    throw new Error('Async error');
  } catch (err) {
    next(err);
  }
});
```

11. Authentication and Authorization

11.1 JWT (JSON Web Tokens)

1. Install:

   ```
   npm install jsonwebtoken
   ```

2. Generate a token:

   ```
   const jwt = require('jsonwebtoken');

   app.post('/login', (req, res) => {
     // validate user...
     const token = jwt.sign({ userId: 123 }, 'secretKey', { expiresIn: '1h'
   });
     res.json({ token });
   });
   ```

3. Verify token (middleware):

   ```
   function authenticateToken(req, res, next) {
     const authHeader = req.headers['authorization'];
     const token = authHeader && authHeader.split(' ')[1];

     if (!token) return res.sendStatus(401);

     jwt.verify(token, 'secretKey', (err, user) => {
       if (err) return res.sendStatus(403);
       req.user = user;
       next();
     });
   }

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

```
  res.send(`Hello user ${req.user.userId}`);
});
```

11.2 OAuth

- Use strategies like **Passport.js** to handle OAuth providers (Google, Facebook, GitHub, etc.).
- Beyond the scope here, but the main idea is leveraging provider tokens to authenticate users in your Express app.

---

12. Security Best Practices

1. **Use `helmet`** for setting secure HTTP headers.
2. **Escape user inputs** to avoid **XSS** attacks.
3. **Limit request size** using `express.json({ limit: '10kb' })`.
4. **Use HTTPS** in production.
5. **Disable `x-powered-by` header** to hide Express usage:

   ```
   app.disable('x-powered-by');
   ```

---

13. Performance and Scaling

13.1 Clustering / PM2

Node.js runs on a single thread by default. Use **PM2** to **cluster** multiple Node.js processes and utilize multi-core CPUs:

```
npm install pm2 -g
pm2 start index.js -i max
```

- `-i max` starts a process for each CPU core.

13.2 Load Balancing

- Use a **reverse proxy** like **NGINX** or **HAProxy** in front of your Express app to distribute requests among multiple instances.

13.3 Caching

- Caching frequently requested data in **Redis** or **Memcached** can massively improve performance.
- Example using Redis to cache responses:

```
const redis = require('redis');
const client = redis.createClient();

app.get('/data', async (req, res) => {
  const cacheData = await client.get('dataKey');
  if (cacheData) {
    return res.send(JSON.parse(cacheData));
  }

  // fetch data from DB or external API
  const freshData = { message: 'fresh data' };
  client.setEx('dataKey', 60, JSON.stringify(freshData));
  res.send(freshData);
});
```

---

14. Testing

14.1 Setup with Jest and Supertest

1. Install:

```
npm install --save-dev jest supertest
```

2. Configure `package.json`:

```
{
  "scripts": {
    "test": "jest"
  }
}
```

3. Example test file (`app.test.js`):

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

const app = express();
app.get('/test', (req, res) => res.send('Hello Test'));

describe('GET /test', () => {
  it('should respond with Hello Test', async () => {
    const res = await request(app).get('/test');
    expect(res.text).toBe('Hello Test');
    expect(res.statusCode).toBe(200);
  });
});
```

4. Run tests:

```
npm test
```

---

15. Common Design Patterns

15.1 MVC (Model-View-Controller)

Organize your app into:

- **Models** – Business logic/data (Mongoose models or other DB abstractions).
- **Views** – Templating (EJS, Pug, or front-end frameworks).
- **Controllers** – Route logic.

15.2 Service/Repository Pattern

Separate your **data access** (Repositories) from your **business logic** (Services) and your **controllers**. This helps keep your code clean and testable.

---

16. Deployment

16.1 Containerization with Docker

1. **Dockerfile** example:

   ```
   FROM node:18
   WORKDIR /app
   COPY package*.json ./
   RUN npm install
   COPY . .
   EXPOSE 3000
   CMD ["node", "index.js"]
   ```

2. **Build and Run**:

   ```
   docker build -t express-app .
   docker run -p 3000:3000 express-app
   ```

16.2 Cloud Providers

- **AWS EC2** – Deploy your container or Node app on a virtual machine.
- **Heroku** – Simple, straightforward hosting with a free tier (limited resources).
- **Azure / GCP** – Host Node containers or direct Node.js apps.

---

17. Putting It All Together: Example REST API

Here's a mini example illustrating multiple concepts:

```
const express = require('express');
const app = express();
const session = require('express-session');
const cors = require('cors');
const helmet = require('helmet');
const bodyParser = require('body-parser');

// Middleware
app.use(helmet());
app.use(cors());
```

```
app.use(bodyParser.json());
app.use(session({
  secret: 'keyboard cat',
  resave: false,
  saveUninitialized: true,
}));

// Mock data
let users = [
  { id: 1, name: 'John', age: 30 },
  { id: 2, name: 'Jane', age: 25 },
];

// Routes
app.get('/users', (req, res) => {
  res.json(users);
});

app.post('/users', (req, res) => {
  const { name, age } = req.body;
  const newUser = { id: Date.now(), name, age };
  users.push(newUser);
  res.json(newUser);
});

app.get('/users/:id', (req, res) => {
  const userId = parseInt(req.params.id, 10);
  const user = users.find(u => u.id === userId);
  if (!user) return res.status(404).send('User not found');
  res.json(user);
});

app.put('/users/:id', (req, res) => {
  const userId = parseInt(req.params.id, 10);
  const { name, age } = req.body;
  let user = users.find(u => u.id === userId);
  if (!user) return res.status(404).send('User not found');

  user = Object.assign(user, { name, age });
  res.json(user);
});

app.delete('/users/:id', (req, res) => {
  const userId = parseInt(req.params.id, 10);
  users = users.filter(u => u.id !== userId);
  res.send(`User ${userId} deleted`);
});

// Error-Handling
app.use((err, req, res, next) => {
  console.error('Central Error Handler:', err);
  res.status(500).send('Internal Server Error');
});

// Start Server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Example app listening at http://localhost:${PORT}`);
});
```

---

18. Conclusion and Next Steps

1. **Practice, Practice, Practice**
   - Build small CRUD apps to get comfortable with routes, middleware, sessions, and error handling.
2. **Dive into Databases**
   - Use MongoDB (with Mongoose) or PostgreSQL (with `pg`) to store and retrieve real data.
3. **Explore Frontend Integration**
   - Serve a React/Angular/Vue SPA from an Express backend or host them separately.
4. **Master Debugging**
   - Use `console.log()`, Node.js Inspector (`node --inspect`), or Visual Studio Code breakpoints.
5. **Expand Your Toolbox**
   - Look into advanced topics like GraphQL (`apollo-server-express`), microservices, or RabbitMQ for messaging.

Final Thoughts

Express.js is one of the most popular ways to build web servers in Node.js, and it's popular for good reason: it's lightweight, flexible, and backed by a huge ecosystem. Mastering **routing**, **middleware**, **error handling**, **security**, and **scalability** will equip you to build robust applications that can handle real-world traffic and complexity.

Happy coding!

Let's go **deep into Node.js** — from the absolute basics to advanced concepts, including architecture patterns, performance optimizations, and real-world examples. By the end, you'll have a complete understanding of how Node.js works under the hood and how to use it to build **scalable**, **high-performance** applications.

🚀 1. What is Node.js?

Node.js is an open-source, cross-platform JavaScript runtime environment that allows you to:

✅ Run JavaScript code **outside the browser**.
✅ Build fast, scalable network applications using a **non-blocking, event-driven** model.
✅ Use **single-threaded asynchronous execution** with a built-in thread pool for heavy computation.

✅ Key Features

- Built on **Chrome's V8 Engine** (converts JS into machine code).
- Uses **libuv** for asynchronous I/O handling.
- **Event-driven architecture** using an **event loop**.
- Single-threaded with a **thread pool** for CPU-intensive tasks.
- Lightweight and fast due to non-blocking I/O.

🏆 2. How Node.js Works

Node.js follows an **event-driven, non-blocking I/O** model using the **event loop**.

✅ Core Components

1. **V8 Engine** – Executes JavaScript code.
2. **libuv** – Handles asynchronous I/O (file system, networking, etc.).
3. **Event Loop** – Orchestrates execution of asynchronous code.
4. **Thread Pool** – Executes CPU-bound tasks outside the main thread.

✅ How a Node.js Request Works

1. Client sends an HTTP request.
2. Event loop registers the request and sends it to the worker thread pool (if I/O is involved).
3. Event loop handles other requests in parallel.
4. Once the worker thread returns the result, the event loop sends it back to the client.

👉 **Example:**

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

const server = http.createServer((req, res) => {
  res.end('Hello from Node.js');
});

server.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

➡️ Even if multiple requests hit the server simultaneously, the event loop will continue to handle them without blocking.

🔥 3. Event Loop

The **event loop** is the core of Node.js's asynchronous programming model.

✅ Phases of the Event Loop

1. **Timers Phase** – Executes `setTimeout()` and `setInterval()` callbacks.
2. **Pending I/O Phase** – Executes callbacks from I/O operations (e.g., file read/write).
3. **Idle/Prepare Phase** – Internal functions are processed.
4. **Poll Phase** – Handles new I/O events and executes ready callbacks.
5. **Check Phase** – Executes `setImmediate()` callbacks.
6. **Close Phase** – Executes close events like socket closure.

✅ Order of Execution Example

```
setTimeout(() => console.log('setTimeout'), 0);
setImmediate(() => console.log('setImmediate'));
```

```
process.nextTick(() => console.log('nextTick'));
```

Output:

```
nextTick
setTimeout
setImmediate
```

👉 `process.nextTick()` runs immediately after the current operation but **before** the event loop moves to the next phase.
👉 `setImmediate()` executes in the **Check Phase** (after I/O events).

🌐 4. Callbacks, Promises, and Async/Await

✅ Callbacks

Callbacks are the traditional way of handling asynchronous operations in Node.js:

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

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

✅ Promises

Promises provide better control over asynchronous flow:

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

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

✅ Async/Await

Async/Await provides clean, synchronous-looking asynchronous code:

```javascript
const fs = require('fs').promises;

const readFile = async () => {
  try {
    const data = await fs.readFile('file.txt', 'utf8');
    console.log(data);
  } catch (err) {
    console.error(err);
  }
};

readFile();
```

⚙️ 5. Streams

Streams allow you to handle data in chunks instead of loading it all into memory.

✅ Types of Streams

1. **Readable** – Read data from a source.
2. **Writable** – Write data to a destination.
3. **Duplex** – Act as both readable and writable.
4. **Transform** – Modify data as it passes through.

✅ Example: Readable Stream

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

const stream = fs.createReadStream('file.txt');

stream.on('data', (chunk) => {
  console.log(chunk.toString());
});
```

➡️ Handles large files without memory overflow.

✅ Example: Transform Stream

```
const { Transform } = require('stream');

const transform = new Transform({
  transform(chunk, encoding, callback) {
    this.push(chunk.toString().toUpperCase());
    callback();
  }
});

process.stdin.pipe(transform).pipe(process.stdout);
```

➡️ Transforms input to uppercase in real-time.

🌍 6. File System (fs)

Node.js provides an asynchronous and synchronous file system API:

✅ Example: Read a File

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

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

✅ Example: Write to a File

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

✅ Example: Append to a File

```
fs.appendFile('file.txt', '\nMore data', (err) => {
  if (err) throw err;
  console.log('Data appended');
});
```

🔥 7. Networking

Create a simple TCP server using Node.js `net` module:

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

const server = net.createServer(socket => {
  socket.write('Hello from server!\n');
  socket.on('data', (data) => console.log(data.toString()));
});

server.listen(8000, () => {
  console.log('TCP server running on port 8000');
});
```

⚡ 8. HTTP/HTTPS

✅ HTTP Server Example

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

const server = http.createServer((req, res) => {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello HTTP');
});

server.listen(3000);
```

✅ HTTPS Example

```
const https = require('https');
const fs = require('fs');
const options = {
  key: fs.readFileSync('key.pem'),
  cert: fs.readFileSync('cert.pem')
};

https.createServer(options, (req, res) => {
  res.writeHead(200);
  res.end('Hello HTTPS');
}).listen(443);
```

🚀 9. Child Processes

Spawn a child process using `child_process`:

```
const { spawn } = require('child_process');

const ls = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', (data) => {
  console.log(`Output: ${data}`);
});

ls.stderr.on('data', (data) => {
  console.error(`Error: ${data}`);
});
```

➡️ Use child processes for CPU-bound tasks.

🎯 10. Worker Threads

Worker threads allow Node.js to handle CPU-bound tasks in parallel:

```
const { Worker } = require('worker_threads');

const worker = new Worker('./worker.js');

worker.on('message', (message) => {
  console.log(`Worker message: ${message}`);
});
```

➡️ Separate memory and execution context for parallelism.

🏆 Next Steps

✅ Master the event loop and asynchronous patterns.
✅ Handle file system, streams, and HTTP effectively.
✅ Explore worker threads and child processes for CPU-bound tasks.
✅ Use clustering and load balancing for scalability.
✅ Secure applications with HTTPS, OAuth, and JWT.

Here's a list of **common Node.js modules** that you'll likely use when building a **MERN-based payroll, attendance, and KYC app**. I'll group them based on their specific use cases from the examples we've covered:

✅ 1. HTTP and Networking

👉 1.1. `http` *(Built-in)*

- Used to create an HTTP server for handling requests and responses.
- Ideal for creating RESTful APIs without using Express.

Example:

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

const server = http.createServer((req, res) => {
  res.end('Hello World');
});

server.listen(5000);
```

👉 1.2. `https` *(Built-in)*

- Same as `http`, but used for creating HTTPS servers with SSL/TLS.

Example:

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

const options = {
  key: fs.readFileSync('key.pem'),
  cert: fs.readFileSync('cert.pem'),
};

https.createServer(options, (req, res) => {
  res.end('Secure Server');
}).listen(443);
```

👉 1.3. `url` *(Built-in)*

- Used to parse URL queries.

Example:

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

const parsedUrl = url.parse('http://localhost:5000?employeeId=123', true);
console.log(parsedUrl.query.employeeId); // 123
```

✅ 2. File System (FS)

👉 2.1. `fs` *(Built-in)*

- Used for reading and writing files.
- Useful for generating CSV or JSON payroll reports.

Example:

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

fs.writeFileSync('payroll.csv', 'EmployeeId,Hours
Worked,Salary\n123,40,1000');
```

✅ 3. Path Management

👉 3.1. `path` *(Built-in)*

- Helps with file and directory path handling.

Example:

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

const filePath = path.join(__dirname, 'data', 'payroll.csv');
console.log(filePath);
```

✅ 4. Data Handling and Parsing

👉 4.1. `querystring` *(Built-in)*

- Used to parse query strings.

Example:

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

const parsed = querystring.parse('name=John&age=30');
console.log(parsed.name); // John
```

👉 4.2. `buffer` *(Built-in)*

- Used to work with binary data directly.

Example:

```
const buffer = Buffer.from('Hello World');
console.log(buffer.toString()); // Hello World
```

✅ 5. Database Handling

👉 5.1. `mongodb` *(External)*

- Official MongoDB driver for Node.js.

Install:

```
npm install mongodb
```

Example:

```
const { MongoClient } = require('mongodb');

const client = new MongoClient('mongodb://localhost:27017');
const db = client.db('company');
const attendance = db.collection('attendance');
```

👉 5.2. `mongoose` *(External)*

- MongoDB ORM for schema-based models.

Install:

```
npm install mongoose
```

Example:

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

const attendanceSchema = new mongoose.Schema({
  employeeId: mongoose.Types.ObjectId,
  clockIn: Date,
  clockOut: Date,
});

const Attendance = mongoose.model('Attendance', attendanceSchema);
```

✅ 6. Authentication and Security

👉 6.1. `jsonwebtoken` *(External)*

- Used for authentication using JWT tokens.

Install:

```
npm install jsonwebtoken
```

Example:

```
const jwt = require('jsonwebtoken');

const token = jwt.sign({ id: 123 }, 'SECRET_KEY', { expiresIn: '1h' });
```

👉 6.2. `bcryptjs` *(External)*

- Used for password hashing and comparison.

Install:

```
npm install bcryptjs
```

Example:

```
const bcrypt = require('bcryptjs');

const hashed = bcrypt.hashSync('password', 10);
const isMatch = bcrypt.compareSync('password', hashed);
```

👉 6.3. `crypto` *(Built-in)*

- Used for encryption and creating secure hashes.

Example:

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

const hash = crypto.createHash('sha256').update('password').digest('hex');
console.log(hash);
```

✅ 7. API Handling

👉 7.1. `axios` *(External)*

- Promise-based HTTP client (used for external KYC APIs).

Install:

```
npm install axios
```

Example:

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

const response = await axios.post('https://kyc-provider.com', {
  documentType: 'ID',
  documentNumber: '12345',
});
console.log(response.data);
```

👉 7.2. `node-fetch` *(External)*

- Lightweight HTTP client.

Install:

```
npm install node-fetch
```

Example:

```
const fetch = require('node-fetch');
const response = await fetch('https://api.example.com/data');
const data = await response.json();
```

✅ 8. Real-time Communication

👉 8.1. `ws` *(External)*

- Lightweight WebSocket library for real-time data streaming.

Install:

```
npm install ws
```

Example:

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });
wss.on('connection', (ws) => {
  ws.send('Welcome!');
});
```

👉 8.2. `socket.io` *(External)*

- Full-featured WebSocket library for bi-directional communication.

Install:

```
npm install socket.io
```

Example:

```
const { Server } = require('socket.io');

const io = new Server(server);
io.on('connection', (socket) => {
  socket.emit('message', 'Hello from server');
});
```

✅ 9. File Upload and Parsing

👉 9.1. `multer` *(External)*

- Used for handling file uploads.

Install:

```
npm install multer
```

Example:

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

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

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

✅ 10. Utility Libraries

👉 10.1. `lodash` *(External)*

- Utility library for array and object manipulation.

Install:

```
npm install lodash
```

Example:

```
const _ = require('lodash');
const array = [1, 2, 3, 4];
const chunked = _.chunk(array, 2);
console.log(chunked); // [[1, 2], [3, 4]]
```

👉 10.2. `moment` *(External)*

- Date and time manipulation.

Install:

```
npm install moment
```

Example:

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

const formatted = moment().format('YYYY-MM-DD');
console.log(formatted);
```

✅ 11. Logging

👉 11.1. `winston` *(External)*

- Logging library with support for file and console output.

Install:

```
npm install winston
```

Example:

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

const logger = winston.createLogger({
  transports: [new winston.transports.Console()],
});
```

```
logger.info('Server started');
```

🎯 🔥 Summary

Category	Module
HTTP & Networking	`http`, `https`, `url`
File Handling	`fs`, `path`
Authentication	`jsonwebtoken`, `bcryptjs`, `crypto`
Database	`mongodb`, `mongoose`
API	`axios`, `node-fetch`
Real-time	`ws`, `socket.io`
Utilities	`lodash`, `moment`
Logging	`winston`

Yes, you can absolutely use **ES Modules (ESM)** with **Node.js common libraries** (like `fs`, `path`, `http`, etc.) — but there are some key points and differences to keep in mind.

✅ 1. Enabling ES Modules in Node.js

To enable ES modules in Node.js, you need to:

✅ Option 1: Add `"type": "module"` to `package.json`

Add the following to your `package.json` file:

```
{
  "type": "module"
}
```

- This tells Node.js to treat `.js` files as **ES modules** by default.

✅ Option 2: Use `.mjs` File Extension

Alternatively, you can use the `.mjs` extension for ES module files:

- `app.mjs` → Treated as an ES module.
- `app.js` → Treated as a CommonJS module (if `"type": "module"` is missing).

✅ 2. Importing Node.js Common Libraries in ES Modules

Node.js built-in modules (like `fs`, `path`, `http`, etc.) are fully compatible with ES modules — but you need to use the `import` syntax.

✅ Example: Importing Node.js Core Modules

Module	Import Example
fs (file system)	`import fs from 'fs';`
path	`import path from 'path';`
http	`import http from 'http';`
url	`import { URL } from 'url';`
crypto	`import crypto from 'crypto';`

✅ Example: Using `fs` with ESM

```
// fsExample.mjs
import fs from 'fs/promises';

const readFile = async () => {
  try {
    const data = await fs.readFile('./example.txt', 'utf-8');
    console.log(data);
  } catch (err) {
    console.error('Error reading file:', err);
  }
};

readFile();
```

✅ Example: Using `path` with ESM

```
// pathExample.mjs
import path from 'path';
import { fileURLToPath } from 'url';

// Convert import.meta.url to file path
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

console.log('Filename:', __filename);
console.log('Directory:', __dirname);
```

> `import.meta.url` is a special property in ESM that returns the full file URL.
> You need to convert it to a file path using `fileURLToPath` from the `url` module.

✅ Example: Using `http` with ESM

```
// httpExample.mjs
import http from 'http';

const server = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello from Node.js using ES Modules');
});

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

✅ Example: Using `crypto` with ESM

```
// cryptoExample.mjs
import crypto from 'crypto';

const hash = crypto.createHash('sha256');
hash.update('password');
console.log('Hash:', hash.digest('hex'));
```

✅ Example: Using `url` with ESM

```
// urlExample.mjs
import { URL } from 'url';

const myUrl = new URL('https://example.com/path?name=John');
console.log(myUrl.hostname); // example.com
console.log(myUrl.pathname); // /path
console.log(myUrl.searchParams.get('name')); // John
```

✅ 3. Importing Third-Party Modules

You can use `import` with third-party libraries installed via `npm`:

1. Install the package:

```
npm install axios
```

2. Import and use it:

```
// axiosExample.mjs
import axios from 'axios';

const fetchData = async () => {
  const response = await
axios.get('https://jsonplaceholder.typicode.com/posts/1');
  console.log(response.data);
};
```

```
fetchData();
```

✅ 4. Importing CommonJS Modules into ES Modules

You **cannot directly import CommonJS modules** using `import` in ES modules — but you can work around this using `createRequire`.

✅ Example: Importing CommonJS Modules with `createRequire`

1. Import `createRequire` from the `module` core library:

```
import { createRequire } from 'module';

const require = createRequire(import.meta.url);

// Import CommonJS module
const lodash = require('lodash');

console.log(lodash.chunk([1, 2, 3, 4], 2));
```

✅ Example: Importing `__dirname` and `__filename` in ES Modules

In CommonJS:

- `__dirname` and `__filename` are available automatically.

In ES modules:

- You need to define them manually using `import.meta.url` and `fileURLToPath`:

```
import { fileURLToPath } from 'url';
import path from 'path';

const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);
```

```
console.log(__filename);
console.log(__dirname);
```

✅ 5. Exporting in ES Modules

You can export values from an ES module using two methods:

✅ 1. Named Export

Exports multiple values:

```
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
```

Importing:

```
import { add, subtract } from './math.js';
```

✅ 2. Default Export

Exports a single value:

```
const multiply = (a, b) => a * b;
export default multiply;
```

Importing:

```
import multiply from './math.js';
```

✅ 3. Mixed Export

You can mix named and default exports:

```
export const add = (a, b) => a + b;
export default (a, b) => a * b;
```

Importing:

```
import multiply, { add } from './math.js';
```

✅ 6. Mixing CommonJS and ES Modules

You **cannot mix** `require` and `import` in the same file: ✅ If using `"type": "module"` → Use `import/export` only.
✅ If using CommonJS → Use `require/module.exports` only.

✅ 7. Dynamic Imports with ESM

ES modules support dynamic imports using `import()`:

- `import()` returns a **Promise**.
- You can use `await` with it.

✅ Example:

```
const math = await import('./math.js');
console.log(math.add(2, 3));
```

✅ 8. Differences Between CommonJS and ES Modules

Feature	CommonJS	ES Modules
Syntax	`require()`	`import/export`
Import Timing	Synchronous	Asynchronous
Tree-Shaking	❌ No	✅ Yes

Feature	CommonJS	ES Modules
Dynamic Import	✅ Yes	✅ Yes
Browser Compatibility	❌ No	✅ Yes
Re-exporting	❌ No	✅ Yes
Default Export	`module.exports = value`	`export default value`
Top-Level Await	❌ No	✅ Yes

🚀 🔥 Best Practices

✅ Use **ES Modules** for new projects.
✅ Use **CommonJS** for backward compatibility or legacy code.
✅ Use `createRequire()` if you need to import CommonJS into an ES Module.
✅ Use dynamic imports (`import()`) for better performance.
✅ Always define `__dirname` and `__filename` manually in ES Modules.

Here's a comprehensive list of **methods exposed** by the most commonly used **Node.js core libraries** and their **purposes** — organized by library:

✅ 1. `fs` (File System)

The `fs` module allows you to work with the file system.

Method	Purpose
`fs.readFile()`	Read a file asynchronously
`fs.readFileSync()`	Read a file synchronously
`fs.writeFile()`	Write data to a file asynchronously
`fs.writeFileSync()`	Write data to a file synchronously
`fs.appendFile()`	Append data to a file asynchronously
`fs.appendFileSync()`	Append data to a file synchronously
`fs.unlink()`	Delete a file asynchronously

Method	Purpose
`fs.unlinkSync()`	Delete a file synchronously
`fs.rename()`	Rename a file asynchronously
`fs.renameSync()`	Rename a file synchronously
`fs.mkdir()`	Create a directory asynchronously
`fs.mkdirSync()`	Create a directory synchronously
`fs.rmdir()`	Remove a directory asynchronously
`fs.rmdirSync()`	Remove a directory synchronously
`fs.stat()`	Get information about a file asynchronously
`fs.statSync()`	Get information about a file synchronously
`fs.readdir()`	Read the contents of a directory asynchronously
`fs.readdirSync()`	Read the contents of a directory synchronously
`fs.existsSync()`	Check if a file exists (synchronous)
`fs.watch()`	Watch for file changes
`fs.createReadStream()`	Create a readable file stream
`fs.createWriteStream()`	Create a writable file stream

✅ 2. `path` (File Path Utilities)

The `path` module allows you to handle and manipulate file paths.

Method	Purpose
`path.basename()`	Get the last part of a path
`path.dirname()`	Get the directory name of a path
`path.extname()`	Get the extension of a file
`path.join()`	Join multiple path segments together
`path.resolve()`	Resolve a path into an absolute path
`path.parse()`	Parse a path into an object
`path.format()`	Format an object into a path string
`path.isAbsolute()`	Check if a path is absolute
`path.normalize()`	Normalize a path (removes extra `..` and `.`)

Method	Purpose
`path.relative()`	Get the relative path between two paths

✅ 3. `http` (HTTP Server)

The `http` module allows you to create HTTP servers and handle requests.

Method	Purpose
`http.createServer()`	Create an HTTP server
`server.listen()`	Start the server and listen on a port
`server.close()`	Stop the server
`server.on()`	Attach an event listener to the server
`request.write()`	Send data to the server
`request.end()`	End the request
`request.setHeader()`	Set HTTP headers
`request.getHeader()`	Get an HTTP header
`request.removeHeader()`	Remove an HTTP header
`response.write()`	Send a response to the client
`response.end()`	End the response
`response.setHeader()`	Set response headers
`response.getHeader()`	Get a response header
`response.statusCode`	Set the HTTP status code

✅ 4. `https` (HTTPS Server)

The `https` module works the same way as `http`, but with SSL/TLS.

Method	Purpose
`https.createServer()`	Create an HTTPS server
`https.request()`	Make an HTTPS request

Method	Purpose
`https.get()`	Make a simple HTTPS GET request

✅ 5. `url` (URL Handling)

The `url` module allows you to parse and format URLs.

Method	Purpose
`new URL()`	Create a new URL object
`url.format()`	Convert a URL object into a string
`url.parse()`	Parse a URL string into an object
`url.resolve()`	Resolve a target URL relative to a base URL

✅ 6. `querystring` (Query String)

The `querystring` module allows you to parse and format URL query strings.

Method	Purpose
`querystring.stringify()`	Create a query string from an object
`querystring.parse()`	Parse a query string into an object
`querystring.escape()`	Escape special characters in a query string
`querystring.unescape()`	Unescape special characters in a query string

✅ 7. `crypto` (Cryptography)

The `crypto` module provides cryptographic functions.

Method	Purpose
`crypto.createHash()`	Create a hash object

Method	Purpose
`crypto.createHmac()`	Create an HMAC object
`crypto.randomBytes()`	Generate random bytes
`crypto.pbkdf2()`	Derive a key using PBKDF2
`crypto.sign()`	Sign data with a private key
`crypto.verify()`	Verify signed data
`crypto.createCipher()`	Create a cipher for encryption
`crypto.createDecipher()`	Create a decipher for decryption
`crypto.createDiffieHellman()`	Create a Diffie-Hellman key exchange object

✅ 8. `events` (Event Handling)

The `events` module allows you to work with the EventEmitter pattern.

Method	Purpose
`eventEmitter.on()`	Listen for an event
`eventEmitter.emit()`	Emit an event
`eventEmitter.once()`	Listen for an event only once
`eventEmitter.removeListener()`	Remove a listener
`eventEmitter.removeAllListeners()`	Remove all listeners

✅ 9. `child_process` (Subprocesses)

The `child_process` module allows you to create and manage subprocesses.

Method	Purpose
`child_process.exec()`	Execute a shell command
`child_process.spawn()`	Spawn a new process
`child_process.fork()`	Create a new Node.js process
`child_process.execFile()`	Execute a file directly

✅ 10. `stream` (Streaming)

The `stream` module provides an API for working with streaming data.

Method	Purpose
`stream.pipe()`	Pipe a readable stream into a writable stream
`stream.on()`	Handle stream events
`stream.write()`	Write data to a stream
`stream.end()`	End a writable stream
`stream.read()`	Read data from a readable stream
`stream.push()`	Push data into a readable stream

✅ 11. `tls` (Transport Layer Security)

The `tls` module allows you to create secure TLS/SSL servers and clients.

Method	Purpose
`tls.createServer()`	Create a TLS server
`tls.connect()`	Create a TLS client connection

✅ 12. `zlib` (Compression)

The `zlib` module allows you to compress and decompress data.

Method	Purpose
`zlib.gzip()`	Compress data using Gzip
`zlib.gunzip()`	Decompress Gzip-compressed data
`zlib.deflate()`	Compress data using Deflate
`zlib.inflate()`	Decompress Deflate-compressed data

✅ 13. `dns` (Domain Name System)

The `dns` module allows you to work with DNS.

Method	Purpose
`dns.lookup()`	Look up a hostname
`dns.resolve()`	Resolve a domain name
`dns.reverse()`	Reverse lookup an IP address
`dns.getServers()`	Get the list of DNS servers

✅ 14. `os` (Operating System)

The `os` module provides information about the operating system.

Method	Purpose
`os.platform()`	Get the operating system platform
`os.type()`	Get the operating system type
`os.release()`	Get the operating system release
`os.cpus()`	Get information about CPUs
`os.totalmem()`	Get total system memory
`os.freemem()`	Get free memory

🚀 What is the Event Loop in Node.js?

The **Event Loop** is the mechanism that allows **Node.js** to handle multiple operations (like I/O, network requests, timers, etc.) **asynchronously** — without blocking the main thread.

👉 Node.js is **single-threaded**, but it achieves high performance and scalability by offloading heavy tasks to the **event loop** and underlying system libraries (like `libuv`).

✅ Why is the Event Loop Important?

- **JavaScript is single-threaded** — it can only execute one task at a time.
- However, Node.js can handle **thousands of concurrent operations** without creating multiple threads using the event loop.
- The event loop allows Node.js to handle I/O, network requests, and timers **asynchronously** without blocking execution.

✅ How the Event Loop Works

The event loop works in **phases** — it continuously cycles through these phases and processes any tasks queued up for execution.

🎯 Key Principles:

✅ Tasks are scheduled and executed in a specific order.
✅ Node.js processes one task at a time — **non-blocking** tasks are delegated to the event loop.
✅ Tasks are added to different **queues** based on their type.

✅ The 6 Phases of the Event Loop

The event loop has **six distinct phases**:

Phase	Description	Example
1. Timers	Executes callbacks scheduled by `setTimeout()` and `setInterval()`	`setTimeout()`
2. I/O Callbacks	Executes callbacks from I/O operations (like file reads)	`fs.readFile()`
3. Idle/Prepare	Internal use only – not accessible to developers	–
4. Poll	Retrieves new I/O events; executes I/O-related callbacks	Socket I/O
5. Check	Executes `setImmediate()` callbacks	`setImmediate()`
6. Close	Executes cleanup callbacks (like closing connections)	`socket.on('close')`

✅ How Tasks Enter the Event Loop

➡️ There are 4 main sources of tasks:

1. **Timers** → `setTimeout()`, `setInterval()`
2. **I/O Operations** → File reads/writes, network requests, etc.
3. **Microtasks** → `Promise.then()`, `queueMicrotask()`
4. **Immediate Execution** → `setImmediate()`

✅ Event Loop Priority Order

1. **Microtasks Queue** (Highest Priority)
    - `Promise.then()`
    - `queueMicrotask()`
2. **Timers** (Scheduled Execution)
    - `setTimeout()` and `setInterval()`
3. **I/O Callbacks**
    - Callbacks from asynchronous I/O (like file reads)
4. **Check Phase**
    - `setImmediate()`
5. **Close Phase**
    - Cleanup operations

✅ Step-by-Step Breakdown

🎯 1. Start the Event Loop

- Node.js starts the event loop when the app starts.

🎯 2. Execute Microtasks First

- **Microtasks** (`Promise.then()` and `queueMicrotask()`) are processed **immediately** after the current operation.

- If new microtasks are created during execution, they are added to the queue and executed **before moving to the next phase**.

🎯 3. Process the Timer Phase

- Executes expired timers (`setTimeout()` and `setInterval()`).
- If no timers are ready, the loop moves on.

🎯 4. Process I/O Phase

- Executes I/O callbacks (like file reads and network requests).
- If there's no I/O to process, the loop moves on.

🎯 5. Process the `Check` Phase

- Executes `setImmediate()` callbacks.
- `setImmediate()` is always executed after I/O callbacks.

🎯 6. Process Close Phase

- Executes any cleanup or closing tasks (like closing file descriptors or network sockets).

🎯 7. Repeat

- The event loop continues running until there are **no more tasks** left to execute.

✅ Microtasks vs Macrotasks

Type	Description	Examples
Microtasks	Executed immediately after the current operation finishes	`Promise.then()`, `queueMicrotask()`
Macrotasks	Scheduled in the event loop's next cycle	`setTimeout()`, `setInterval()`, `setImmediate()`

👉 **Microtasks** have a **higher priority** than macrotasks.
👉 Microtasks will execute **before the event loop moves to the next phase**.

✅ How Promises and Timers Work Together

Example Flow:

1. First, **synchronous code** runs.
2. **Microtasks** (`Promise.then()`) execute next.
3. Then, **timers** (`setTimeout`) execute.
4. **I/O callbacks** are processed.
5. `setImmediate()` executes in the **check phase**.

✅ Order of Execution

Example:

```
setTimeout(() => console.log('setTimeout'), 0);
setImmediate(() => console.log('setImmediate'));
Promise.resolve().then(() => console.log('Promise'));
console.log('Sync code');
```

🔎 Output Order Explanation:

1. `Sync code` → Synchronous code executes first.
2. `Promise` → Promise resolves in the **microtask queue** (higher priority).
3. `setTimeout()` → Timer executes in the **timers phase**.
4. `setImmediate()` → Executes in the **check phase** (next phase after I/O).

✅ Actual Output:

```
Sync code
Promise
setTimeout
setImmediate
```

✅ setImmediate vs setTimeout

Feature	`setTimeout()`	`setImmediate()`
Execution Phase	Timer phase	Check phase
Timing	Executes after specified delay	Executes after I/O phase
Priority	Lower than microtasks	Lower than microtasks
Use Case	Delay-based execution	Execute after I/O events

✅ Common Mistakes with the Event Loop

❌ 1. Blocking the Event Loop

- Running CPU-heavy code (e.g., while loops) blocks the event loop.
- Solution → Offload to a worker thread or split the task into smaller pieces.

❌ 2. Mixing Promises and Callbacks Improperly

- Mixing async/await with callbacks can cause race conditions.
- Solution → Stick to one pattern (async/await or callbacks).

❌ 3. Forgetting to Handle Microtasks

- Unhandled `Promise` rejections remain uncaught.
- Solution → Use `.catch()` or `try-catch` around async code.

✅ How to Optimize the Event Loop

✅ Keep functions **small** and **fast**.
✅ Use **async/await** for non-blocking code.
✅ Offload CPU-bound tasks to a **worker thread**.
✅ Avoid **nested setTimeout()** calls.
✅ Handle microtasks properly with `Promise.then()` and `queueMicrotask()`.
✅ Use `setImmediate()` for post-I/O processing.
✅ Don't block the event loop with **synchronous I/O**.

✅ Key Takeaways

1. Node.js is single-threaded — but the event loop allows it to handle concurrency.
2. The event loop has **6 phases**.
3. **Microtasks** have a higher priority than **timers** and **I/O**.
4. `setImmediate()` always runs **before the next event loop cycle**.
5. Blocking the event loop = bad performance.
6. Use **worker threads** for CPU-bound tasks.
