Node.js Design Patterns

Building robust and scalable Applications with Microservices, APIs, Async Programming, and Clean Architecture

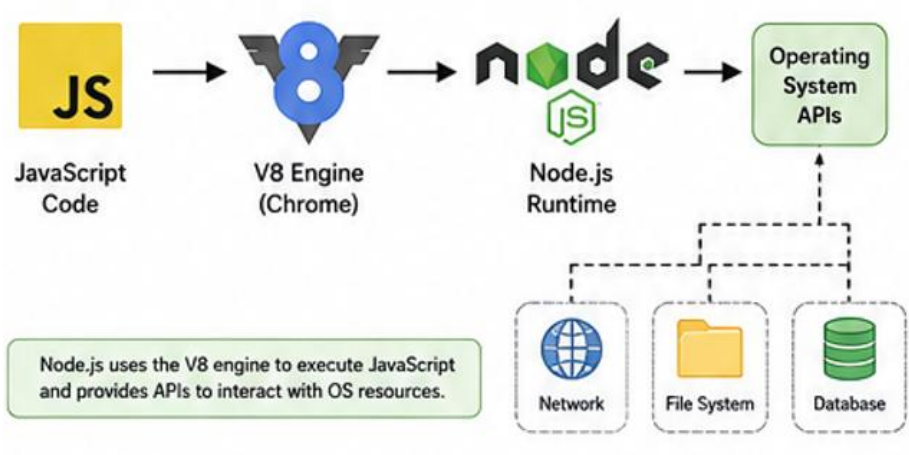
Hernando Abella

Thank you for trusting our publishing house. If you can evaluate our work and give us a comment on amazon, we will appreciate it very much!

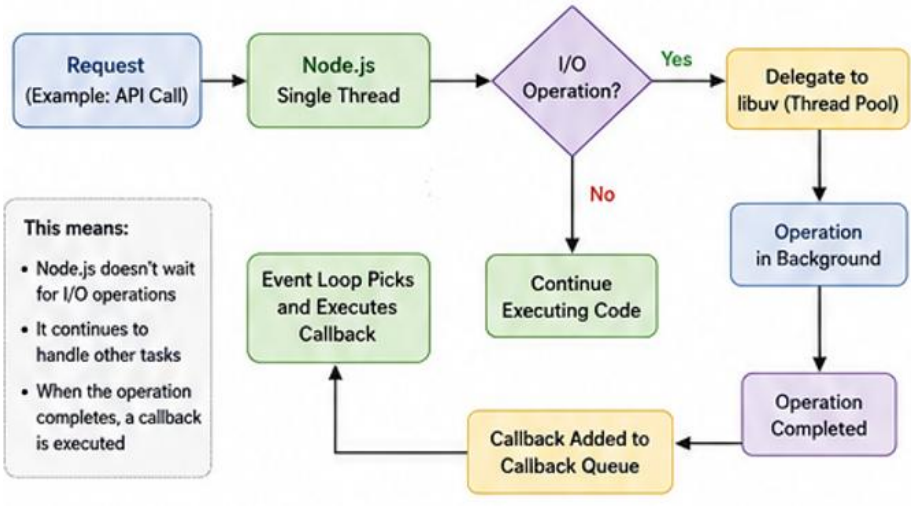
WHAT IS NODE.JS?

Node.js is a **JavaScript runtime built on Chrome's V8 engine** that allows you to run JavaScript outside the browser. Unlike traditional server-side technologies, Node.js is designed for **high-performance, non-blocking, event-driven applications**.

At its core, Node.js is not just a language runtime—it's an **execution model optimized for I/O-heavy workloads**, such as APIs, real-time apps, and streaming systems.



THE BIG IDEA: NON-BLOCKING I/O



Traditional servers (like PHP or Java-based ones) often use a **thread-per-request model**:

- Each request gets its own thread
- Threads consume memory and CPU
- Scaling becomes expensive

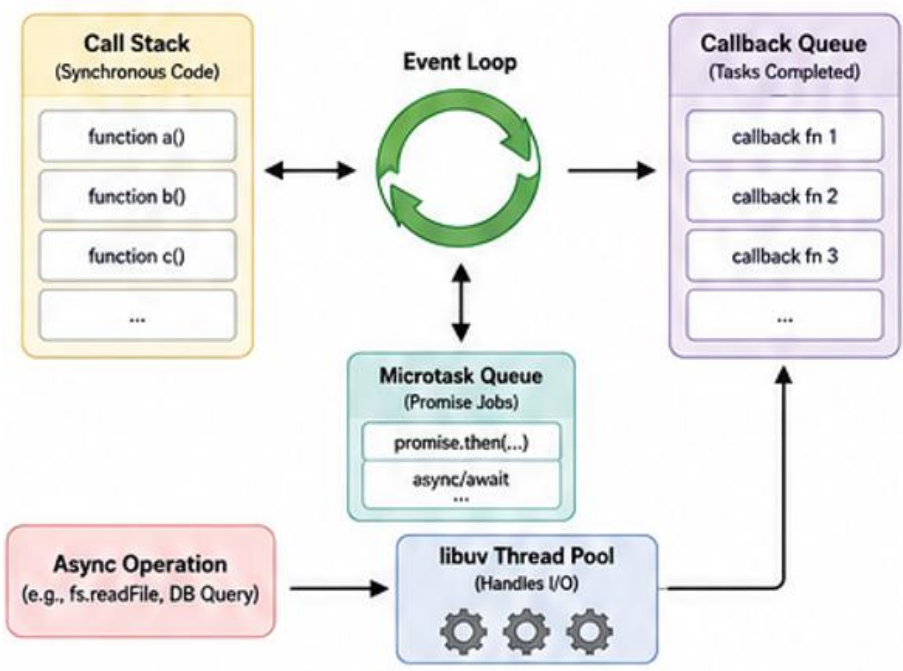
Node.js flips this model:

It uses a **single-threaded, non-blocking architecture**

This means:

- It doesn't wait for operations (like database or file reads)
- It delegates those operations and continues executing other code
- When the operation completes, it comes back via a callback

THE EVENT LOOP (HEART OF NODE.JS)



Event Loop is the core mechanism that makes Node.js work.

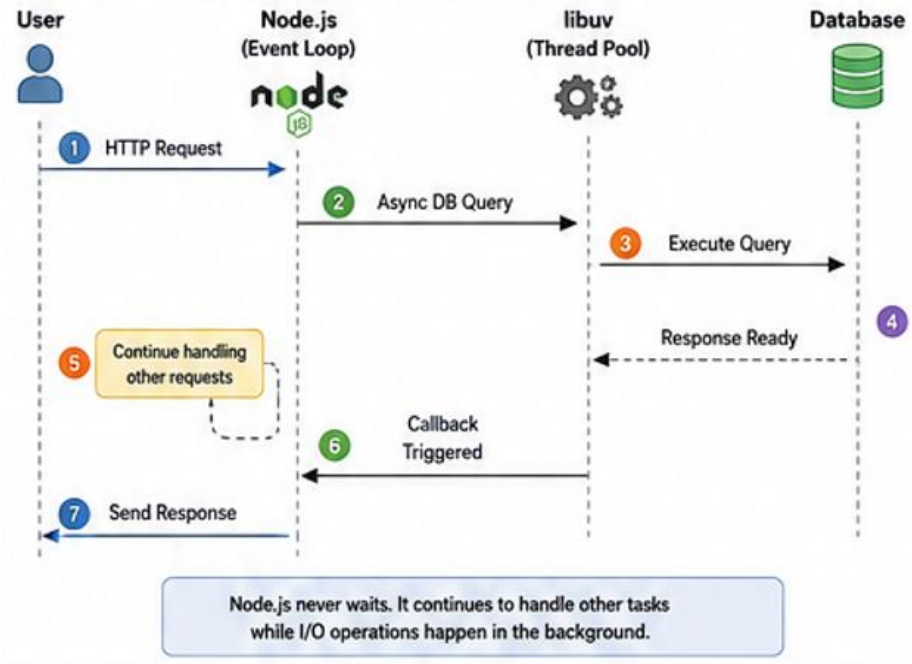
Instead of creating multiple threads, Node.js uses:

- **One main thread**
- **An event loop to manage tasks**
- **A callback queue for completed operations**

How it works:

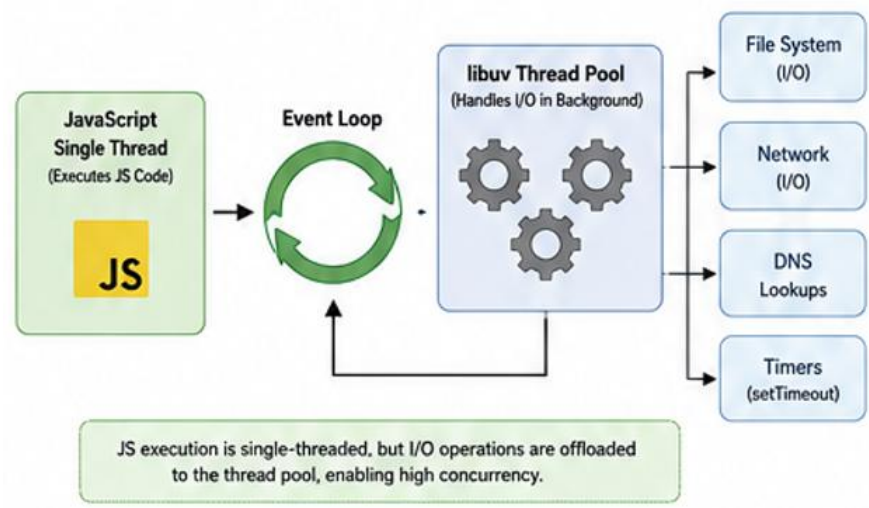
1. Code runs synchronously first
2. Async tasks are delegated (e.g., file system, network)
3. Completed tasks are queued
4. Event loop picks them and executes callbacks

Visualizing the Flow



This architecture allows Node.js to handle **thousands of concurrent connections efficiently** without spawning thousands of threads.

SINGLE THREAD ≠ SINGLE EXECUTION



A common misconception:

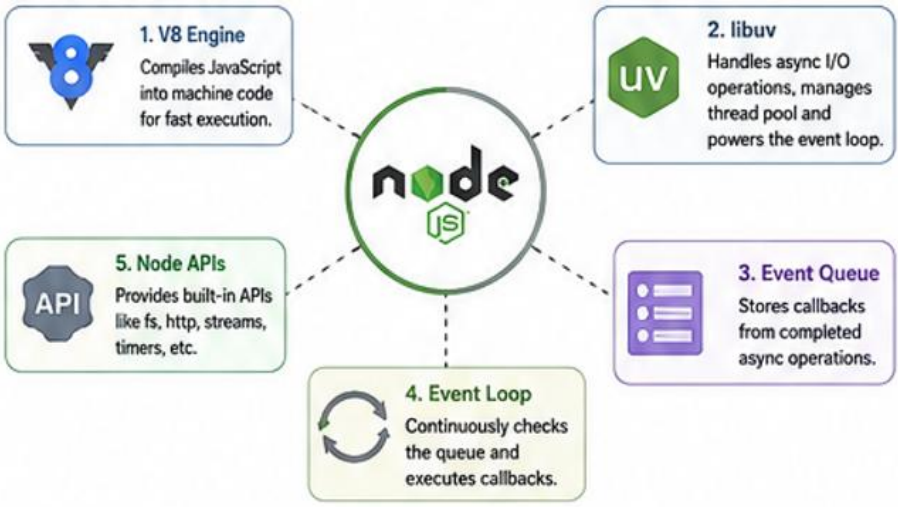
“Node.js is single-threaded, so it can’t handle multiple things at once.”

Reality:

- JavaScript execution is single-threaded
- **I/O operations are handled by a thread pool (libuv)**
- The system itself is highly concurrent

So Node.js achieves **concurrency without complexity**.

CORE COMPONENTS OF NODE.JS ARCHITECTURE



1. V8 ENGINE

- Compiles JavaScript into machine code
- Extremely fast execution

2. LIBUV

- Handles async I/O operations
- Manages thread pool
- Powers the event loop

3. EVENT QUEUE

- Stores callbacks from completed async operations

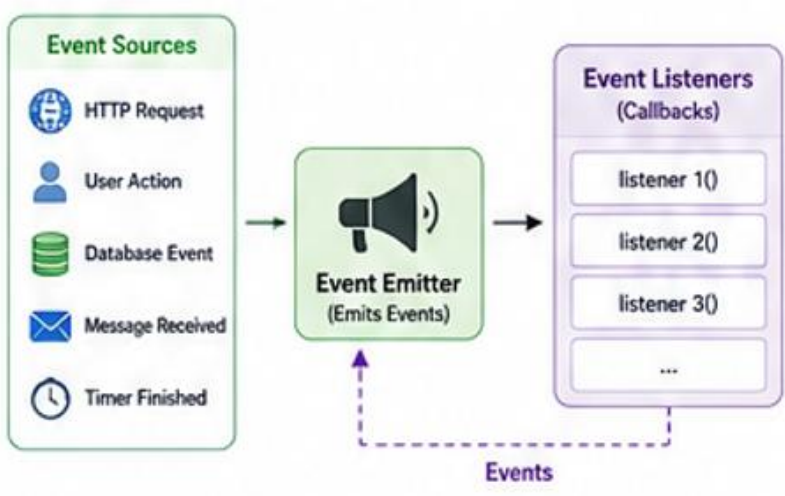
4. EVENT LOOP

- Continuously checks the queue
- Executes callbacks

5. NODE APIS

- File System (`fs`)
- HTTP
- Streams
- Timers

6. EVENT-DRIVEN ARCHITECTURE



Node.js follows an **event-driven model**, meaning:

- Actions trigger events
- Events trigger listeners (callbacks)

Example:

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

emitter.on('userCreated', (user) => {
  console.log('User created:', user);
});
```

```
emitter.emit('userCreated', { name: 'Hernando' });
```

This pattern is fundamental for:

- Real-time apps
- Microservices
- Messaging systems

WHY NODE.JS ARCHITECTURE MATTERS

Understanding this architecture helps you:

- Avoid blocking the event loop
- Design scalable systems
- Choose the right patterns (streams, queues, workers)
- Optimize performance

WHEN TO USE NODE.JS

Best for:

- APIs (REST / GraphQL)
- Real-time apps (chat, gaming)
- Streaming services
- Microservices

Not ideal for:

- CPU-heavy tasks (image processing, ML)
- Complex synchronous computations

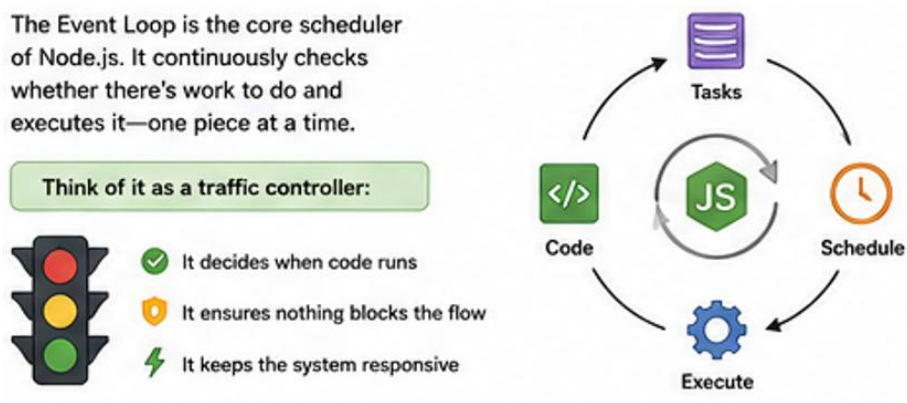
Key Takeaways:

- Node.js uses a **non-blocking, event-driven architecture**
- The **event loop** is the core mechanism

- It achieves **high concurrency with a single thread**
- Powered by **V8 + libuv**
- Best suited for **I/O-heavy applications**

UNDERSTANDING THE EVENT LOOP

WHAT IS THE EVENT LOOP REALLY?



The Event Loop is the **core scheduler** of Node.js. It continuously checks whether there's work to do and executes it—one piece at a time.

Think of it as a **traffic controller**:

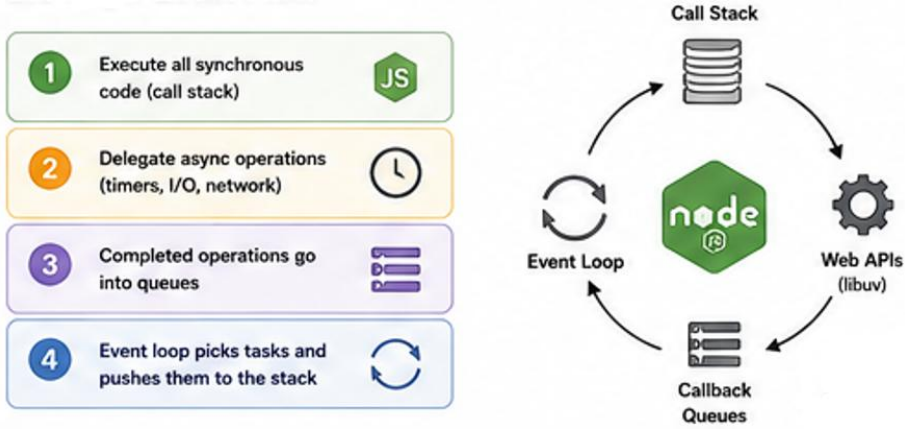
- It decides WHEN code runs
- It ensures NOTHING BLOCKS THE FLOW
- It keeps the system responsive

THE EXECUTION MODEL (MENTAL MODEL)

At runtime, Node.js operates with:

- **Call Stack** → where synchronous code runs
- **Web APIs / libuv** → where async tasks are executed
- **Callback Queues** → where completed tasks wait
- **Event Loop** → orchestrates everything

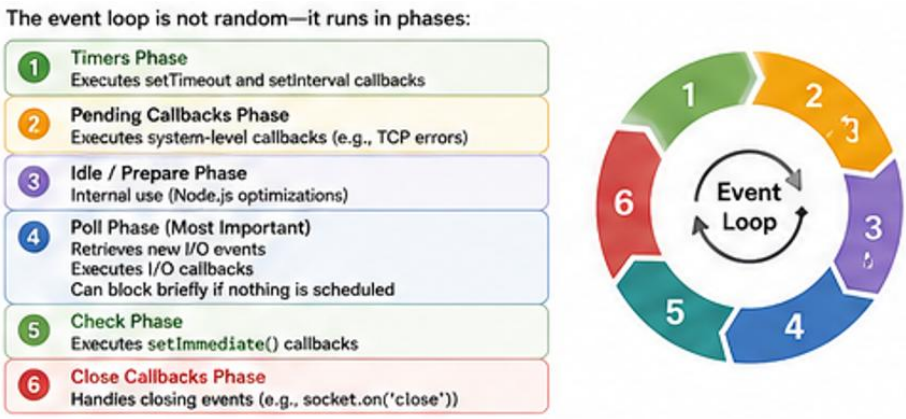
HIGH-LEVEL FLOW



Step-by-step:

1. Execute all synchronous code (call stack)
2. Delegate async operations (timers, I/O, network)
3. Completed operations go into queues
4. Event loop picks tasks and pushes them to the stack

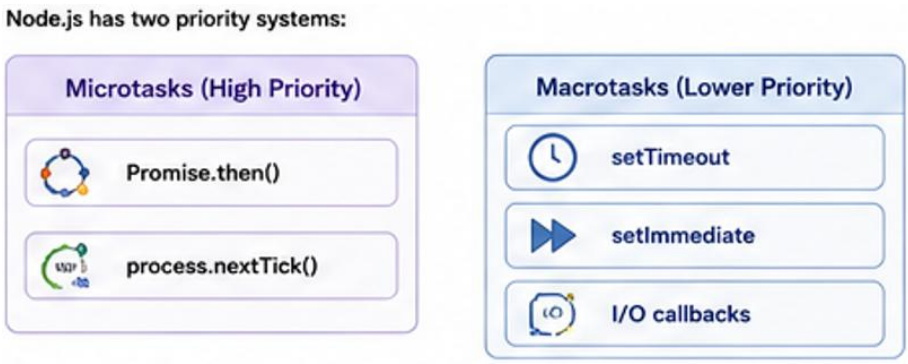
THE PHASES OF THE EVENT LOOP



The event loop is not random—it runs in **phases**:

1. **Timers Phase**
    - Executes `setTimeout` and `setInterval` callbacks
2. **Pending Callbacks Phase**
    - Executes system-level callbacks (e.g., TCP errors)
3. **Idle / Prepare Phase**
    - Internal use (Node.js optimizations)
4. **Poll Phase (Most Important)**
    - Retrieves new I/O events
    - Executes I/O callbacks
    - Can block briefly if nothing is scheduled
5. **Check Phase**
    - Executes `setImmediate()` callbacks
6. **Close Callbacks Phase**
    - Handles closing events (e.g., `socket.on('close')`)

MICROTASKS VS MACROTASKS (CRITICAL CONCEPT)



Node.js has **two priority systems**:

MICROTASKS (HIGH PRIORITY)

- `Promise.then()`
- `process.nextTick()`

MACROTASKS (LOWER PRIORITY)

- `setTimeout`
- `setImmediate`
- I/O callbacks

Rule: Microtasks run BEFORE the next event loop phase

EXAMPLE: EXECUTION ORDER

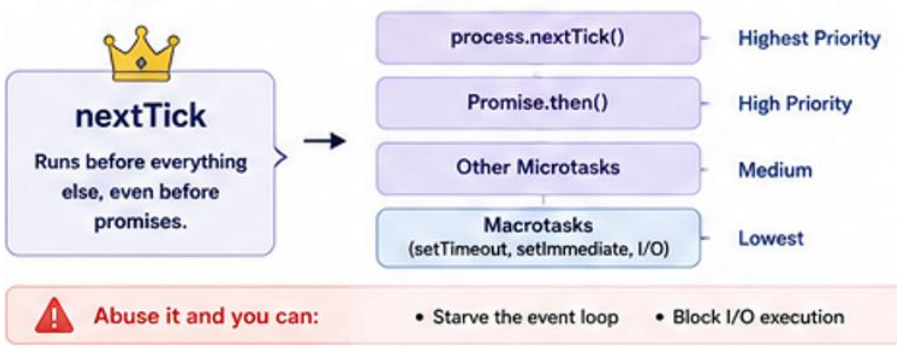
```
console.log('Start');

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

setImmediate(() => console.log('Immediate'));

Promise.resolve().then(() => console.log('Promise'));
```

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

console.log('End');
```

EXPECTED OUTPUT:

```
Start
End
NextTick
Promise
Timeout / Immediate (order may vary)
```

WHY PROCESS.NEXTTICK() IS SPECIAL

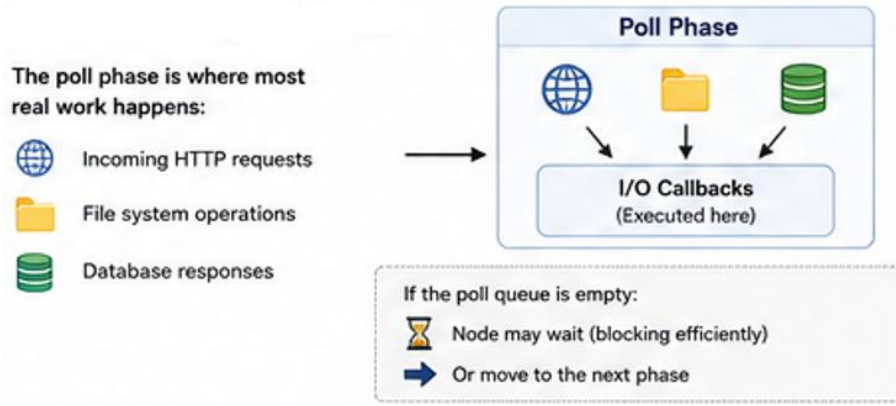


process.nextTick runs **before everything else**, even before promises.

Abuse it and you can:

- Starve the event loop
- Block I/O execution

POLL PHASE DEEP DIVE



The **poll phase** is where most real work happens:

- Incoming HTTP requests
- File system operations
- Database responses

If the poll queue is empty:

- Node may wait (blocking efficiently)
- Or move to the next phase

SETTIMEOUT VS SETIMMEDIATE

This is a classic Node.js interview trap:

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

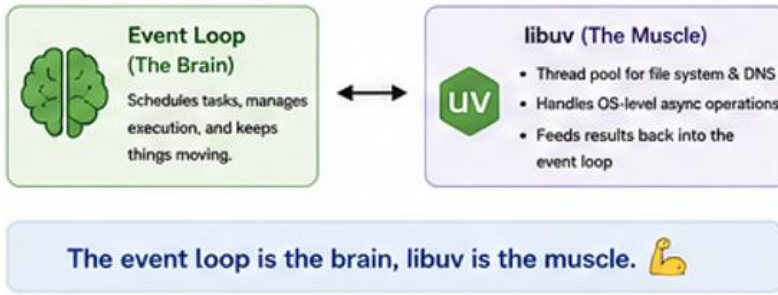
Which runs first?

Answer:

- Depends on context
- Inside I/O → `setImmediate` usually wins

- Outside → timing may vary

EVENT LOOP + LIBUV



libuv is what actually powers async behavior:

- Thread pool for file system & DNS
- Handles OS-level async operations
- Feeds results back into the event loop

The event loop is the **brain**, libuv is the **muscle**.

COMMON PITFALLS

Blocking the event loop:

```
while (true) {}
```

Heavy CPU tasks:

```
for (let i = 0; i < 1e9; i++) {}
```

These freeze everything because:

- The event loop cannot move forward

BEST PRACTICES

- ✓ Keep callbacks fast

- ✓ Use async APIs

Offload CPU work to:

- Worker Threads
- Background jobs

- ✓ Prefer Promises / async-await over callbacks

REAL-WORLD INSIGHT

The event loop is why Node.js can:

- Handle **10,000+ concurrent connections**
- Power apps like:
  - APIs
  - Real-time chats
  - Streaming services

But it's also why:

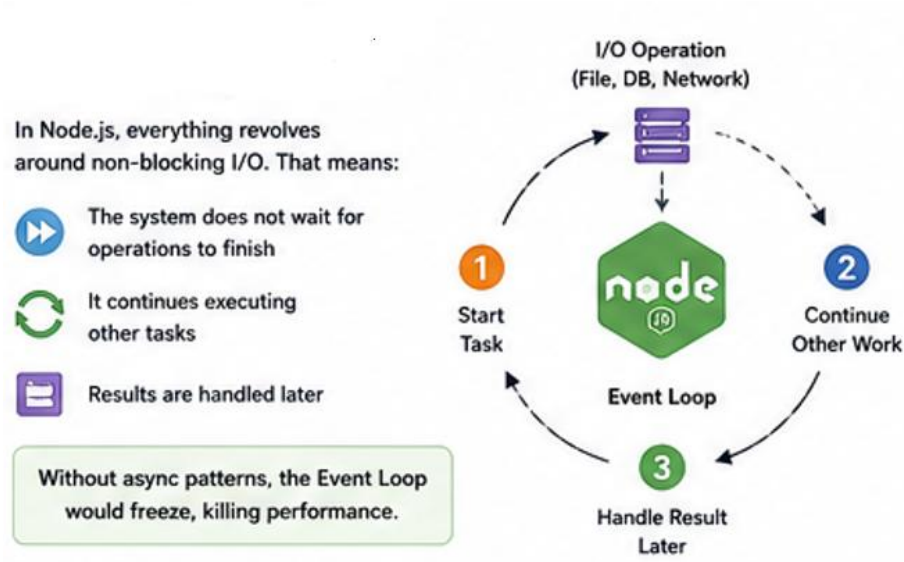
A single bad function can slow down your entire system.

KEY TAKEAWAYS

- The event loop is the **core execution engine**
- It runs in **phases**
- **Microtasks > Macrotasks** in priority
- Powered by **libuv**
- Performance depends on **not blocking the loop**

ASYNCHRONOUS PROGRAMMING PATTERNS

WHY ASYNCHRONOUS PROGRAMMING EXISTS



In Node.js, everything revolves around **non-blocking I/O**. That means:

- The system **does not wait** for operations to finish
- It continues executing other tasks
- Results are handled later

Without async patterns, the Event Loop would **freeze**, killing performance.

THE EVOLUTION OF ASYNC IN JAVASCRIPT

Node.js async patterns evolved in stages:

1. **Callbacks** → simple but messy
2. **Promises** → structured chaining
3. **Async/Await** → synchronous-looking code
4. **Advanced Patterns** → parallelism, control flow, resilience

CALLBACK PATTERN (THE FOUNDATION)

The earliest and simplest async pattern.

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

KEY CHARACTERISTICS:

- Functions passed as arguments
- Executed after task completion
- Uses **error-first convention**

CALLBACK HELL (THE PROBLEM)

```
getUser(id, (err, user) => {
  getOrders(user, (err, orders) => {
    getProducts(orders, (err, products) => {
      console.log(products);
    });
  });
});
```

Problems:

- Hard to read
- Difficult to debug
- Error handling becomes chaotic

PROMISE PATTERN (STRUCTURED ASYNC)

Promises solve callback chaos by representing a **future value**.

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

STATES OF A PROMISE:

- Pending
- Fulfilled
- Rejected

PROMISE CHAINING

```
getUser(id)
  .then(user => getOrders(user))
  .then(orders => getProducts(orders))
  .then(products => console.log(products))
  .catch(err => console.error(err));
```

- ✓ Cleaner than callbacks
- ✓ Centralized error handling

ASYNC/AWAIT (MODERN STANDARD)

Async/await is syntactic sugar over promises.

```
async function main() {
  try {
    const user = await getUser(id);
    const orders = await getOrders(user);
    const products = await getProducts(orders);
    console.log(products);
  } catch (err) {
    console.error(err);
  }
}
```

- ✓ Looks synchronous
- ✓ Easier to read & maintain
- ✓ Best default choice

SEQUENTIAL VS PARALLEL EXECUTION

SEQUENTIAL (SLOWER)

```
const a = await taskA();
const b = await taskB();
```

PARALLEL (FASTER)

```
const [a, b] = await Promise.all([taskA(), taskB()]);
```

Use parallel execution when tasks are independent.

CONCURRENCY PATTERNS

1. PARALLEL EXECUTION

```
await Promise.all([
  fetchUsers(),
  fetchOrders(),
  fetchProducts()
]);
```

2. RACE CONDITION

```
await Promise.race([
  fetchFast(),
  fetchSlow()
]);
```

ALL SETTLED

```
await Promise.allSettled([
  task1(),
  task2()
]);
```

✓ Useful when you don't want failure to stop everything

ERROR HANDLING PATTERNS

TRY/CATCH (ASYNC/AWAIT)

```
try {
  await riskyTask();
} catch (err) {
  console.error(err);
}
```

CENTRALIZED CATCH (PROMISES)

```
doTask()
  .then(step1)
  .then(step2)
  .catch(handleError);
```

RETRY PATTERN

```
async function retry(fn, retries = 3) {
  try {
    return await fn();
  } catch (err) {
    if (retries === 0) throw err;
    return retry(fn, retries - 1);
  }
}
```

CONTROL FLOW PATTERNS

WATERFALL (SEQUENTIAL DEPENDENCY)

- Tasks depend on previous results

PARALLEL (INDEPENDENT TASKS)

- Run simultaneously

QUEUE PATTERN

- Tasks processed one by one

BATCH PROCESSING

- Process items in chunks

STREAMS (ADVANCED ASYNC PATTERN)

Streams handle **large data piece by piece** instead of loading everything into memory.

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

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

- ✓ Memory efficient
- ✓ Ideal for large files & real-time data

EVENT-BASED ASYNC PATTERN

Using events instead of direct calls:

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

emitter.on('done', () => console.log('Task finished'));

setTimeout(() => emitter.emit('done'), 1000);
```

- ✓ Decouples components
- ✓ Used in real-time systems

COMMON MISTAKES

Blocking inside async:

```
await slowTask();
while(true) {} // destroys everything
```

Not handling errors:

```
await riskyTask(); // crash if fails
```

Over-awaiting:

```
await a();
await b(); // should be parallel
```

BEST PRACTICES

- ✓ Prefer **async/await**
- ✓ Use **Promise.all** for parallelism
- ✓ Always handle errors
- ✓ Avoid deep nesting
- ✓ Use streams for large data
- ✓ Keep functions small and composable

REAL-WORLD INSIGHT

Async patterns are what allow Node.js to:

- Handle thousands of requests
- Run APIs efficiently
- Process real-time data
- Scale without heavy hardware

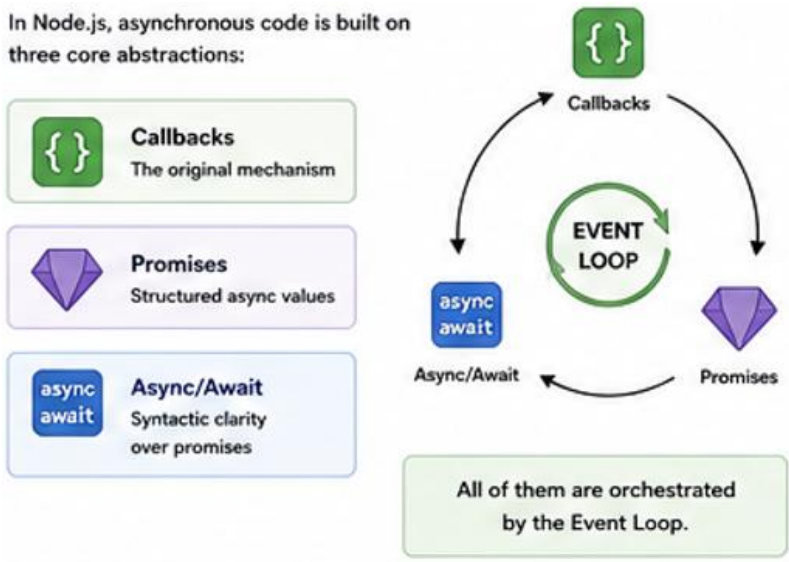
But misuse leads to:

- Slow apps
- Memory leaks
- Race conditions

KEY TAKEAWAYS

- Async programming is **core to Node.js**
- Evolution: **Callbacks → Promises → Async/Await**
- Use **parallel execution** when possible
- Master **error handling patterns**
- Advanced patterns unlock **real scalability**

CALLBACKS, PROMISES, AND ASYNC/AWAIT



THE BIG PICTURE

In Node.js, asynchronous code is built on three core abstractions:

- **Callbacks** → the original mechanism
- **Promises** → structured async values

- **Async/Await** → syntactic clarity over promises

All of them are orchestrated by the Event Loop.

CALLBACKS — THE PRIMITIVE LAYER

A **callback** is simply a function passed to another function to be executed later.

```
setTimeout(() => {
  console.log('Executed later');
}, 1000);
```

ERROR-FIRST CALLBACK CONVENTION

Node.js standardized callbacks like this:

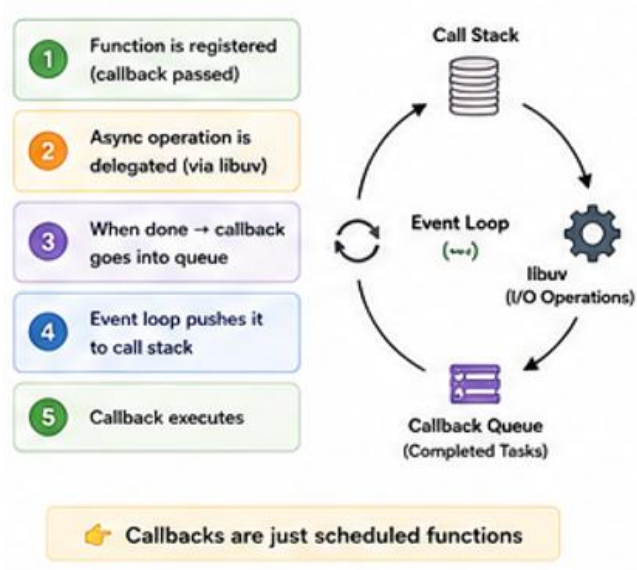
```
function callback(err, result) {}
```

Example:

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

- ✓ Predictable structure
- × Manual error handling everywhere
- × Hard to compose

WHAT ACTUALLY HAPPENS (INTERNALLY)



1. Function is registered (callback passed)
2. Async operation is delegated (via **libuv**)
3. When done → callback goes into queue
4. Event loop pushes it to call stack

Callbacks are **just scheduled functions**

PROMISES — THE ABSTRACTION LAYER

A **Promise** represents a **value that will exist in the future**.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve('Done'), 1000);
});
```

PROMISE STATES

- **Pending** → initial
- **Fulfilled** → success

- **Rejected** → failure

THENABLES & CHAINING

```
getUser()
  .then(user => getOrders(user))
  .then(orders => getProducts(orders))
  .then(console.log)
  .catch(console.error);
```

WHY THIS WORKS

Each `.then()`:

- Returns a **new promise**
- Passes its result forward

This creates a **chain of async transformations**

MICROTASKS (CRITICAL INTERNAL DETAIL)

Promises don't go to the normal queue.

They go to the **microtask queue**, which has higher priority.

EXECUTION ORDER EXAMPLE:

```
console.log('A');

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

Promise.resolve().then(() => console.log('C'));

console.log('D');
```

OUTPUT:

```
A
D
```

```
C
B
```

Why?

- Sync code runs first
- **Microtasks (Promises)** run next
- Then macrotasks (`setTimeout`)

ASYNC/AWAIT — THE SYNTACTIC LAYER

`async/await` is built directly on top of promises.

```
async function example() {
  const data = await fetchData();
  console.log(data);
}
```

Internally, this becomes:

```
function example() {
  return fetchData().then(data => {
    console.log(data);
  });
}
```

HOW AWAIT WORKS INTERNALLY

When Node.js sees:

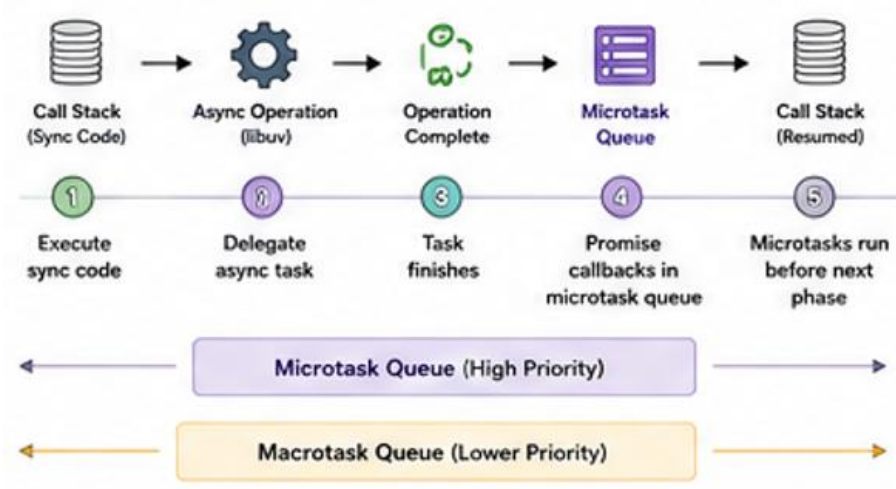
```
await something();
```

It does:

1. Pause the function
2. Let other code run
3. Resume when promise resolves

But it does **NOT block the event loop**

EXECUTION FLOW VISUALIZATION



SEQUENTIAL VS PARALLEL TRAP

SEQUENTIAL (SLOW)

```
const a = await getA();
const b = await getB();
```

PARALLEL (CORRECT)

```
const [a, b] = await Promise.all([getA(), getB()]);
```

`await` can accidentally **serialize your code**

ERROR HANDLING DIFFERENCES

CALLBACKS

```
callback(err, result);
```

PROMISES

```
promise.catch(err => {});
```

ASYNC/AWAIT

```
try {
  await task();
} catch (err) {
  console.error(err);
}
```

PROMISE INTERNALS (WHAT MOST DEVS DON'T KNOW)

- Promises are **eager** (they execute immediately)
- They are **immutable once settled**
- `.then()` always returns a new promise
- Errors automatically propagate down the chain

HIDDEN PITFALLS

UNHANDLED PROMISE REJECTIONS

```
await riskyTask(); // crashes if not caught
```

MIXING CALLBACK + PROMISE

```
fs.readFile('file', (err, data) => {
  return Promise.resolve(data); // useless
});
```

OVERUSING AWAIT IN LOOPS

```
for (const item of items) {
  await process(item); // slow
}
```

Better:

```
await Promise.all(items.map(process));
```

WHEN TO USE WHAT

USE CALLBACKS:

- Legacy APIs
- Low-level Node.js internals

USE PROMISES:

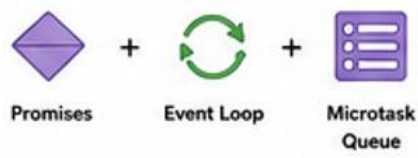
- Libraries
- Functional chaining

USE ASYNC/AWAIT:

- Default choice
- Clean, readable code

REAL-WORLD INSIGHT

All modern Node.js code is essentially:



- Predict execution order
- Avoid race conditions
- Optimize performance

KEY TAKEAWAYS

- Callbacks are the **foundation**
- Promises add **structure and composability**
- Async/await adds **clarity and readability**
- **Microtasks (Promises) run before macrotasks**
- Misusing `await` can hurt performance

MODULES AND DEPENDENCY MANAGEMENT

WHY MODULES MATTER

As applications grow, managing code in a single file becomes impossible.
Modules solve this by allowing you to:

- Split code into **reusable units**
- Encapsulate logic
- Manage dependencies cleanly

In Node.js, everything is modular by design.

WHAT IS A MODULE?

A **module** is simply a file that exports functionality:

```
// math.js
function add(a, b) {
  return a + b;
}

module.exports = { add };
// app.js
const { add } = require('./math');

console.log(add(2, 3));
```

Each file has its **own scope**, avoiding global pollution.

MODULE SYSTEMS IN NODE.JS

There are **two main module systems**:

1. COMMONJS (CJS)

- Default in Node.js (historically)

- Uses `require()` and `module.exports`

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

- ✓ Simple
- ✓ Synchronous loading

2. ES MODULES (ESM)

- Modern JavaScript standard
- Uses `import` / `export`

```
import fs from 'fs';
export function add(a, b) {}
```

- ✓ Static structure
- ✓ Better for optimization
- ✓ Works in browser + Node

CJS VS ESM (MENTAL MODEL)

Feature	CommonJS	ES Modules
Syntax	require()	import/export
Loading	Dynamic	Static
Execution	Runtime	Pre-analyzed
Async Support	Limited	Native

MODULE RESOLUTION (HOW NODE FINDS FILES)

When you write:

```
require('express');
```

Node.js resolves it in this order:

1. Core modules (`fs`, `path`)
2. `node_modules/`

3. Local files (`./`, `../`)

RESOLUTION FLOW

Understanding this helps avoid:

- Dependency conflicts
- Import errors
- Duplicate packages

THE MODULE WRAPPER

Node.js wraps every file like this:

```
(function (exports, require, module, __filename, __dirname) {
  // your code here
});
```

This is why you can use:

- `__dirname`
- `module.exports`
- `require`

DEPENDENCY MANAGEMENT WITH NPM

npm is the default package manager.

PACKAGE.JSON BASICS

```
{
  "name": "my-app",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.18.0"
  }
}
```

TYPES OF DEPENDENCIES

- **dependencies** → needed in production
- **devDependencies** → development only
- **peerDependencies** → required by libraries

SEMANTIC VERSIONING (SEMVER)

Version format:

`MAJOR.MINOR.PATCH`

Example:

```
^1.2.3
```

- ^ → allows minor + patch updates
- ~ → allows patch only

Prevents breaking changes.

NODE MODULES STRUCTURE

When installing packages:

```
node_modules/
  express/
  lodash/
package.json
package-lock.json
```

`package-lock.json` ensures:

- Exact versions
- Reproducible installs

DEPENDENCY GRAPH (REAL INSIGHT)

A single package can bring:

- Dozens or hundreds of dependencies

This is why:

- Security matters
- Updates matter
- Size matters

BEST PRACTICES

- ✓ Prefer **ES Modules** for new projects
- ✓ Keep dependencies minimal
- ✓ Use `package-lock.json`
- ✓ Avoid global installs
- ✓ Use environment-based configs

ADVANCED PATTERNS

1. BARREL EXPORTS

```
export * from './user';
export * from './auth';
```

2. DEPENDENCY INJECTION

```
function createService(db) {
  return {
    getUsers: () => db.query()
  };
}
```

✓ Decouples modules

✓ Easier testing

3. PLUGIN ARCHITECTURE

- Load modules dynamically
- Extend functionality

COMMON PITFALLS

❌ Circular dependencies
❌ Huge dependency trees
❌ Mixing CJS and ESM incorrectly
❌ Forgetting to lock versions

REAL-WORLD INSIGHT

In real systems:

- Modules = **architecture building blocks**
- Dependencies = **risk + leverage**

The best developers:

- Write **small, composable modules**
- Control dependencies carefully
- Think in **systems, not files**

KEY TAKEAWAYS

- Modules enable **scalable architecture**
- Node.js supports **CJS + ESM**
- npm manages dependencies
- Module resolution is critical to understand
- Dependency management impacts performance, security, and maintainability

WRITING CLEAN AND MAINTAINABLE CODE

WHY CLEAN CODE MATTERS

In Node.js, messy code doesn't just look bad—it **breaks scalability**.

Clean code gives you:

- Faster development

- Easier debugging
- Better collaboration
- Long-term maintainability

Rule:

Code is read far more often than it is written.

THE CORE PHILOSOPHY

Clean code is about:

- **Clarity over cleverness**
- **Simplicity over complexity**
- **Consistency over personal style**

MEANINGFUL NAMING

Bad naming creates confusion instantly.

Poor:

```
const d = new Date();
const x = users.filter(u => u.a);
```

Clean:

```
const currentDate = new Date();
const activeUsers = users.filter(user => user.isActive);
```

Names should:

- Reveal intent
- Be searchable
- Avoid abbreviations

SMALL, FOCUSED FUNCTIONS

Functions should do **one thing only**.

Bad:

```
function processUser(user) {
  validate(user);
  saveToDB(user);
  sendEmail(user);
}
```

Better:

```
function validateUser(user) {}
function saveUser(user) {}
function notifyUser(user) {}
```

Benefits:

- Easier testing
- Reusable logic
- Clear responsibility

AVOID DEEP NESTING

Deep nesting = hard to read.

```
if (user) {
  if (user.isActive) {
    if (user.isAdmin) {
      // logic
    }
  }
}
```

Use early returns:

```
if (!user) return;
if (!user.isActive) return;
if (!user.isAdmin) return;

// logic
```

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

Each module/function should have **one reason to change**.

Instead of:

```
class UserService {
  saveUser() {}
  sendEmail() {}
  generateReport() {}
}
```

Split responsibilities:

```
class UserService {}
class EmailService {}
class ReportService {}
```

DRY (DON'T REPEAT YOURSELF)

Duplicate logic = future bugs.

Avoid:

```
if (user.role === 'admin') {}
if (user.role === 'admin') {}
```

Use:

```
function isAdmin(user) {
  return user.role === 'admin';
}
```

KISS (KEEP IT SIMPLE, STUPID)

Avoid overengineering.

Avoid:

```
class UserFactoryManagerSingleton {}
```

Use:

```
function createUser(data) {
  return { ...data };
}
```

Simpler = easier to maintain.

CONSISTENT CODE STYLE

Use tools like:

- ESLint
- Prettier

Consistency reduces:

- Cognitive load
- Code review friction

ERROR HANDLING DISCIPLINE

Bad error handling kills systems.

Avoid:

```
try {
  doSomething();
} catch (e) {}
```

Use:

```
try {
  doSomething();
} catch (error) {
  console.error(error);
  throw error;
}
```

Always:

- Log errors
- Handle gracefully
- Fail explicitly

AVOID MAGIC NUMBERS & STRINGS

Avoid:

```
if (status === 3) {}
```

Use:

```
const STATUS_ACTIVE = 3;
if (status === STATUS_ACTIVE) {}
```

CODE STRUCTURE (FOLDER ORGANIZATION)

A scalable Node.js structure:

```
src/
  modules/
    user/
      user.controller.js
      user.service.js
      user.repository.js
  shared/
  config/
```

Organize by **feature**, not by type.

COMMENTS: WHEN TO USE THEM

Bad comments:

```
// increment i
i++;
```

Good comments:

```
// Retry logic to handle transient network failures
```

- ✓ Code should explain HOW
- ✓ Comments explain WHY

IMMUTABILITY & PREDICTABILITY

Avoid mutating data:

```
user.name = 'John';
```

use instead:

```
const updatedUser = { ...user, name: 'John' };
```

- Predictable code = fewer bugs.

SEPARATION OF CONCERNS

Split logic into layers:

- Controller → handles request
- Service → business logic
- Repository → database

This makes systems:

- Testable
- Scalable
- Maintainable

CLEAN CODE VS BAD CODE

REFACTORING MINDSET

Clean code is not written once—it's **refactored continuously**.

Always:

- Improve naming
- Reduce duplication
- Simplify logic

REAL-WORLD INSIGHT

Most software fails not because of:

- Bugs
- Performance

- But because of: **Unmaintainable codebases**

Clean code is a **business advantage**, not just a technical one.

KEY TAKEAWAYS

- Write code for **humans, not machines**

- Keep functions **small and focused**
- Use **clear naming**
- Avoid duplication
- Structure code by **responsibility**
- Refactor constantly

CORE DESIGN PRINCIPLES

SOLID PRINCIPLES IN JAVASCRIPT

WHAT IS SOLID?

SOLID is a set of 5 design principles that help you write:

- Scalable systems
- Maintainable code
- Flexible architectures

Even in Node.js (which is not strictly OOP), these principles are **extremely powerful**.

THE FIVE PRINCIPLES

Principle	Meaning
S	Single Responsibility Principle
O	Open/Closed Principle
L	Liskov Substitution Principle
I	Interface Segregation Principle
D	Dependency Inversion Principle

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

“A module should have one reason to change.”

BAD DESIGN

```
class UserService {
  createUser(user) {
    // save to DB
  }
```

```
  sendEmail(user) {
    // send email
  }
}
```

GOOD DESIGN

```
class UserService {
  createUser(user) {}
}

class EmailService {
  sendEmail(user) {}
}
```

Benefits:

- Easier testing
- Easier changes
- Clear responsibilities

OPEN/CLOSED PRINCIPLE (OCP)

Open for extension, closed for modification.”

BAD DESIGN

```
function calculateDiscount(user) {
  if (user.type === 'premium') return 20;
  if (user.type === 'vip') return 30;
}
```

GOOD DESIGN

```
class Discount {
  calculate() {}
}
```

```
class PremiumDiscount extends Discount {
  calculate() { return 20; }
}

class VipDiscount extends Discount {
  calculate() { return 30; }
}
```

Now you can add new types **without modifying existing code**.

LISKOV SUBSTITUTION PRINCIPLE (LSP)

"Subtypes must be replaceable for their base types."

BAD DESIGN

```
class Bird {
  fly() {}
}

class Penguin extends Bird {
  fly() {
    throw new Error("Can't fly");
  }
}
```

GOOD DESIGN

```
class Bird {}

class FlyingBird extends Bird {
  fly() {}
}

class Penguin extends Bird {}
```

Avoid breaking expectations of base classes.

INTERFACE SEGREGATION PRINCIPLE (ISP)

“Don’t force modules to depend on things they don’t use.”

BAD DESIGN

```
class Worker {
  work() {}
  eat() {}
}
```

GOOD DESIGN

```
class Workable {
  work() {}
}

class Eatable {
  eat() {}
}
```

Split large interfaces into smaller ones.

DEPENDENCY INVERSION PRINCIPLE (DIP)

“Depend on abstractions, not concrete implementations.”

BAD DESIGN

```
class UserService {
  constructor() {
    this.db = new MySQLDatabase();
  }
}
```

GOOD DESIGN

```
class UserService {
  constructor(db) {
    this.db = db;
```

```
  }
}
```

Now you can inject:

- MySQL
- MongoDB
- Mock DB for testing

SOLID IN FUNCTIONAL JAVASCRIPT

Even without classes, SOLID applies:

```
function createUserService({ db, emailService }) {
  return {
    createUser: (user) => {
      db.save(user);
      emailService.send(user);
    }
  };
}
```

This is **Dependency Injection + SRP** in functional style.

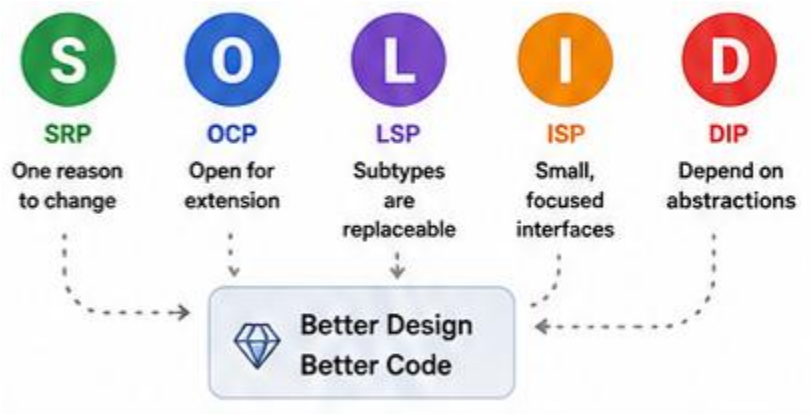
REAL-WORLD NODE.JS EXAMPLE (LAYERED ARCHITECTURE)

```
Controller → Service → Repository → Database
```

- Controller → handles HTTP
- Service → business logic
- Repository → data access

Each layer follows **SRP + DIP**

SOLID VISUALIZATION



COMMON MISTAKES

- × Overengineering everything
- × Using classes when functions are enough
- × Ignoring simplicity (KISS)
- × Misusing inheritance

WHEN NOT TO USE SOLID

Don’t apply SOLID blindly:

- Small scripts
- Simple apps
- MVPs

Use it when:

- Complexity grows
- Team size increases
- Codebase scales

REAL-WORLD INSIGHT

Senior engineers don’t just write code…

hey design systems that:

- Survive change
- Scale easily
- Are easy to reason about

SOLID is the **foundation of that mindset**.

KEY TAKEAWAYS

- **SRP** → one responsibility
- **OCP** → extend, don’t modify
- **LSP** → don’t break expectations
- **ISP** → keep interfaces small
- **DIP** → depend on abstractions

DRY, KISS, AND YAGNI IN NODE.JS

WHY THESE PRINCIPLES MATTER

In Node.js, most problems don’t come from lack of features…

They come from:

- Overengineering
- Duplication
- Unnecessary complexity

DRY, KISS, and YAGNI are your defense system.

THE THREE PRINCIPLES AT A GLANCE

Principle	Meaning
DRY	Don’t Repeat Yourself
KISS	Keep It Simple, Stupid

YAGNI	You Aren’t Gonna Need It

DRY — DON’T REPEAT YOURSELF

“Every piece of knowledge should have a single, unambiguous representation.”

BAD (DUPLICATION)

```
function calculatePrice(price) {
  return price * 1.19;
}

function calculateTotal(price) {
  return price * 1.19;
}
```

GOOD (REUSABLE LOGIC)

```
function applyTax(price) {
  return price * 1.19;
}
```

TYPES OF DUPLICATION

1. **Code duplication**
2. **Logic duplication**
3. **Knowledge duplication** (most dangerous)

WHEN NOT TO DRY

Premature DRY = bad abstraction

Avoid:

```
function process(entity) {
  // too generic, unclear
}
```

Sometimes duplication is okay **until patterns emerge**.

KISS — KEEP IT SIMPLE, STUPID

“Simplicity is the ultimate sophistication.”

OVERENGINEERED

```
class UserManagerFactoryBuilder {
  createUser() {}
}
```

SIMPLE

```
function createUser(data) {
  return { ...data };
}
```

KISS IN PRACTICE

- Prefer **functions over classes** when possible
- Avoid unnecessary abstractions
- Write code that a junior dev can understand

COMPLEXITY WARNING SIGNS

- Too many layers
- Too many abstractions
- Hard-to-follow logic

YAGNI — YOU AREN’T GONNA NEED IT

“Don’t build for the future—build for the present.”

OVERBUILDING

```
class PaymentProcessor {
  processStripe() {}
  processPayPal() {}
  processCrypto() {}
}
```

(You only needed Stripe…)

FOCUSED

```
function processStripePayment() {}
```

Add complexity **only when needed**.

THE BALANCE BETWEEN THEM

These principles can conflict:

- DRY → abstraction
- KISS → simplicity
- YAGNI → minimalism

The goal is **balance**, not perfection.

REAL NODE.JS EXAMPLES

CASE 1: API SERVICE

Overengineered:

```
class UserServiceFactory {
  createService() {}
}
```

Clean:

```
function getUser(id) {
  return db.users.findById(id);
```

```
}
```

CASE 2: VALIDATION LOGIC

Repeated:

```
if (!email.includes('@')) {}
if (!email.includes('@')) {}
```

DRY:

```
function isValidEmail(email) {
  return email.includes('@');
}
```

CASE 3: PREMATURE ABSTRACTION

Avoid:

```
function handleEntity(entityType, data) {}
```

Use:

```
function createUser(data) {}
function createOrder(data) {}
```

PRACTICAL RULES (WHAT PROS ACTUALLY DO)

Follow this order:

1. **Start simple (KISS)**
2. **Avoid future assumptions (YAGNI)**
3. **Refactor duplication later (DRY)**

ANTI-PATTERNS

- ✖ Over-DRY (too abstract)
- ✖ Enterprise syndrome (too many layers)
- ✖ Premature optimization

- ✗ Generic everything

REAL-WORLD INSIGHT

Junior devs: Try to make code “perfect”

Senior devs: Keep code **simple and adaptable**

The real skill is knowing:

- When to abstract
- When to duplicate
- When to stop

MENTAL MODEL

Before writing code, ask:

- **DRY** → Am I repeating myself?
- **KISS** → Is this the simplest solution?
- **YAGNI** → Do I actually need this now?

KEY TAKEAWAYS

- **DRY** reduces duplication
- **KISS** reduces complexity
- **YAGNI** prevents overbuilding
- Balance matters more than strict rules
- Simplicity wins in real systems

FUNCTIONAL VS OBJECT-ORIENTED PATTERNS

WHY THIS MATTERS

In Node.js, you’re not forced into one paradigm.

You can write:

- Functional code
- Object-Oriented code
- Or a hybrid

The real skill is **knowing when to use each**.

THE TWO PARADIGMS

Paradigm	Focus
Functional Programming (FP)	Functions & data transformations
Object-Oriented Programming (OOP)	Objects & behavior encapsulation

FUNCTIONAL PROGRAMMING (FP)

"Programs are built using pure functions."

CORE PRINCIPLES

- **Pure functions** (no side effects)
- **Immutability**
- **Function composition**
- **Declarative style**

EXAMPLE (FUNCTIONAL)

```
const users = [
  { name: 'Ana', active: true },
  { name: 'Juan', active: false }
];

const activeUsers = users
  .filter(user => user.active)
  .map(user => user.name);
```

PURE FUNCTION

```
function add(a, b) {
  return a + b;
```

}

Same input → same output

IMPURE FUNCTION

```
let total = 0;

function addToTotal(x) {
  total += x;
}
```

ADVANTAGES OF FP

- ✓ Predictable
- ✓ Easy to test
- ✓ Less bugs
- ✓ Great for async flows

DISADVANTAGES

- ✗ Can become abstract
- ✗ Harder for beginners
- ✗ Not ideal for modeling complex systems

OBJECT-ORIENTED PROGRAMMING (OOP)

"Programs are built using objects that encapsulate behavior."

CORE PRINCIPLES

- Encapsulation
- Abstraction
- Inheritance

- Polymorphism

EXAMPLE (OOP)

```
class User {
  constructor(name) {
    this.name = name;
  }

  greet() {
    return `Hello ${this.name}`;
  }
}

const user = new User('Hernando');
console.log(user.greet());
```

ADVANTAGES OF OOP

- ✓ Great for modeling real-world systems
- ✓ Organizes large codebases
- ✓ Works well with SOLID principles

DISADVANTAGES

- × Can lead to overengineering
- × Inheritance complexity
- × Harder to refactor

KEY DIFFERENCES

Aspect	Functional	OOP
Focus	Data flow	Objects
State	Immutable	Mutable
Structure	Functions	Classes
Style	Declarative	Imperative

REAL NODE.JS USAGE

FUNCTIONAL IS USED FOR:

- Data transformations
- Middleware pipelines
- Utility functions

OOP IS USED FOR:

- Services
- Domain models
- Complex systems

HYBRID APPROACH (BEST IN NODE.JS)

Most real-world Node.js apps use **both**.

EXAMPLE (HYBRID)

```
// Functional utility
const formatUser = user => ({
  ...user,
  name: user.name.toUpperCase()
});

// OOP service
class UserService {
  constructor(db) {
    this.db = db;
```

```
  }

  async getUser(id) {
    const user = await this.db.find(id);
    return formatUser(user);
  }
}
```

This gives:

- Clean logic (FP)
- Structured architecture (OOP)

WHEN TO USE EACH

USE FUNCTIONAL WHEN:

- Transforming data
- Writing utilities
- Handling async flows
- You want simplicity

USE OOP WHEN:

- Modeling business logic
- Managing stateful systems
- Applying SOLID principles

ANTI-PATTERNS

- ✖ Forcing OOP everywhere
- ✖ Overusing classes
- ✖ Writing overly abstract FP code
- ✖ Ignoring readability

REAL-WORLD INSIGHT

Senior developers don't argue: "FP vs OOP"

They ask: "What's the simplest and most maintainable solution here?"

MENTAL MODEL

- Use **functions for behavior**
- Use **objects for structure**
- Combine both intentionally

KEY TAKEAWAYS

- Node.js supports **multiple paradigms**
- FP = simplicity & predictability
- OOP = structure & scalability
- Best approach = **hybrid architecture**
- Choose based on the problem, not preference

COMPOSITION OVER INHERITANCE

THE CORE IDEA

"Favor composition over inheritance."

In Node.js, this principle means:

- Instead of building deep class hierarchies
- You **combine small, reusable pieces of behavior**

WHAT IS INHERITANCE?

Inheritance creates a hierarchy:

```
class Animal {
  eat() {}
}
```

```
class Dog extends Animal {
  bark() {}
}
```

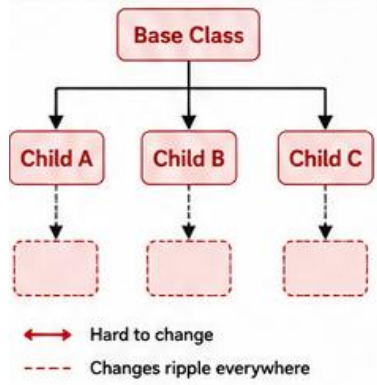
Behavior is passed **top → down**

PROBLEM WITH INHERITANCE

As systems grow:

- Tight coupling
- Fragile hierarchies
- Hard to modify
- “Ripple effects” on changes

INHERITANCE TRAP VISUALIZATION



WHAT IS COMPOSITION?

Composition = building objects by combining behaviors.

Instead of:

```
class Dog extends Animal {}
```

You do:.

```
const canEat = () => ({
  eat: () => console.log('eating')
});

const canBark = () => ({
  bark: () => console.log('barking')
});

const createDog = () => ({
  ...canEat(),
  ...canBark()
});
```

WHY COMPOSITION WINS

FLEXIBILITY

- Mix only what you need

REUSABILITY

- Share behaviors across different objects

LOOSE COUPLING

- No rigid hierarchy

EASIER TESTING

- Test small units independently

COMPOSITION VS INHERITANCE

Aspect	Inheritance	Composition
Structure	Hierarchical	Flexible
Coupling	Tight	Loose
Reusability	Limited	High
Maintainability	Harder	Easier

REAL NODE.JS EXAMPLE

INHERITANCE-BASED SERVICE

```
class BaseService {
  log() {}
}

class UserService extends BaseService {
  getUser() {}
}
```

COMPOSITION-BASED SERVICE

```
const withLogger = (service) => ({
  ...service,
  log: (msg) => console.log(msg)
});

const createUserService = (db) => {
  const service = {
    getUser: (id) => db.find(id)
  };

  return withLogger(service);
};
```

You can now:

- Add/remove features dynamically
- Reuse behaviors across services

FUNCTIONAL COMPOSITION (POWER MOVE)

```
const compose = (...fns) => (x) =>
  fns.reduceRight((v, f) => f(v), x);
```

```
const addTax = x => x * 1.19;
const applyDiscount = x => x * 0.9;

const finalPrice = compose(addTax, applyDiscount);

console.log(finalPrice(100));
```

This is **clean, reusable, and scalable logic**

COMPOSITION PATTERNS IN NODE.JS

1. MIXINS

```
const canLog = {
  log() { console.log('log'); }
};
```

2. HIGHER-ORDER FUNCTIONS

```
const withAuth = (handler) => (req, res) => {
  // auth logic
  return handler(req, res);
};
```

3. MIDDLEWARE (EXPRESS STYLE)

One of the best real-world examples of composition:

```
app.use(authMiddleware);
app.use(loggingMiddleware);
```

Each function:

- Adds behavior
- Stacks cleanly

WHEN INHERITANCE IS STILL OK

Use inheritance when:

- ✓ There is a **true "is-a" relationship**
- ✓ Behavior is stable
- ✓ Hierarchy is simple

Example:

```
class Error {}
class ValidationError extends Error {}
```

ANTI-PATTERNS

- × Deep inheritance trees
- × "God classes"
- × Forcing everything into classes
- × Over-composing (too many layers)

REAL-WORLD INSIGHT

Modern JavaScript (and Node.js) is moving toward:

Functional composition + small modules

Even large systems (APIs, microservices) rely heavily on:

- Middleware
- Hooks
- Plugins

All of these are **composition patterns**.

MENTAL MODEL

Instead of asking: "What should this class extend?"

Ask: "What behaviors can I compose together?"

KEY TAKEAWAYS

- Composition builds systems from **small reusable pieces**
- Avoid deep inheritance hierarchies
- Prefer **functions + composition** in Node.js
- Use inheritance only when it truly fits
- Composition leads to **flexible and scalable systems**

SEPARATION OF CONCERNS (SOC)

THE CORE IDEA

“Each part of the system should focus on a single concern.”

In Node.js, this means:

- Don’t mix responsibilities
- Keep logic **organized and isolated**
- Make systems easier to **scale and maintain**

WHAT IS A “CONCERN”?

A **concern** is a specific responsibility in your application:

- Handling HTTP requests
- Business logic
- Database access
- Authentication
- Logging

Each of these should live in **separate layers or modules**.

THE PROBLEM WITHOUT SOC

Everything mixed together:

```
app.post('/users', async (req, res) => {
  // validation
```

```
  if (!req.body.email) return res.status(400).send();

  // business logic
  const user = { ...req.body };

  // database
  await db.users.insert(user);

  // side effect
  await sendEmail(user.email);

  res.send(user);
});
```

This creates:

- Tight coupling
- Hard-to-test code
- Difficult maintenance

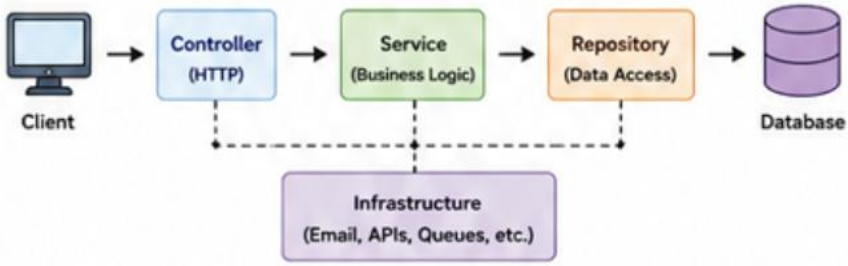
CLEAN SEPARATION (LAYERED APPROACH)

Split responsibilities:

```
// controller
app.post('/users', userController.create);

// controller
const userController = {
  create: async (req, res) => {
    const user = await userService.create(req.body);
    res.send(user);
  }
};

// service
const userService = {
```

```
  create: async (data) => {
    validateUser(data);
    const user = await userRepository.save(data);
    await emailService.send(user.email);
    return user;
  }
};
```

ARCHITECTURE VISUALIZATION



COMMON LAYERS IN NODE.JS

1. CONTROLLER LAYER

- Handles HTTP requests/responses
- No business logic

2. SERVICE LAYER

- Core business logic
- Orchestrates operations

3. REPOSITORY LAYER

- Database interaction
- Data persistence

4. INFRASTRUCTURE LAYER

- External services (email, APIs, queues)

BENEFITS OF SEPARATION OF CONCERNS

- ✓ Easier testing
- ✓ Better readability
- ✓ Independent changes
- ✓ Scalable architecture

You can change:

- Database → without touching business logic
- API → without touching persistence

SOC + OTHER PRINCIPLES

SoC works together with:

- SOLID principles → structure
- DRY → avoid duplication
- Composition → flexible design

This is how real systems are built.

REAL-WORLD EXAMPLE (API FLOW)

```
Request → Controller → Service → Repository → Database
```

Each layer:

- Has a single responsibility
- Can be tested independently

FOLDER STRUCTURE EXAMPLE

```
src/
  modules/
    user/
      user.controller.js
      user.service.js
      user.repository.js
  shared/
  config/
```

Organized by **feature**, not by type.

CROSS-CUTTING CONCERNS

Some concerns affect multiple parts:

- Logging
- Authentication
- Error handling

Handle them with:

- Middleware
- Decorators
- Composition

EXAMPLE (MIDDLEWARE)

```
app.use(authMiddleware);
app.use(loggerMiddleware);
```

Keeps core logic clean

ANTI-PATTERNS

- ✗ God files (everything in one place)
- ✗ Fat controllers
- ✗ Business logic in routes

- ✗ Direct DB access everywhere

WHEN NOT TO OVERDO IT

⚠️ Too much separation can hurt:

- Small apps don't need many layers
- Avoid unnecessary abstractions

Balance with:

- **KISS**
- **YAGNI**

REAL-WORLD INSIGHT

Bad systems look like:

- ✗ "Everything depends on everything"

Good systems look like:

- ✓ "Each piece does one job well"
- ✓ That's Separation of Concerns in action.

MENTAL MODEL

Before writing code, ask:

- What is this piece responsible for?
- Does it belong here?
- Can it be isolated?

KEY TAKEAWAYS

- Separate responsibilities into **clear layers**
- Avoid mixing logic (HTTP, business, DB)

- Use controllers, services, repositories
- Handle cross-cutting concerns with middleware
- Balance structure with simplicity

CREATIONAL PATTERNS

FACTORY PATTERN

THE CORE IDEA

"Create objects without exposing the creation logic."

Instead of using `new` everywhere, you centralize object creation in a **factory**.

In Node.js, this is extremely useful for:

- Managing complexity
- Creating flexible systems
- Decoupling code

THE PROBLEM WITHOUT A FACTORY

Direct instantiation everywhere:

```
const userService = new UserService(new MySQLDatabase());
const orderService = new OrderService(new MongoDatabase());
```

Problems:

- Tight coupling
- Hard to change dependencies
- Repetition

BASIC FACTORY PATTERN

Centralized creation:

```
function createUser(name) {
  return {
```

```
    name,
    createdAt: new Date()
  };
}

const user = createUser('Hernando');
```

Simple but powerful.

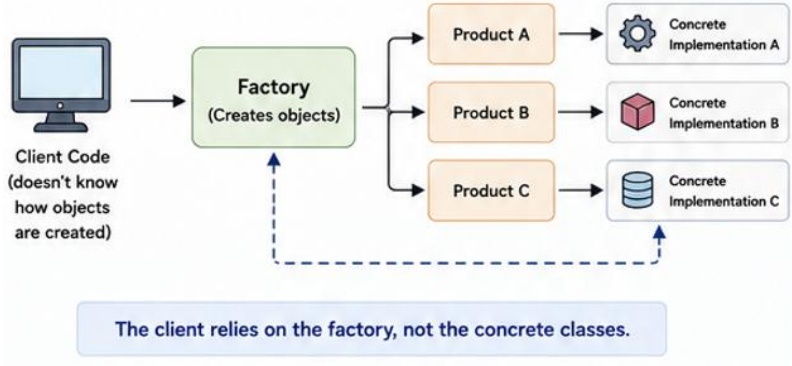
FACTORY WITH CONDITIONAL LOGIC

```
function createPayment(method) {
  if (method === 'stripe') {
    return new StripePayment();
  }

  if (method === 'paypal') {
    return new PayPalPayment();
  }
}
```

Now object creation is:

- Centralized
- Controlled

FACTORY PATTERN VISUALIZATION



REAL NODE.JS EXAMPLE (SERVICE FACTORY)

```
function createUserService(dbType) {
  const db =
    dbType === 'mongo'
      ? new MongoDB()
      : new MySQLDatabase();

  return {
    getUser: (id) => db.find(id)
  };
}

const userService = createUserService('mongo');
```

You can switch databases **without changing business logic**.

FACTORY + DEPENDENCY INJECTION

```
function createUserService({ db, logger }) {
  return {
    getUser: (id) => {
      logger.log('Fetching user');
      return db.find(id);
    }
  };
}
```

This is a **modern Node.js pattern**:

- Factory builds the object
- Dependencies are injected

CLASS-BASED FACTORY

```
class PaymentFactory {
  static create(type) {
    switch (type) {
```

```
      case 'stripe':
        return new StripePayment();
      case 'paypal':
        return new PayPalPayment();
      default:
        throw new Error('Invalid type');
    }
  }
}
```

FUNCTIONAL FACTORY

```
const createLogger = (level) => ({
  log: (msg) => console.log(`[${level}] ${msg}`)
});

const logger = createLogger('INFO');
```

Cleaner, simpler, more flexible.

WHEN TO USE FACTORY PATTERN

Use it when:

- ✓ Object creation is complex
- ✓ You need different variations
- ✓ You want to decouple dependencies
- ✓ You want testable code

REAL-WORLD USE CASES

- Database connectors
- Payment systems
- Logger systems

- API clients
- Service initialization

ANTI-PATTERNS

- ✗ Overengineering simple objects
- ✗ Huge “God factory” with too many conditions
- ✗ Using factories where simple functions work

FACTORY VS CONSTRUCTOR

Approach	Use Case	Approach
Constructor (`new`)	Simple objects	Constructor (`new`)
Factory	Flexible, dynamic creation	Factory

REAL-WORLD INSIGHT

Factories are everywhere in Node.js:

- Frameworks
- Libraries
- Dependency injection systems

They allow systems to be:

- Configurable
- Testable
- Scalable

MENTAL MODEL

Instead of: “Where do I create this object?”

Think: “Who is responsible for creating this object?”

Answer: **The factory**

KEY TAKEAWAYS

- Factory centralizes object creation
- Reduces coupling
- Improves flexibility
- Works great with dependency injection
- Prefer **functional factories** in Node.js

ABSTRACT FACTORY PATTERN

THE CORE IDEA

"Provide an interface to create **families of related objects** without specifying their concrete classes."

If the **Factory Pattern** creates ONE OBJECT, the **Abstract Factory** creates **multiple related objects that must work together**.

In Node.js, this is powerful when building:

- Multi-environment systems
- Pluggable architectures
- Cross-platform services

THE PROBLEM IT SOLVES

Imagine you support multiple databases:

Without Abstract Factory:

```
const db = new MySQLDatabase();
const userRepo = new MySQLUserRepository();
const orderRepo = new MySQLOrderRepository();
```

Switching to Mongo means:

- Rewriting everything
- Risk of mixing incompatible components

THE SOLUTION: ABSTRACT FACTORY

Create a **factory that produces a family of related components**.

EXAMPLE

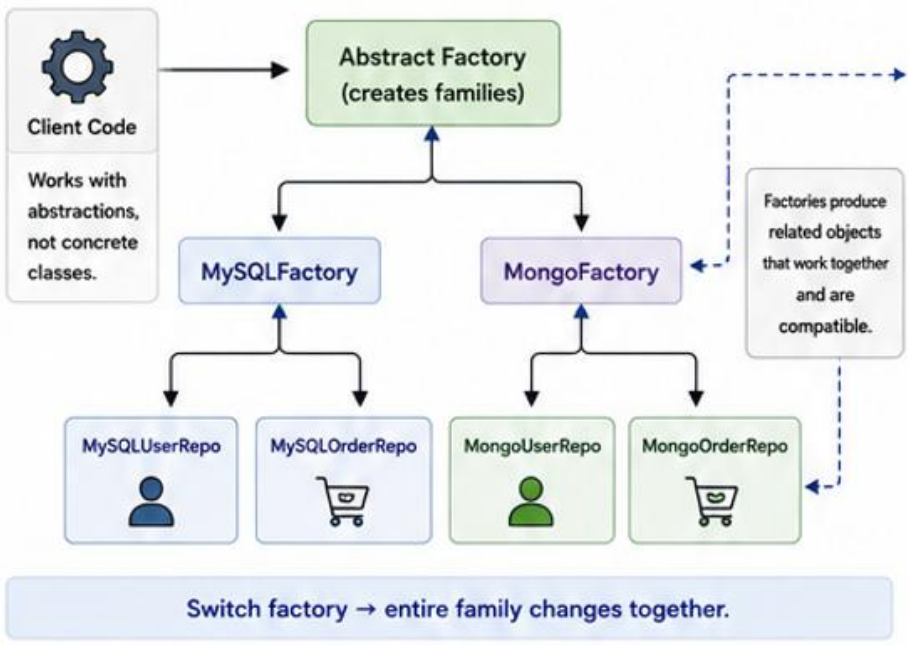
```
class MySQLFactory {
  createUserRepository() {
    return new MySQLUserRepository();
  }

  createOrderRepository() {
    return new MySQLOrderRepository();
  }
}

class MongoFactory {
  createUserRepository() {
    return new MongoUserRepository();
  }

  createOrderRepository() {
    return new MongoOrderRepository();
  }
}
```

USAGE:

```
function initApp(factory) {
  const userRepo = factory.createUserRepository();
  const orderRepo = factory.createOrderRepository();

  return { userRepo, orderRepo };
}

const factory = new MongoFactory();
const app = initApp(factory);
```

Now everything is **consistent and swappable**.

VISUALIZATION



FUNCTIONAL ABSTRACT FACTORY (NODE.JS STYLE)

Prefer this in modern Node.js:

```
const createMySQLFactory = () => ({
  createUserRepository: () => ({
    find: (id) => `MySQL user ${id}`
  }),
  createOrderRepository: () => ({
    find: (id) => `MySQL order ${id}`
  })
});
```

```
const createMongoFactory = () => ({
  createUserRepository: () => ({
    find: (id) => `Mongo user ${id}`
  }),
  createOrderRepository: () => ({
    find: (id) => `Mongo order ${id}`
  })
});
```

USAGE:

```
const factory = createMongoFactory();

const userRepo = factory.createUserRepository();
console.log(userRepo.find(1));
```

REAL-WORLD EXAMPLE

MULTI-ENVIRONMENT APP

```
const factories = {
  dev: createMockFactory,
  prod: createMySQLFactory
};

const factory = factories[process.env.NODE_ENV]();
```

Same app, different implementations.

BENEFITS

- ✓ Ensures compatibility between components
- ✓ Makes systems easily swappable
- ✓ Reduces tight coupling
- ✓ Improves scalability

WHEN TO USE IT

Use Abstract Factory when:

- ✓ You have **families of related objects**
- ✓ You need **consistency across components**
- ✓ You want **environment-based configurations**
- ✓ You are building frameworks or libraries

ABSTRACT FACTORY VS FACTORY

Pattern	Purpose	Pattern
Factory	Creates one object	Factory
Abstract Factory	Creates families of objects	Abstract Factory

ANTI-PATTERNS

- × Overengineering simple systems
- × Too many factories (complexity explosion)
- × Using it when only one object is needed

REAL-WORLD INSIGHT

Abstract Factory is used in:

- ORMs (database switching)
- UI frameworks (themes/components)
- Cloud SDKs (AWS/GCP adapters)

It's a **scaling pattern**, not a beginner one.

MENTAL MODEL

Instead of:

“How do I create this object?”

Think:

“How do I create a **set of related objects that must work together**?”

KEY TAKEAWAYS

- Abstract Factory creates **families of objects**
- Ensures consistency across components
- Enables easy swapping of implementations
- Works best in **large, scalable systems**
- Prefer **functional factories** in Node.js

BUILDER PATTERN

THE CORE IDEA

“Construct complex objects step by step.”

The **Builder Pattern** separates:

- **How an object is built**
- From **what the object is**

In Node.js, this is useful when objects:

- Have many optional fields
- Require configuration
- Are built incrementally

THE PROBLEM IT SOLVES

Messy object creation:

```
const user = new User(
  'Hernando',
  'hernando@email.com',
```

```
  true,
  'admin',
  'theme-dark',
  true
);
```

Problems:

- Hard to read
- Easy to break
- Order matters (dangerous)

BUILDER PATTERN SOLUTION

Step-by-step construction:

```
class UserBuilder {
  constructor(name) {
    this.user = { name };
  }

  setEmail(email) {
    this.user.email = email;
    return this;
  }

  setRole(role) {
    this.user.role = role;
    return this;
  }

  setActive(isActive) {
    this.user.isActive = isActive;
    return this;
  }

  build() {
```

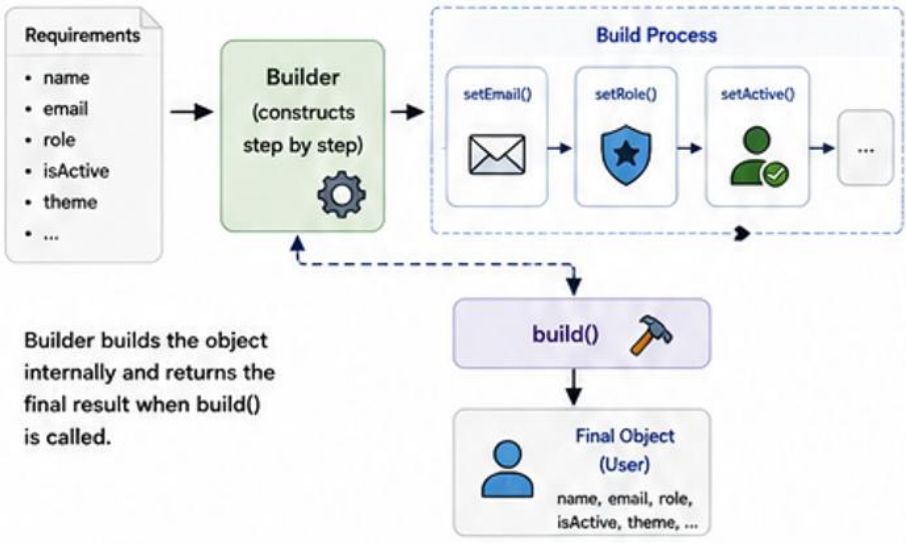
```
    return this.user;
  }
}
```

USAGE:

```
const user = new UserBuilder('Hernando')
  .setEmail('hernando@email.com')
  .setRole('admin')
  .setActive(true)
  .build();
```

Clean, readable, flexible.

VISUALIZATION



FLUENT INTERFACE (KEY FEATURE)

Builder uses **method chaining**:

```
builder.setA().setB().setC();
```

This is called a **fluent interface**.

FUNCTIONAL BUILDER (NODE.JS STYLE)

Preferred in modern Node.js:

```
const createUserBuilder = (name) => {
  const user = { name };

  return {
    setEmail: (email) => {
      user.email = email;
      return this;
    },
    setRole: (role) => {
      user.role = role;
      return this;
    },
    build: () => user
  };
};
```

REAL-WORLD EXAMPLE

API QUERY BUILDER

```
class QueryBuilder {
  constructor() {
    this.query = {};
  }

  select(fields) {
    this.query.fields = fields;
    return this;
  }

  where(condition) {
```

```
    this.query.where = condition;
    return this;
  }

  limit(n) {
    this.query.limit = n;
    return this;
  }

  build() {
    return this.query;
  }
}
```

USAGE:

```
const query = new QueryBuilder()
  .select(['name', 'email'])
  .where({ active: true })
  .limit(10)
  .build();
```

WHEN TO USE BUILDER PATTERN

Use it when:

- ✓ Object has many optional parameters
- ✓ Order of parameters matters
- ✓ You want readable construction
- ✓ You need step-by-step configuration

BUILDER VS FACTORY

Pattern	Purpose	Pattern
Factory	Creates objects in one step	Factory

Builder	Creates objects step-by-step	Builder

ANTI-PATTERNS

- × Using builder for simple objects
- × Overengineering small configs
- × Too many chained methods

ADVANCED USE CASES

- HTTP request builders
- Database query builders
- Config generators
- SDK design
- Test data builders

REAL-WORLD INSIGHT

Many popular libraries use Builder:

- ORMs (query builders)
- HTTP clients
- Cloud SDKs

It improves:

- Developer experience
- Readability
- Flexibility

MENTAL MODEL

Instead of: “Pass everything at once”

Think: “Build it step by step”

KEY TAKEAWAYS

- Builder constructs objects **step by step**
- Improves readability and flexibility
- Uses **fluent interfaces (method chaining)**
- Ideal for complex configurations
- Prefer simple solutions for simple cases

SINGLETON PATTERN

THE CORE IDEA

"Ensure a class has only **one instance**, and provide a global access point to it."

In Node.js, this pattern is extremely common because:

- Modules are **cached after first load**
- Shared resources are needed (DB, config, logger)

THE PROBLEM IT SOLVES

Some resources should NOT be created multiple times:

- Database connections
- Logger instances
- Configuration managers

Without Singleton:

```
const db1 = new Database();
const db2 = new Database(); // wasteful, dangerous
```

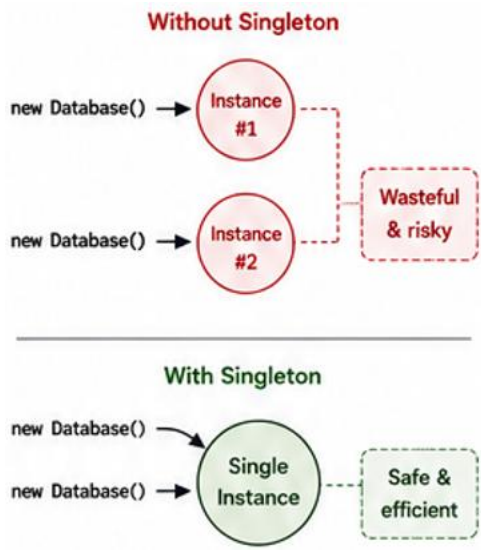
Problems:

- Memory waste
- Conflicts
- Multiple connections

BASIC SINGLETON IMPLEMENTATION

```
class Database {
  constructor() {
    if (Database.instance) {
      return Database.instance;
    }

    this.connection = 'Connected';
    Database.instance = this;
  }
}

const db1 = new Database();
const db2 = new Database();

console.log(db1 === db2); // true
```

VISUALIZATION



NODE.JS NATIVE SINGLETON (IMPORTANT INSIGHT)

In Node.js, you often **don't need to implement Singleton manually**.

Because: Modules are cached after the first `require()` or `import`.

EXAMPLE

```
// app.js
const db1 = require('./db');
const db2 = require('./db');

console.log(db1 === db2); // true
```

This is already a **Singleton**.

REAL-WORLD EXAMPLES

1. DATABASE CONNECTION

```
let instance;

function createDB() {
  if (!instance) {
    instance = { connected: true };
  }
  return instance;
}

module.exports = createDB();
```

2. LOGGER

```
class Logger {
  log(msg) {
    console.log(msg);
  }
}
```

```
module.exports = new Logger();
```

3. CONFIG MANAGER

```
module.exports = {
  port: process.env.PORT || 3000
};
```

WHEN TO USE SINGLETON

Use it when:

✓ You need **one shared instance**
✓ Resource is expensive to create
✓ State must be shared globally

WHEN NOT TO USE IT

❌ When you need multiple instances
❌ When it introduces hidden dependencies
❌ When testing becomes difficult

COMMON PITFALLS

HIDDEN GLOBAL STATE

```
const db = require('./db'); // global dependency
```

Hard to test and mock.

TIGHT COUPLING

Everything depends on the same instance → less flexibility.

BETTER ALTERNATIVE (DEPENDENCY INJECTION)

```
function createUserService(db) {
  return {
```

```
    getUser: (id) => db.find(id)
  };
}
```

More flexible and testable.

SINGLETON VS FACTORY

Pattern	Purpose
Singleton	One shared instance
Factory	Flexible object creation

REAL-WORLD INSIGHT

Singleton is one of the most:

Used patterns
Misused patterns

Senior developers are careful because:

- It introduces **global state**
- It can make systems rigid

MENTAL MODEL

Instead of asking: “Can I make this a singleton?”

Ask: “Should this exist only once in the system?”

KEY TAKEAWAYS

- Singleton ensures **one instance only**
- Node.js modules already behave like singletons
- Useful for shared resources (DB, config, logger)
- Can introduce hidden complexity
- Prefer dependency injection when flexibility is needed

PROTOTYPE PATTERN

THE CORE IDEA

"Create new objects by **cloning existing ones** instead of building from scratch."

In Node.js, this pattern is especially natural because JavaScript is **prototype-based**, not class-based at its core.

WHY IT EXISTS

Sometimes creating objects is:

- Expensive (complex setup)
- Repetitive
- Requires default configurations

Instead of recreating everything, you **clone a base object (prototype)**.

THE JAVASCRIPT REALITY

In JavaScript, **everything already uses prototypes**:

```
const obj = {};
console.log(obj.__proto__);
```

Every object inherits from another object.

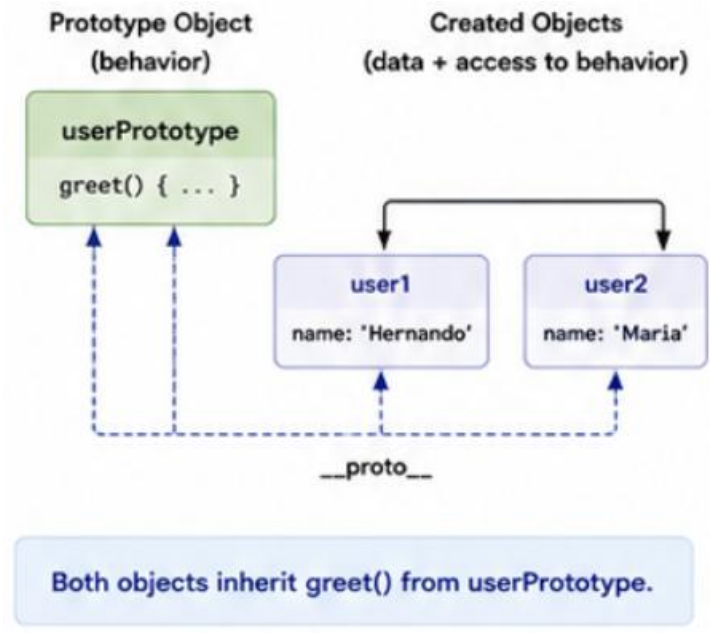
BASIC PROTOTYPE PATTERN

```
const userPrototype = {
  greet() {
    return `Hello ${this.name}`;
  }
};
```

```
const user1 = Object.create(userPrototype);
user1.name = 'Hernando';

console.log(user1.greet());
```

`user1` **inherits behavior** from `userPrototype`.

VISUALIZATION



CLONING OBJECTS (CORE CONCEPT)

SHALLOW CLONE

```
const original = { name: 'Hernando' };
const clone = { ...original };
```

DEEP CLONE

```
const clone = JSON.parse(JSON.stringify(original));
```

Copies nested data as well.

REAL PROTOTYPE PATTERN EXAMPLE

```
const carPrototype = {
  start() {
    console.log(`${this.model} started`);
  }
};

function createCar(model) {
  const car = Object.create(carPrototype);
  car.model = model;
  return car;
}

const car1 = createCar('Tesla');
const car2 = createCar('BMW');
```

Behavior is reused without redefining methods.

PROTOTYPE VS CLASS

```
class User {
  greet() {}
}
```

Under the hood, this is still **prototype-based**.

EQUIVALENT:

```
function User() {}

User.prototype.greet = function () {};
```

WHEN TO USE PROTOTYPE PATTERN

Use it when:

- Object creation is expensive
- You need many similar objects
- You want to reuse behavior efficiently
- You want lightweight inheritance

REAL-WORLD USE CASES

- Object templates
- Game development (entities)
- UI component reuse
- Caching pre-configured objects

PROTOTYPE VS FACTORY VS BUILDER

Pattern	Purpose
Prototype	Clone existing objects
Factory	Create new objects
Builder	Construct step-by-step

ADVANTAGES

- ✓ Fast object creation
- ✓ Memory efficient (shared methods)
- ✓ Native to JavaScript

DISADVANTAGES

- × Harder to understand
- × Debugging prototype chains can be tricky

- ✖ Deep cloning can be expensive

SHARED MUTABLE STATE

```
const proto = { items: [] };

const a = Object.create(proto);
const b = Object.create(proto);

a.items.push(1);

console.log(b.items); // also affected
```

Because both share the same reference.

FIX

```
const createObj = () => ({
  items: []
});
```

REAL-WORLD INSIGHT

Most developers don't explicitly use "Prototype Pattern"…

But they use it **indirectly all the time**:

- Classes
- Object.create
- Inheritance

Understanding it gives you:

- Deep JavaScript knowledge
- Better debugging skills
- More control over performance

MENTAL MODEL

Instead of: “Create from scratch”

Think: “Clone and extend what already exists”

Key Takeaways

- Prototype pattern is **native to JavaScript**
- Objects inherit behavior via prototype chain
- Useful for cloning and reuse
- Be careful with shared references
- Powers how JavaScript actually works under the hood

STRUCTURAL PATTERNS

ADAPTER PATTERN

THE CORE IDEA

“Convert the interface of a class into another interface the client expects.”

The **Adapter Pattern** allows incompatible systems to **work together without changing their code**.

In Node.js, this is extremely common when integrating:

- Third-party APIs
- Legacy systems
- Different libraries

THE PROBLEM IT SOLVES

You have two systems with **different interfaces**:

```
// New system expects:
payment.process(amount);
```

But your existing system provides:

```
stripe.makePayment(value);
```

They don't match → integration breaks.

THE ADAPTER SOLUTION

Create a wrapper that translates:

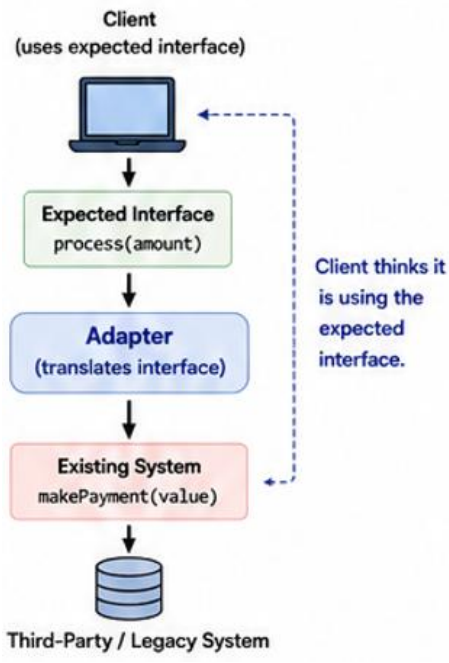
```
class StripeAdapter {
  constructor(stripe) {
    this.stripe = stripe;
  }

  process(amount) {
    return this.stripe.makePayment(amount);
  }
}
```

USAGE:

```
const stripe = new Stripe();
const payment = new StripeAdapter(stripe);

payment.process(100);
```

Now both systems work together seamlessly.

VISUALIZATION



FUNCTIONAL ADAPTER (NODE.JS STYLE)

Preferred in modern Node.js:

```
const stripeAdapter = (stripe) => ({
  process: (amount) => stripe.makePayment(amount)
});

const payment = stripeAdapter(stripe);
payment.process(100);
```

✓ Simple
✓ Flexible
✓ Composable

REAL-WORLD EXAMPLE

MULTIPLE PAYMENT PROVIDERS

```
const stripeAdapter = (stripe) => ({
  pay: (amount) => stripe.charge(amount)
});

const paypalAdapter = (paypal) => ({
  pay: (amount) => paypal.send(amount)
});
```

UNIFIED USAGE:

```
function processPayment(provider, amount) {
  return provider.pay(amount);
}
```

Now all providers share the same interface.

API RESPONSE ADAPTER

External API:

```
{
  "user_name": "Hernando"
}
```

ADAPTER:

```
function adaptUser(apiResponse) {
  return {
    name: apiResponse.user_name
  };
}
```

Keeps your internal system clean.

ADAPTER VS FAÇADE

Pattern	Purpose

Adapter	Convert interface
Facade	Simplify interface

WHEN TO USE ADAPTER PATTERN

Use it when:

✓ Integrating third-party libraries
✓ Working with legacy code
✓ Standardizing different APIs
✓ Avoiding changes to existing code

BENEFITS

✓ Decouples systems
✓ Improves flexibility
✓ Makes integrations clean
✓ Avoids modifying existing code

ANTI-PATTERNS

❌ Overusing adapters for simple cases
❌ Creating unnecessary abstraction layers
❌ Hiding too much logic inside adapters

REAL-WORLD INSIGHT

Adapters are everywhere in Node.js:

- Payment integrations
- API clients
- Database drivers
- Microservices communication

They act as **translation layers** between systems.

MENTAL MODEL

Instead of: “Change the existing system”

Think: “Wrap it and adapt it”

KEY TAKEAWAYS

- Adapter converts one interface into another
- Enables incompatible systems to work together
- Common in integrations and APIs
- Prefer functional adapters in Node.js
- Keeps your core system clean and stable

DECORATOR PATTERN

THE CORE IDEA

“Add behavior to an object **dynamically**, without modifying its original structure.”

The **Decorator Pattern** lets you:

- Extend functionality
- Keep code flexible
- Avoid modifying existing logic

In Node.js, this pattern appears everywhere—especially in:

- Middleware
- Logging
- Authentication
- API handlers

THE PROBLEM IT SOLVES

You have a function:

```
function getUser(id) {
```

```
  return db.find(id);
}
```

Now you want to:

- Add logging
- Add authentication
- Add caching

Without decorator:

```
function getUser(id) {
  console.log('Fetching user');
  // auth check
  return db.find(id);
}
```

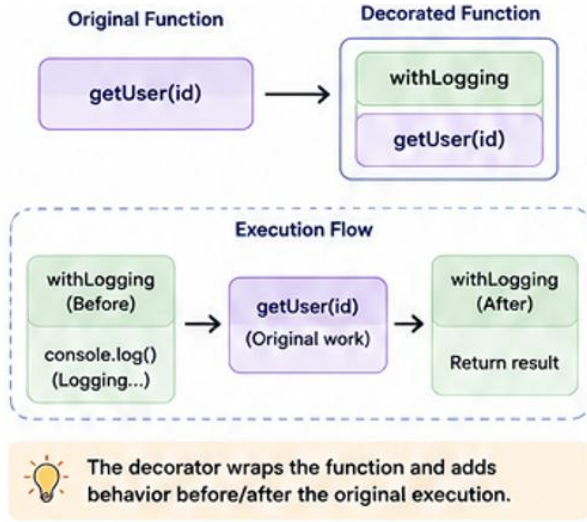
This mixes concerns and breaks clean architecture.

THE DECORATOR SOLUTION

✓ Wrap the function instead of modifying it:

```
function withLogging(fn) {
  return function (...args) {
    console.log('Calling function...');
    return fn(...args);
  };
}

const getUserWithLogging = withLogging(getUser);
```

VISUALIZATION



MULTIPLE DECORATORS (STACKING)

You can layer behaviors:

```
const withAuth = (fn) => (...args) => {
  console.log('Auth check');
  return fn(...args);
};

const withCache = (fn) => (...args) => {
  console.log('Cache check');
  return fn(...args);
};

const enhanced = withAuth(withCache(getUser));
```

Execution order:

```
Auth → Cache → getUser
```

REAL NODE.JS EXAMPLE (EXPRESS MIDDLEWARE)

This is basically the **Decorator Pattern in action**:

```
app.use(authMiddleware);
app.use(loggerMiddleware);

app.get('/user', handler);
```

Each middleware:

- Wraps the request
- Adds behavior
- Passes control forward

CLASS-BASED DECORATOR

```
class LoggerDecorator {
  constructor(service) {
    this.service = service;
  }

  getUser(id) {
    console.log('Logging...');
    return this.service.getUser(id);
  }
}
```

FUNCTIONAL DECORATOR (PREFERRED)

```
const withTiming = (fn) => async (...args) => {
  const start = Date.now();
  const result = await fn(...args);
  console.log('Time:', Date.now() - start);
  return result;
};

const getUserTimed = withTiming(getUser);
```

✓ Simple
✓ Composable
✓ Very Node.js-friendly

REAL-WORLD USE CASES

- Logging
- Authentication
- Authorization
- Caching
- Rate limiting
- Metrics

DECORATOR VS INHERITANCE

Approach	Behavior
Inheritance	Static, rigid
Decorator	Dynamic, flexible

Decorators win in most Node.js systems.

BENEFITS

✓ No modification of original code
✓ Composable behavior
✓ Flexible architecture
✓ Works great with functions

ANTI-PATTERNS

❌ Too many nested decorators (hard to debug)
❌ Hidden side effects
❌ Losing clarity of execution flow

REAL-WORLD INSIGHT

Modern backend systems are basically:

Chains of decorators

Examples:

- Express middleware
- API pipelines
- GraphQL resolvers
- Serverless handlers

MENTAL MODEL

Instead of:

“Modify the function”

Think:

“Wrap the function and enhance it”

KEY TAKEAWAYS

- Decorator adds behavior dynamically
- Keeps original code untouched
- Enables composition of features
- Core to middleware systems
- Prefer functional decorators in Node.js

PROXY PATTERN

THE CORE IDEA

“Provide a **surrogate (placeholder)** to control access to another object.”

The **Proxy Pattern** sits between the client and the real object, allowing you to:

- Control access

- Add logic before/after execution
- Optimize performance

In Node.js, proxies are used for:

- Caching
- Validation
- Logging
- Security

THE PROBLEM IT SOLVES

You have a direct call:

```
function fetchUser(id) {
  return database.find(id);
}
```

Now you want to:

- Cache results
- Restrict access
- Log usage

Without proxy:

- You modify the original function
- Mix concerns
- Break clean architecture

THE PROXY SOLUTION

Create an intermediary:

```
function createUserProxy(service) {
  const cache = new Map();

  return {
```

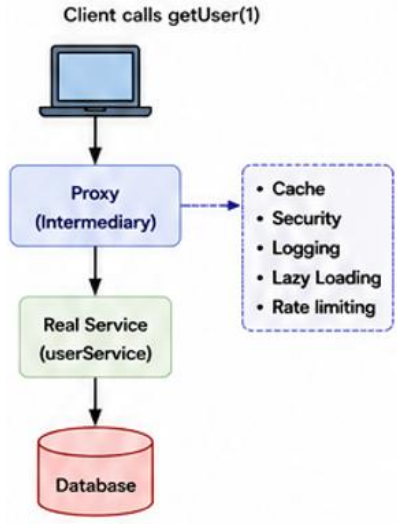
```
    getUser(id) {
      if (cache.has(id)) {
        console.log('From cache');
        return cache.get(id);
      }

      const result = service.getUser(id);
      cache.set(id, result);
      return result;
    }
  };
}
```

USAGE:

```
const userService = {
  getUser: (id) => database.find(id)
};

const proxy = createUserProxy(userService);

proxy.getUser(1);
proxy.getUser(1); // cached
```

You added caching **without changing the original service**.

VISUALIZATION



JAVASCRIPT NATIVE PROXY (POWER FEATURE)

JavaScript has a built-in Proxy object.

EXAMPLE:

```
const user = { name: 'Hernando' };

const proxy = new Proxy(user, {
  get(target, prop) {
    console.log(`Accessing ${prop}`);
    return target[prop];
  }
});
```

```
console.log(proxy.name);
```

Intercepts property access dynamically.

TYPES OF PROXIES

1. VIRTUAL PROXY (LAZY LOADING)

```
const createImageProxy = () => {
  let image = null;

  return {
    display() {
      if (!image) {
        image = loadImage();
      }
      return image;
    }
  };
};
```

2. PROTECTION PROXY

```
const secureService = (service) => ({
  getUser: (id, role) => {
    if (role !== 'admin') {
      throw new Error('Unauthorized');
    }
    return service.getUser(id);
  }
});
```

3. CACHING PROXY

Already shown — improves performance

PROXY VS DECORATOR

Pattern	Purpose
Decorator	Add behavior
Proxy	Control access

They look similar but have different intent.

REAL-WORLD USE CASES

- API caching layers
- Authentication/authorization
- Rate limiting
- Lazy loading
- Logging access

BENEFITS

✓ Adds control without modifying original object
✓ Improves performance (caching)
✓ Enhances security
✓ Keeps architecture clean

ANTI-PATTERNS

❌ Overusing proxies → complexity
❌ Hidden behavior (hard debugging)
❌ Too many layers of indirection

REAL-WORLD INSIGHT

Proxies are heavily used in:

- ORMs
- Frontend frameworks (reactivity systems)
- Backend caching systems
- API gateways

They act as **gatekeepers**.

MENTAL MODEL

Instead of: “Call the object directly”

Think: “Place a controller in front of it”

KEY TAKEAWAYS

- Proxy controls access to an object
- Can add caching, security, logging
- JavaScript has built-in Proxy support
- Keeps original logic untouched
- Useful for performance and control

FACADE PATTERN

THE CORE IDEA

“Provide a **simple interface** to a complex system.”

The **Facade Pattern** hides complexity behind a clean, easy-to-use API.

In Node.js, this is extremely common when:

- Systems grow complex
- Multiple services need orchestration
- You want clean APIs

THE PROBLEM IT SOLVES

Imagine a complex flow:

```
const user = await db.getUser(id);
const orders = await db.getOrders(user.id);
const payment = await paymentService.getPayment(user.id);
const shipping = await shippingService.getShipping(user.id);
```

Problems:

- Too many dependencies
- Hard to reuse

- Complex logic everywhere

THE FACADE SOLUTION

Wrap complexity behind a single interface:

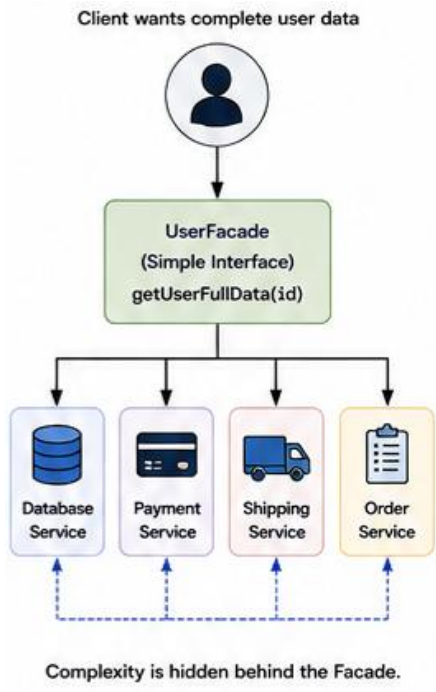
```
class UserFacade {
  constructor({ db, paymentService, shippingService }) {
    this.db = db;
    this.paymentService = paymentService;
    this.shippingService = shippingService;
  }

  async getUserFullData(id) {
    const user = await this.db.getUser(id);
    const orders = await this.db.getOrders(user.id);
    const payment = await this.paymentService.getPayment(user.id);
    const shipping = await this.shippingService.getShipping(user.id);

    return { user, orders, payment, shipping };
  }
}
```

USAGE:

```
const facade = new UserFacade({
  db,
  paymentService,
  shippingService
});

const data = await facade.getUserFullData(1);
```

Clean, simple, reusable.

VISUALIZATION



FUNCTIONAL FACADE (NODE.JS STYLE)

Preferred approach:

```
const createUserFacade = ({ db, payment, shipping }) => ({
  getFullUser: async (id) => {
    const user = await db.getUser(id);
    return {
      user,
      orders: await db.getOrders(id),
      payment: await payment.get(id),
      shipping: await shipping.get(id)
    };
  }
});
```

REAL-WORLD EXAMPLE

API ENDPOINT AS FACADE

```
app.get('/dashboard', async (req, res) => {
  const data = await dashboardFacade.getData(req.user.id);
  res.json(data);
});
```

The endpoint becomes a **facade over multiple services**.

FACADE VS ADAPTER VS PROXY

Pattern	Purpose
Facade	Simplify complex system
Adapter	Convert interface
Proxy	Control access

WHEN TO USE FACADE PATTERN

Use it when:

✓ You have complex subsystems
✓ You want a clean API
✓ You want to reduce coupling
✓ You want reusable workflows

BENEFITS

✓ Simplifies usage
✓ Reduces complexity
✓ Improves readability
✓ Decouples components

ANTI-PATTERNS

❌ “God Facade” (too much responsibility)
❌ Hiding too much logic (hard debugging)
❌ Overusing for simple systems

REAL-WORLD INSIGHT

Facades are everywhere in Node.js:

- API layers
- Service aggregators
- Microservice gateways
- SDK wrappers

They act as **entry points to complexity**.

MENTAL MODEL

Instead of: “Expose everything”

Think: “Expose only what’s necessary”

KEY TAKEAWAYS

- Facade simplifies complex systems
- Provides a clean, unified interface
- Reduces coupling between components
- Common in APIs and service layers
- Prefer functional facades in Node.js

BRIDGE PATTERN

THE CORE IDEA

“Decouple an abstraction from its implementation so the two can vary independently.”

The **Bridge Pattern** splits a system into:

- **Abstraction** → what the system does
- **Implementation** → how it does it

In Node.js, this helps avoid:

- Tight coupling
- Exploding class combinations
- Rigid architectures

THE PROBLEM IT SOLVES

Imagine you have:

- Multiple **notification types** (Email, SMS)
- Multiple **providers** (SendGrid, Twilio)

WITHOUT BRIDGE (EXPLOSION PROBLEM)

```
class EmailSendGrid {}
class EmailTwilio {}
class SMSSendGrid {}
class SMSTwilio {}
```

Problem:

- Combinatorial explosion
- Hard to scale

THE BRIDGE SOLUTION

Separate concerns:

- **Abstraction** → Notification
- **Implementation** → Provider

EXAMPLE

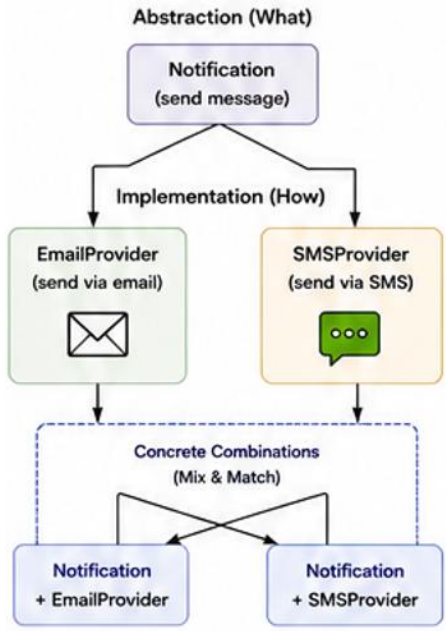
```
class Notification {
  constructor(provider) {
    this.provider = provider;
  }

  send(message) {
    return this.provider.send(message);
  }
}

class EmailProvider {
  send(message) {
    console.log('Sending Email:', message);
  }
}

class SMSProvider {
  send(message) {
    console.log('Sending SMS:', message);
  }
}
```

USAGE:

```
const email = new Notification(new EmailProvider());
email.send('Hello');

const sms = new Notification(new SMSProvider());
sms.send('Hi');
```

You can mix and match freely.

VISUALIZATION



FUNCTIONAL BRIDGE (NODE.JS STYLE)

Preferred in modern Node.js:

```
const createNotification = (provider) => ({
  send: (msg) => provider.send(msg)
});

const emailProvider = {
  send: (msg) => console.log('Email:', msg)
};

const smsProvider = {
  send: (msg) => console.log('SMS:', msg)
};
```

```
const notifier = createNotification(emailProvider);
notifier.send('Hello');
```

REAL-WORLD EXAMPLE

PAYMENT SYSTEM

```
const createPayment = (gateway) => ({
  pay: (amount) => gateway.process(amount)
});

const stripe = {
  process: (amount) => console.log('Stripe:', amount)
};

const paypal = {
  process: (amount) => console.log('PayPal:', amount)
};

const payment = createPayment(stripe);
payment.pay(100);
```

You can switch gateways without touching logic.

BRIDGE VS ADAPTER VS FAÇADE

Pattern	Purpose
Bridge	Decouple abstraction & implementation
Adapter	Convert interface
Facade	Simplify interface

WHEN TO USE BRIDGE PATTERN

Use it when:

✓ You have multiple dimensions of variation
✓ You want to avoid class explosion

✓ You want independent scalability
✓ You want flexible architecture

BENEFITS

✓ Decouples components
✓ Improves scalability
✓ Avoids duplication
✓ Flexible combinations

ANTI-PATTERNS

❌ Overengineering simple systems
❌ Too many abstraction layers
❌ Confusing structure if overused

REAL-WORLD INSIGHT

Bridge is used in:

- Payment systems
- Notification systems
- UI frameworks (themes vs components)
- Database drivers

It's a **scaling pattern** for growing systems.

MENTAL MODEL

Instead of: "Create combinations of everything"

Think: "Separate what changes independently"

KEY TAKEAWAYS

- Bridge separates abstraction from implementation

- Prevents class explosion
- Enables flexible combinations
- Works great in scalable systems
- Prefer functional bridges in Node.js

COMPOSITE PATTERN

THE CORE IDEA

"Treat individual objects and groups of objects **the same way**."

The **Composite Pattern** lets you build **tree-like structures** where:

- A single object (leaf)
- A group of objects (composite)

Both share the **same interface**.

In Node.js, this is useful for:

- File systems
- UI components
- Menu structures
- Nested data

THE PROBLEM IT SOLVES

Without Composite, you often write:

```
if (node.type === 'file') {
  // handle file
} else {
  // handle folder
}
```

Problems:

- Conditionals everywhere
- Hard to extend

- Not scalable

THE COMPOSITE SOLUTION

Unify behavior under one interface.

EXAMPLE

```
class File {
  constructor(name) {
    this.name = name;
  }

  display() {
    console.log(this.name);
  }
}

class Folder {
  constructor(name) {
    this.name = name;
    this.children = [];
  }

  add(child) {
    this.children.push(child);
  }

  display() {
    console.log(this.name);
    this.children.forEach(child => child.display());
  }
}
```

USAGE:

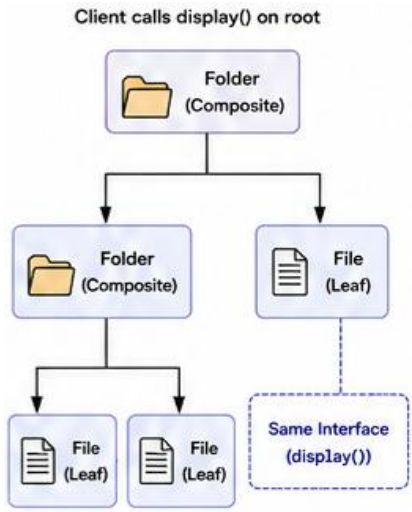
```
const file1 = new File('file1.txt');
```

```
const file2 = new File('file2.txt');

const folder = new Folder('Documents');
folder.add(file1);
folder.add(file2);

folder.display();
```

You treat:

- Files and folders the **same way** (`display()`)

VISUALIZATION



FUNCTIONAL COMPOSITE (NODE.JS STYLE)

Preferred approach:

```
const createFile = (name) => ({
  display: () => console.log(name)
});
```

```
const createFolder = (name) => {
  const children = [];

  return {
    add: (child) => children.push(child),
    display: () => {
      console.log(name);
      children.forEach(child => child.display());
    }
  };
};
```

REAL-WORLD EXAMPLE

MENU SYSTEM

```
const menu = createFolder('Main Menu');

menu.add(createFile('Home'));
menu.add(createFile('About'));

const settings = createFolder('Settings');
settings.add(createFile('Profile'));
settings.add(createFile('Security'));

menu.add(settings);

menu.display();
```

Nested structures handled uniformly.

KEY CONCEPT: TREE STRUCTURE

- Root → Parent
- Branch → Composite
- Leaf → Individual object

Everything follows the **same interface**.

COMPOSITE VS OTHER PATTERNS

Pattern	Purpose
Composite	Tree structures
Decorator	Add behavior
Proxy	Control access

WHEN TO USE COMPOSITE PATTERN

Use it when:

✓ You have hierarchical data
✓ You want uniform operations
✓ You want recursive structures
✓ You want to avoid conditionals

BENEFITS

✓ Simplifies code
✓ Eliminates conditionals
✓ Scales naturally
✓ Works well with recursion

ANTI-PATTERNS

❌ Overengineering flat structures
❌ Making everything a tree unnecessarily
❌ Hard debugging with deep trees

REAL-WORLD INSIGHT

Composite is used in:

- File systems
- DOM trees

- UI frameworks
- Organizational structures

Anywhere you see **nested structures**, this pattern is likely present.

MENTAL MODEL

Instead of: “Is this a single object or a group?”

Think: “It doesn’t matter—they behave the same”

KEY TAKEAWAYS

- Composite treats single objects and groups uniformly
- Perfect for tree structures
- Eliminates conditionals
- Uses recursion naturally
- Powerful for scalable hierarchical systems

BEHAVIORAL PATTERNS

OBSERVER PATTERN (EVENTEMITTER DEEP DIVE)

THE CORE IDEA

“Define a one-to-many dependency so that when one object changes state, all its dependents are notified.”

This is the **Observer Pattern**.

In Node.js, this pattern is **built-in** via:

EventEmitter

REAL-WORLD ANALOGY

Think of:

- YouTube subscriptions
- Notifications
- Stock price alerts

One source → many listeners

CORE COMPONENTS

Role	Description
Subject	Emits events
Observer	Listens to events
Event	Message/data sent

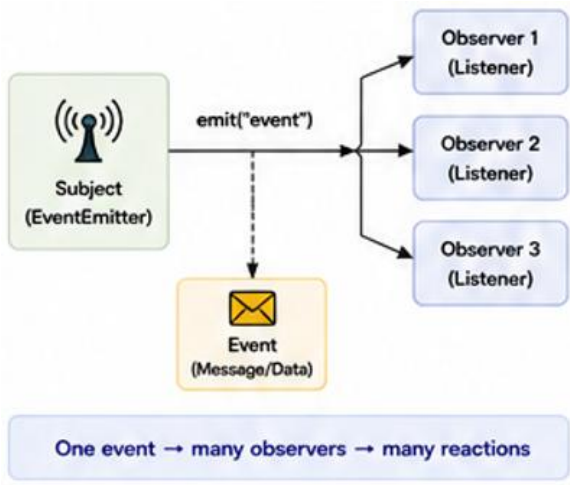
BASIC EVENTEMITTER EXAMPLE

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

const emitter = new EventEmitter();

// Observer
emitter.on('userCreated', (user) => {
  console.log('User created:', user.name);
});

// Trigger event
emitter.emit('userCreated', { name: 'Hernando' });
```

Output:

```
User created: Hernando
```

VISUALIZATION



MULTIPLE OBSERVERS

```
emitter.on('userCreated', sendWelcomeEmail);
emitter.on('userCreated', logUserCreation);
emitter.on('userCreated', updateAnalytics);
```

One event → multiple reactions

✓ Loose coupling
✓ Scalable

REAL NODE.JS EXAMPLE (DECOUPLED SYSTEM)

```
const emitter = new EventEmitter();

function createUser(data) {
  const user = { ...data };
  emitter.emit('userCreated', user);
  return user;
}

// observers
```

```
emitter.on('userCreated', (user) => {
  console.log('Send email to', user.email);
});

emitter.on('userCreated', (user) => {
  console.log('Log user creation');
});
```

Business logic stays clean.

ASYNC OBSERVERS (IMPORTANT)

```
emitter.on('userCreated', async (user) => {
  await sendEmail(user.email);
});
```

But note:

- `emit()` does NOT wait for async listeners
- Execution is not automatically awaited

SAFE PATTERN

```
async function emitAsync(event, data) {
  const listeners = emitter.listeners(event);
  await Promise.all(listeners.map(fn => fn(data)));
}
```

ERROR HANDLING

Special event: `"error"`

```
emitter.on('error', (err) => {
  console.error(err);
});

emitter.emit('error', new Error('Something broke'));
```

⚠️ If no listener:
→ Node.js will crash

ONCE VS ON

```
emitter.once('init', () => {
  console.log('Runs only once');
});
```

Useful for:

- Initialization
- One-time events

REMOVING LISTENERS

```
function handler() {}

emitter.on('event', handler);
emitter.off('event', handler);
```

Prevent memory leaks.

EVENT-DRIVEN ARCHITECTURE

Observer pattern enables:

- Microservices communication
- Background jobs
- Real-time systems

EXAMPLE FLOW

```
User Created → Event → Email + Logging + Analytics
```

OBSERVER VS PUB/SUB

Pattern	Difference
Observer	Direct subscription
Pub/Sub	Uses message broker

Node.js EventEmitter = **Observer (in-process)**

ANTI-PATTERNS

❌ Too many events → chaos
❌ Hidden logic (hard debugging)
❌ Event chains (event triggers event triggers event…)

REAL-WORLD INSIGHT

Event-driven systems are:

Powerful
Scalable
But harder to debug

Senior devs:

- Use events for **decoupling**
- Avoid overusing them for core logic

MENTAL MODEL

Instead of: “Call everything manually”

Think: “Emit an event and let the system react”

KEY TAKEAWAYS

- Observer = one-to-many communication
- EventEmitter is Node.js implementation
- Enables decoupled systems
- Supports scalable architectures
- Must be used carefully to avoid complexity

STRATEGY PATTERN

THE CORE IDEA

"Define a family of algorithms, encapsulate each one, and make them interchangeable."

The **Strategy Pattern** lets you:

- Swap behavior at runtime
- Avoid complex conditionals
- Keep code flexible

In Node.js, this is widely used for:

- Payments
- Validation
- Sorting
- Business rules

THE PROBLEM IT SOLVES

Without Strategy (messy conditionals):

```
function processPayment(method, amount) {
  if (method === 'stripe') {
    // Stripe logic
  } else if (method === 'paypal') {
    // PayPal logic
  } else if (method === 'crypto') {
    // Crypto logic
  }
}
```

Problems:

- Hard to extend
- Violates Open/Closed Principle

- Gets messy fast

THE STRATEGY SOLUTION

Extract each behavior into its own strategy.

EXAMPLE

```
class StripeStrategy {
  pay(amount) {
    console.log('Stripe:', amount);
  }
}

class PayPalStrategy {
  pay(amount) {
    console.log('PayPal:', amount);
  }
}

class PaymentContext {
  constructor(strategy) {
    this.strategy = strategy;
  }

  execute(amount) {
    this.strategy.pay(amount);
  }
}
```

USAGE:

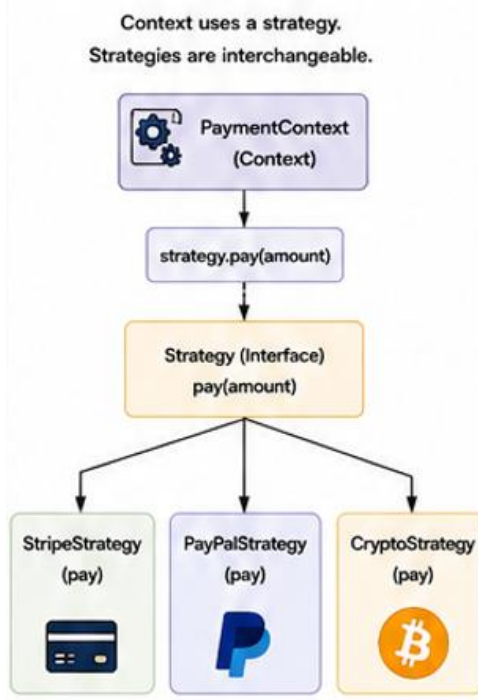
```
const payment = new PaymentContext(new StripeStrategy());
payment.execute(100);
```

Swap strategy easily:

```
payment.strategy = new PayPalStrategy();
```

```
payment.execute(200);
```

VISUALIZATION



FUNCTIONAL STRATEGY

Preferred in modern Node.js:

```
const strategies = {
  stripe: (amount) => console.log('Stripe:', amount),
  paypal: (amount) => console.log('PayPal:', amount),
  crypto: (amount) => console.log('Crypto:', amount)
};

function processPayment(method, amount) {
  return strategies[method](amount);
```

```
}
```

✓ Simple
✓ Extensible
✓ No classes needed

REAL-WORLD EXAMPLE

VALIDATION SYSTEM

```
const validators = {
  email: (val) => val.includes('@'),
  password: (val) => val.length > 6
};

function validate(type, value) {
  return validators[type](value);
}
```

SORTING STRATEGY

```
const sortStrategies = {
  asc: (a, b) => a - b,
  desc: (a, b) => b - a
};

array.sort(sortStrategies.asc);
```

STRATEGY VS FACTORY

Pattern	Purpose
Strategy	Choose behavior at runtime
Factory	Create objects

Strategy = behavior switching
Factory = object creation

WHEN TO USE STRATEGY PATTERN

Use it when:

✓ You have multiple algorithms
✓ You want to switch behavior dynamically
✓ You want to eliminate conditionals
✓ You want scalable logic

BENEFITS

✓ Clean code
✓ Easy to extend
✓ Follows Open/Closed Principle
✓ Improves maintainability

ANTI-PATTERNS

❌ Overengineering simple logic
❌ Too many tiny strategies
❌ Hard-to-track dynamic behavior

REAL-WORLD INSIGHT

Strategy is everywhere in Node.js:

- Payment processors
- Middleware decision logic
- Feature flags
- A/B testing systems

It's one of the most practical patterns.

MENTAL MODEL

Instead of: "Which condition should I use?"

Think: “Which strategy should I plug in?”

KEY TAKEAWAYS

- Strategy encapsulates interchangeable behavior
- Eliminates large conditional blocks
- Enables runtime flexibility
- Works great with functional patterns
- One of the most useful real-world patterns

COMMAND PATTERN

THE CORE IDEA

“Encapsulate a request as an object, allowing you to parameterize, queue, log, or undo operations.”

The **Command Pattern** turns actions into **first-class objects**.

In Node.js, this is powerful for:

- Job queues
- Task scheduling
- Undo/redo systems
- Event-driven workflows

THE PROBLEM IT SOLVES

Direct execution:

```
function createUser(data) {
  // logic here
}
```

Problems:

- Hard to queue
- Hard to log

- Hard to retry
- Hard to undo

THE COMMAND SOLUTION

Wrap the action inside an object:

EXAMPLE

```
class CreateUserCommand {
  constructor(userService, data) {
    this.userService = userService;
    this.data = data;
  }

  execute() {
    return this.userService.create(this.data);
  }
}
```

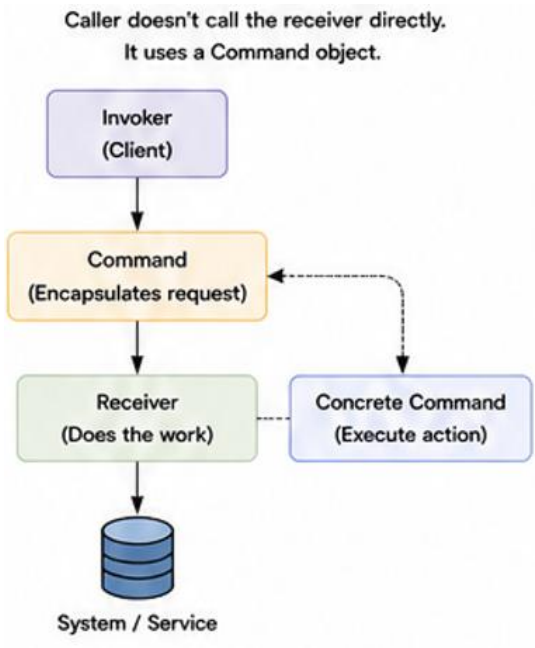
USAGE:

```
const command = new CreateUserCommand(userService, {
  name: 'Hernando'
});

command.execute();
```

Now the action is:

- Encapsulated
- Reusable
- Controllable

VISUALIZATION



FUNCTIONAL COMMAND (NODE.JS STYLE)

Preferred approach:

```
const createUserCommand = (userService, data) => ({
  execute: () => userService.create(data)
});
```

USAGE:

```
const cmd = createUserCommand(userService, { name: 'Hernando' });
cmd.execute();
```

COMMAND QUEUE (REAL POWER)

```
const queue = [];

queue.push(createUserCommand(userService, { name: 'Ana' }));
queue.push(createUserCommand(userService, { name: 'Luis' }));
```

```
queue.forEach(cmd => cmd.execute());
```

Now you can:

- Queue tasks
- Execute later
- Control flow

REAL-WORLD EXAMPLE

JOB PROCESSING SYSTEM

```
function processJobs(jobs) {
  for (const job of jobs) {
    job.execute();
  }
}
```

Used in:

- Background workers
- Task schedulers
- Message queues

UNDO/REDO EXAMPLE

```
class IncrementCommand {
  constructor(counter) {
    this.counter = counter;
  }

  execute() {
    this.counter.value++;
  }

  undo() {
    this.counter.value--;
```

```
    }
}
```

Enables:

- Undo/redo systems
- State management

COMMAND VS STRATEGY

Pattern	Purpose
Command	Encapsulate actions
Strategy	Encapsulate algorithms

Command = action object

Strategy = behavior selection

WHEN TO USE COMMAND PATTERN

Use it when:

✓ You need to queue or schedule tasks

✓ You want undo/redo functionality

✓ You want to log actions

✓ You want decoupled execution

BENEFITS

✓ Decouples sender and receiver

✓ Enables queuing and scheduling

✓ Supports undo/redo

✓ Improves flexibility

ANTI-PATTERNS

❌ Overengineering simple actions
❌ Too many command classes
❌ Hard-to-follow flow if overused

REAL-WORLD INSIGHT

Command pattern is heavily used in:

- Task queues (Bull, RabbitMQ, etc.)
- CLI tools
- Event sourcing systems
- Job schedulers

It's a **core pattern for scalable systems**.

MENTAL MODEL

Instead of: "Call the function"

Think: "Create a command and execute it"

KEY TAKEAWAYS

- Command encapsulates actions as objects
- Enables queuing, logging, undo
- Decouples execution from invocation
- Works great in async systems
- Essential for scalable architectures

ITERATOR PATTERN

THE CORE IDEA

"Provide a way to access elements of a collection **sequentially without exposing its internal structure**."

The **Iterator Pattern** lets you:

- Traverse collections cleanly
- Hide implementation details
- Standardize access

In Node.js, this is deeply integrated into the language via:

- `for...of`
- Iterators
- Generators

THE PROBLEM IT SOLVES

Without an iterator, you might do:

```
for (let i = 0; i < array.length; i++) {
  console.log(array[i]);
}
```

Problems:

- Tightly coupled to structure
- Not reusable
- Doesn't work across different data types

THE ITERATOR SOLUTION

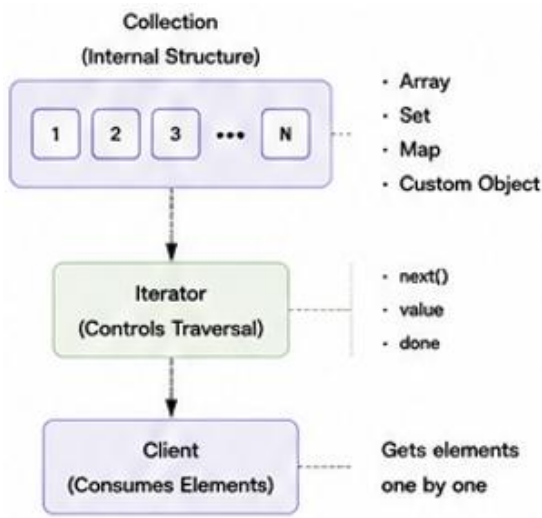
Abstract traversal logic.

BUILT-IN ITERATOR

```
const arr = [1, 2, 3];

for (const item of arr) {
  console.log(item);
}
```

Works because arrays implement the **iterator protocol**.

VISUALIZATION



CUSTOM ITERATOR (MANUAL)

```
const collection = {
  items: [1, 2, 3],
  [Symbol.iterator]() {
    let index = 0;
    const items = this.items;

    return {
      next() {
        if (index < items.length) {
          return { value: items[index++], done: false };
        }
        return { done: true };
      }
    };
  }
```

```
};

for (const item of collection) {
  console.log(item);
}
```

Now your object is iterable.

GENERATORS (MODERN APPROACH)

Much cleaner:

```
function* iterator() {
  yield 1;
  yield 2;
  yield 3;
}

for (const val of iterator()) {
  console.log(val);
}
```

✓ Built-in iterator

✓ Cleaner syntax

REAL-WORLD EXAMPLE

PAGINATION ITERATOR

```
async function* fetchUsers() {
  let page = 1;

  while (true) {
    const users = await getUsers(page);
    if (!users.length) return;

    for (const user of users) {
```

```
      yield user;
    }

    page++;
  }
}
```

USAGE:

```
for await (const user of fetchUsers()) {
  console.log(user);
}
```

Handles large datasets efficiently.

ITERATOR VS LOOP

Approach	Difference
Loop	Manual control
Iterator	Abstracted traversal

WHEN TO USE ITERATOR PATTERN

Use it when:

✓ You want to hide data structure
✓ You need custom traversal
✓ You work with large datasets
✓ You want lazy evaluation

BENEFITS

✓ Cleaner code
✓ Standardized access
✓ Supports lazy loading
✓ Works across structures

ANTI-PATTERNS

❌ Overcomplicating simple loops
❌ Misusing generators for trivial cases
❌ Hard debugging with complex iterators

REAL-WORLD INSIGHT

Iterators power:

- Streams
- Generators
- Async data pipelines
- File processing

They are **fundamental to Node.js internals**.

MENTAL MODEL

Instead of: “How do I loop this?”

Think: “How should this be iterated?”

KEY TAKEAWAYS

- Iterator abstracts traversal logic
- Built into JavaScript (Symbol.iterator)
- Generators simplify implementation
- Supports lazy and async iteration
- Essential for scalable data processing

MEDIATOR PATTERN

THE CORE IDEA

“Define an object that **centralizes communication** between multiple objects.”

Instead of objects talking directly to each other, they communicate through a **mediator**.

In Node.js, this pattern helps:

- Reduce coupling
- Simplify interactions
- Organize complex systems

THE PROBLEM IT SOLVES

Without Mediator:

```
userService.notify(orderService);
orderService.notify(paymentService);
paymentService.notify(userService);
```

Problems:

- Tight coupling
- Hard to maintain
- Complex dependencies

THE MEDIATOR SOLUTION

Centralize communication:

EXAMPLE

```
class Mediator {
  notify(sender, event, data) {
    if (event === 'userCreated') {
      console.log('Notify services about user');
    }
  }
}
```

COMPONENTS:

```
class UserService {
  constructor(mediator) {
    this.mediator = mediator;
```

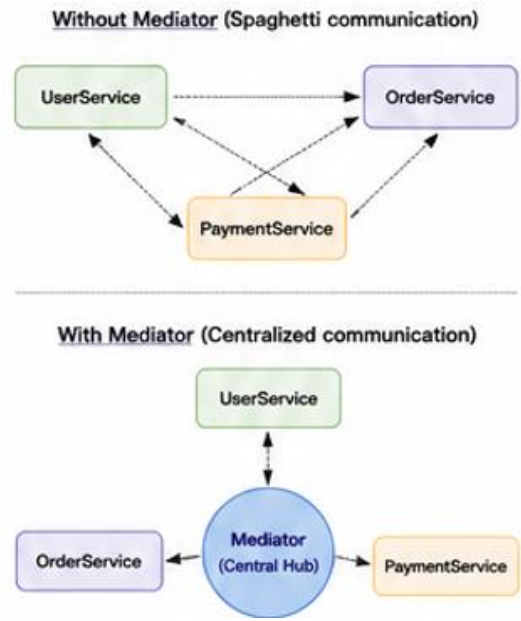
```
  }

  createUser(data) {
    console.log('User created');
    this.mediator.notify(this, 'userCreated', data);
  }
}
```

USAGE:

```
const mediator = new Mediator();
const userService = new UserService(mediator);

userService.createUser({ name: 'Hernando' });
```

Components don't know each other anymore.

VISUALIZATION



FUNCTIONAL MEDIATOR

Preferred approach:

```
const createMediator = () => {
  const events = {};

  return {
    subscribe: (event, handler) => {
      if (!events[event]) events[event] = [];
      events[event].push(handler);
    },
    notify: (event, data) => {
      (events[event] || []).forEach(fn => fn(data));
    }
  };
};
```

USAGE:

```
const mediator = createMediator();

mediator.subscribe('userCreated', (user) => {
  console.log('Send email to', user.name);
});

mediator.notify('userCreated', { name: 'Hernando' });
```

Clean and decoupled.

REAL-WORLD EXAMPLE

CHAT ROOM SYSTEM

```
class ChatRoom {
  send(user, message) {
    console.log(`${user}: ${message}`);
  }
}
```

Users don't talk directly:

- They go through the **chat room (mediator)**

MEDIATOR VS OBSERVER

Pattern	Difference
Mediator	Central controller
Observer	Direct subscriptions

Mediator = **central hub**
Observer = **event broadcast**

WHEN TO USE MEDIATOR PATTERN

Use it when:

✓ Many objects interact
✓ You want to reduce dependencies
✓ Communication logic is complex
✓ You want centralized control

BENEFITS

✓ Loose coupling
✓ Centralized logic
✓ Easier maintenance
✓ Scalable communication

ANTI-PATTERNS

❌ "God Mediator" (too much logic in one place)
❌ Hidden complexity
❌ Overengineering simple interactions

REAL-WORLD INSIGHT

Mediator is used in:

- UI frameworks (component communication)
- Chat systems
- Workflow engines
- Event orchestration systems

It acts as a **communication hub**.

MENTAL MODEL

Instead of: “Objects talk to each other”

Think: “Objects talk through a central hub”

KEY TAKEAWAYS

- Mediator centralizes communication
- Reduces coupling between components
- Simplifies complex interactions
- Must be balanced to avoid over-centralization
- Works great with event-driven systems

STATE PATTERN

THE CORE IDEA

“Allow an object to change its behavior when its internal state changes.”

The **State Pattern** makes an object behave differently depending on its **current state**, without using messy conditionals.

In Node.js, this is extremely useful for:

- Workflows
- Order systems

- Authentication flows
- UI states

THE PROBLEM IT SOLVES

Without State Pattern:

```
function handleOrder(status) {
  if (status === 'pending') {
    console.log('Processing order...');
  } else if (status === 'shipped') {
    console.log('Order shipped');
  } else if (status === 'delivered') {
    console.log('Order delivered');
  }
}
```

Problems:

- Hard to extend
- Violates Open/Closed Principle
- Gets messy fast

THE STATE SOLUTION

Encapsulate each state into its own object.

EXAMPLE

```
class PendingState {
  handle() {
    console.log('Processing order...');
  }
}

class ShippedState {
```

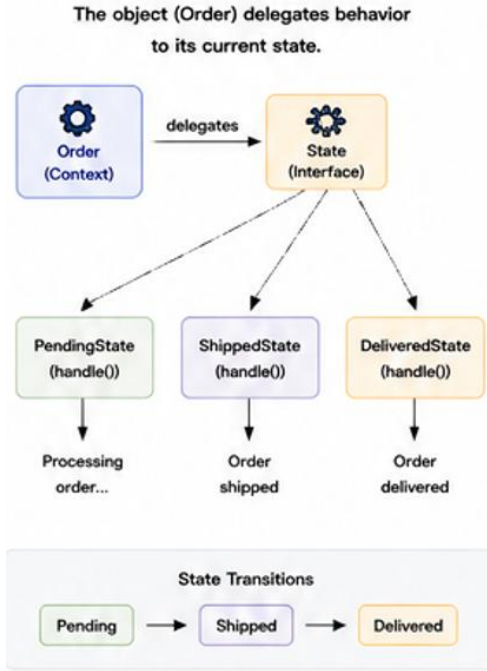
```
  handle() {
    console.log('Order shipped');
  }
}

class DeliveredState {
  handle() {
    console.log('Order delivered');
  }
}

class Order {
  constructor(state) {
    this.state = state;
  }

  setState(state) {
    this.state = state;
  }

  process() {
    this.state.handle();
  }
}
```

USAGE:

```
const order = new Order(new PendingState());
order.process();

order.setState(new ShippedState());
order.process();
```

Behavior changes dynamically.

VISUALIZATION



FUNCTIONAL STATE (NODE.JS STYLE)

Preferred approach:

```
const states = {
  pending: () => console.log('Processing order...'),
  shipped: () => console.log('Order shipped'),
  delivered: () => console.log('Order delivered')
};

function handleOrder(state) {
  return states[state]();
}
```

REAL-WORLD EXAMPLE

AUTHENTICATION FLOW

```
const authStates = {
  loggedOut: () => console.log('Please log in'),
  loggedIn: () => console.log('Welcome user'),
  banned: () => console.log('Access denied')
};
```

USAGE:

```
authStates.loggedOut();
authStates.loggedIn();
```

FINITE STATE MACHINE (ADVANCED)

State pattern is the foundation of:

- Finite State Machines (FSM)
- Workflow engines

EXAMPLE:

```
Pending → Shipped → Delivered
```

Each transition is controlled.

STATE VS STRATEGY

Pattern	Purpose
State	Behavior changes with state
Strategy	Behavior chosen externally

State = internal change

Strategy = external choice

WHEN TO USE STATE PATTERN

Use it when:

✓ Behavior depends on state
✓ You have many conditional branches
✓ You need controlled transitions
✓ You want scalable workflows

BENEFITS

✓ Eliminates conditionals
✓ Improves readability
✓ Makes transitions explicit
✓ Scales well

ANTI-PATTERNS

❌ Overengineering simple logic
❌ Too many small state objects
❌ Hard-to-track transitions if poorly designed

REAL-WORLD INSIGHT

State pattern is used in:

- Order systems
- Game engines
- UI frameworks
- Workflow automation

It's critical for **complex business logic**.

MENTAL MODEL

Instead of: "Check conditions"

Think: "What state am I in?"

KEY TAKEAWAYS

- State encapsulates behavior based on state
- Eliminates complex conditionals
- Enables clear workflows
- Foundation for FSMs
- Very useful in real-world systems

CHAIN OF RESPONSIBILITY

THE CORE IDEA

"Pass a request along a chain of handlers until one of them handles it."

The **Chain of Responsibility** lets multiple objects:

- Handle a request
- Or pass it to the next handler

In Node.js, this pattern is everywhere—especially in:

- Middleware systems
- Request pipelines
- Validation chains

THE PROBLEM IT SOLVES

Without Chain:

```
function handleRequest(req) {
  if (!req.auth) return 'Unauthorized';
  if (!req.valid) return 'Invalid';
  if (!req.data) return 'Missing data';

  return 'Success';
}
```

Problems:

- Hard to extend
- Rigid logic
- Violates Open/Closed Principle

THE CHAIN SOLUTION

Break logic into handlers:

EXAMPLE

```
class Handler {
  setNext(handler) {
    this.next = handler;
    return handler;
  }

  handle(req) {
    if (this.next) {
      return this.next.handle(req);
    }
  }
}

class AuthHandler extends Handler {
  handle(req) {
    if (!req.auth) return 'Unauthorized';
    return super.handle(req);
  }
}

class ValidationHandler extends Handler {
  handle(req) {
    if (!req.valid) return 'Invalid';
    return super.handle(req);
  }
}
```

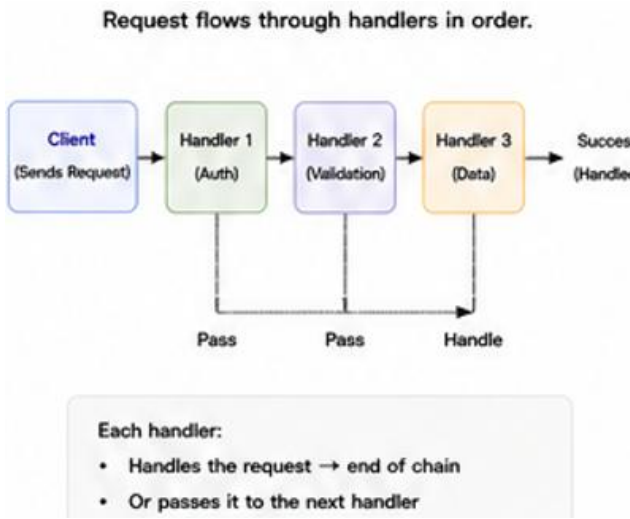
```
class DataHandler extends Handler {
  handle(req) {
    if (!req.data) return 'Missing data';
    return 'Success';
  }
}
```

USAGE:

```
const auth = new AuthHandler();
const validation = new ValidationHandler();
const data = new DataHandler();

auth.setNext(validation).setNext(data);

console.log(auth.handle({ auth: true, valid: true, data: true }));
```

Request flows through the chain.

VISUALIZATION



FUNCTIONAL CHAIN

Preferred approach:

```
const auth = (req, next) => {
  if (!req.auth) return 'Unauthorized';
  return next(req);
};

const validate = (req, next) => {
  if (!req.valid) return 'Invalid';
  return next(req);
};

const process = (req) => 'Success';

const chain = (req) => auth(req, (r) =>
  validate(r, process)
);

console.log(chain({ auth: true, valid: true }));
```

REAL NODE.JS EXAMPLE

```
app.use(authMiddleware);
app.use(validationMiddleware);

app.get('/data', handler);
```

Request flow:

```
Auth → Validation → Handler
```

✓ Classic Chain of Responsibility

REAL-WORLD USE CASES

- HTTP request pipelines
- Middleware systems
- Validation pipelines
- Logging chains
- Error handling

CHAIN VS DECORATOR

Pattern	Purpose
Chain	Pass request along handlers
Decorator	Wrap and enhance behavior

Chain = sequential processing
Decorator = layered enhancement

WHEN TO USE CHAIN PATTERN

Use it when:

✓ Multiple handlers may process a request
✓ You want flexible pipelines
✓ You want to avoid large conditionals
✓ You want scalable processing

BENEFITS

✓ Flexible and extensible
✓ Decouples handlers
✓ Easy to add/remove steps
✓ Clean architecture

ANTI-PATTERNS

❌ Too many handlers (hard debugging)
❌ Hidden flow (unclear execution path)
❌ Handlers that don't follow contract

REAL-WORLD INSIGHT

Most backend systems are basically:

Chains of responsibility

Examples:

- Express middleware
- API gateways
- Request validation systems

MENTAL MODEL

Instead of: “Handle everything in one place”

Think: “Pass the request through a pipeline”

KEY TAKEAWAYS

- Chain passes requests through handlers
- Each handler decides to process or pass
- Eliminates large conditional blocks
- Core pattern in middleware systems
- Enables scalable request pipelines

TEMPLATE METHOD PATTERN

THE CORE IDEA

“Define the **skeleton of an algorithm** in a method, but let subclasses or functions override specific steps.”

The **Template Method Pattern** lets you:

- Keep a fixed workflow
- Customize individual steps

In Node.js, this is common in:

- Frameworks
- Request pipelines
- Data processing flows

THE PROBLEM IT SOLVES

Without Template Method:

```
function processOrder(order) {
  validate(order);
  charge(order);
  ship(order);
}
```

If behavior changes:

- You duplicate logic
- Hard to reuse
- No structure enforcement

THE TEMPLATE SOLUTION

Define the algorithm once, allow steps to vary.

EXAMPLE

```
class OrderProcessor {
  process(order) {
    this.validate(order);
    this.pay(order);
    this.ship(order);
  }

  validate(order) {
    throw new Error('Must implement validate');
  }
```

```
  pay(order) {
    throw new Error('Must implement pay');
  }

  ship(order) {
    console.log('Shipping order');
  }
}
```

CONCRETE IMPLEMENTATION:

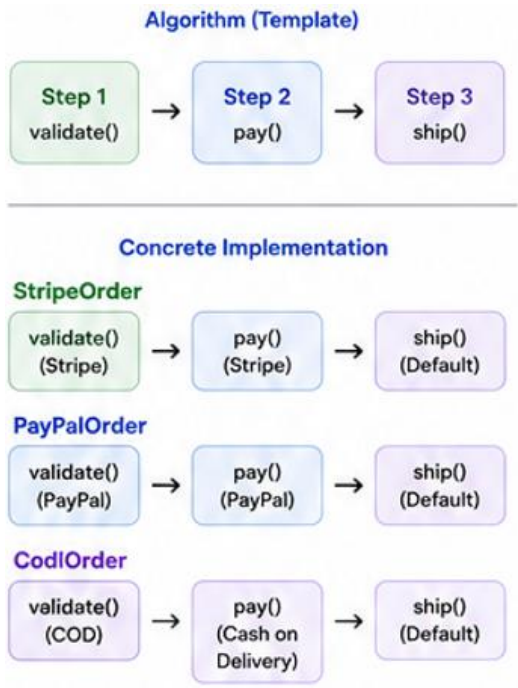
```
class StripeOrder extends OrderProcessor {
  validate(order) {
    console.log('Validating order');
  }

  pay(order) {
    console.log('Paying with Stripe');
  }
}
```

USAGE:

```
const order = new StripeOrder();
order.process({});
```

The structure stays the same, behavior changes.

VISUALIZATION



FUNCTIONAL TEMPLATE

Preferred approach:

```
const processOrder = ({ validate, pay, ship }) => (order) => {
  validate(order);
  pay(order);
  ship(order);
};
```

USAGE:

```
const stripeOrder = processOrder({
  validate: () => console.log('Validate'),
  pay: () => console.log('Stripe payment'),
  ship: () => console.log('Ship order')
});
```

```
stripeOrder({});
```

✓ No classes
✓ Highly flexible

REAL-WORLD EXAMPLE

API REQUEST PIPELINE

```
const handleRequest = ({ auth, validate, handler }) => async (req) =>
{
  await auth(req);
  await validate(req);
  return handler(req);
};
```

Fixed flow:

```
Auth → Validate → Handle
```

TEMPLATE VS STRATEGY

Pattern	Purpose
Template Method	Fixed structure, variable steps
Strategy	Swap entire behavior

Template = structure fixed
Strategy = behavior swapped

WHEN TO USE TEMPLATE METHOD

Use it when:

✓ You have a fixed workflow
✓ Some steps vary
✓ You want consistency
✓ You want to enforce structure

BENEFITS

✓ Enforces architecture
✓ Reduces duplication
✓ Improves consistency
✓ Flexible customization

ANTI-PATTERNS

❌ Too rigid structure
❌ Overengineering simple flows
❌ Deep inheritance chains (in OOP version)

REAL-WORLD INSIGHT

Template Method is used in:

- Frameworks (Express-like pipelines)
- SDKs
- Data processing pipelines
- CI/CD workflows

It's the backbone of **structured systems**.

MENTAL MODEL

Instead of: "Rewrite the whole process"

Think: "Keep the structure, change the steps"

KEY TAKEAWAYS

- Template defines algorithm structure
- Steps can be overridden
- Ensures consistency across implementations
- Works great with functional composition

- Useful in pipelines and workflows

NODE.JS SPECIFIC PATTERNS

MIDDLEWARE PATTERN

THE CORE IDEA

"Process a request through a **chain of functions**, where each function can modify the request, response, or pass control forward."

The **Middleware Pattern** is one of the most important patterns in Node.js.

It combines ideas from:

- Chain of Responsibility
- Decorator
- Functional composition

WHAT IS MIDDLEWARE?

A middleware is simply:

```
(req, res, next) => { ... }
```

Where:

- `req` → request
- `res` → response
- `next()` → passes control to the next middleware

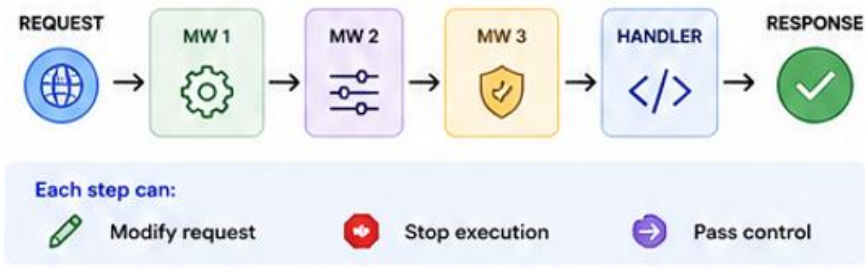
BASIC EXAMPLE

Using Express:

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

This runs for every request.

VISUALIZATION



MIDDLEWARE CHAIN FLOW

```
Request → Middleware 1 → Middleware 2 → Middleware 3 → Handler → Response
```

Each step can:

- Modify request
- Stop execution
- Pass control

MULTIPLE MIDDLEWARE

```
app.use(authMiddleware);
app.use(loggingMiddleware);

app.get('/data', (req, res) => {
  res.send('OK');
});
```

Flow: Auth → Logging → Handler

CUSTOM MIDDLEWARE EXAMPLES

AUTHENTICATION

```
const authMiddleware = (req, res, next) => {
  if (!req.headers.authorization) {
    return res.status(401).send('Unauthorized');
  }
  next();
};
```

LOGGING

```
const loggerMiddleware = (req, res, next) => {
  console.log(req.method, req.url);
  next();
};
```

ERROR HANDLING

```
const errorMiddleware = (err, req, res, next) => {
  console.error(err);
  res.status(500).send('Server Error');
};
```

Special signature (4 params).

WRITING YOUR OWN MIDDLEWARE ENGINE

Core idea:

```
function runMiddleware(req, res, middlewares) {
  let index = 0;

  function next() {
    const middleware = middlewares[index++];
    if (middleware) {
      middleware(req, res, next);
    }
  }

  next();
```

```
}
```

USAGE:

```
runMiddleware(req, res, [
  loggerMiddleware,
  authMiddleware,
  handler
]);
```

This is basically how Express works internally.

ASYNC MIDDLEWARE (IMPORTANT)

```
const asyncMiddleware = async (req, res, next) => {
  try {
    await someAsyncTask();
    next();
  } catch (err) {
    next(err);
  }
};
```

Always handle errors properly.

MIDDLEWARE VS OTHER PATTERNS

Pattern	Role
Middleware	Request pipeline
Chain of Responsibility	Generalized version
Decorator	Adds behavior
Proxy	Controls access

Middleware = **practical implementation** of multiple patterns

REAL-WORLD USE CASES

- Authentication
- Logging
- Validation

- Rate limiting
- CORS handling
- Request transformation

BEST PRACTICES

✓ Keep middleware small and focused
✓ Always call `next()` (or end response)
✓ Handle errors properly
✓ Avoid heavy logic inside middleware

ANTI-PATTERNS

❌ Forgetting `next()` → request hangs
❌ Too many middlewares → performance issues
❌ Mixing responsibilities
❌ Hidden side effects

REAL-WORLD INSIGHT

Modern backends are basically: **Middleware pipelines**

Examples:

- Express
- Koa
- Fastify
- Next.js API routes

MENTAL MODEL

Instead of: “Handle everything in one function”

Think: “Pass the request through a pipeline of transformations”

KEY TAKEAWAYS

- Middleware = chain of functions processing requests
- Core to Node.js frameworks
- Combines multiple design patterns
- Enables scalable and modular systems
- Foundation of modern backend architecture

ERROR-FIRST CALLBACK PATTERN

THE CORE IDEA

"Callbacks receive an **error as the first argument**, and the result as the second."

This is one of the oldest and most fundamental conventions in Node.js.

THE STANDARD SIGNATURE

```
function callback(error, result) { }
```

Rules:

- If **error exists** → handle it
- If **error is null** → use result

BASIC EXAMPLE

```
function fetchUser(id, callback) {
  if (!id) {
    return callback(new Error('Invalid ID'), null);
  }

  const user = { id, name: 'Hernando' };
  callback(null, user);
}
```

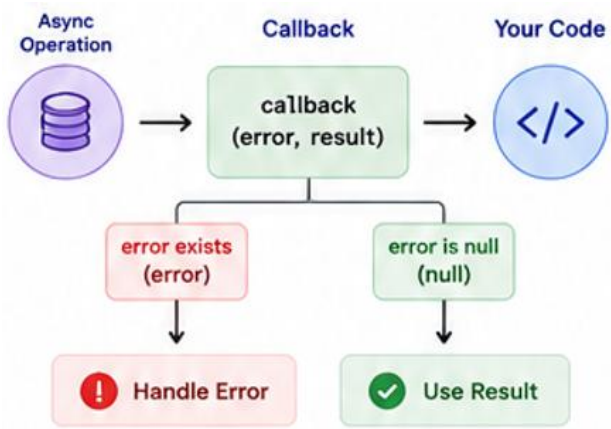
USAGE:

```
fetchUser(1, (err, user) => {
```

```
  if (err) {
    console.error(err.message);
    return;
  }

  console.log(user);
});
```

This pattern forces **explicit error handling**.

VISUALIZATION



WHY THIS PATTERN EXISTS

Before Promises and async/await:

Node.js needed a consistent way to handle async errors.

✓ Simple
✓ Predictable
✓ Standardized across APIs

REAL NODE.JS

Using built-in modules:

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

fs.readFile('file.txt', (err, data) => {
  if (err) {
    return console.error('Error reading file');
  }

  console.log(data.toString());
});
```

Every core API follows this pattern.

CALLBACK HELL (THE PROBLEM)

Nested callbacks:

```
getUser(id, (err, user) => {
  if (err) return;

  getOrders(user.id, (err, orders) => {
    if (err) return;

    getPayment(orders, (err, payment) => {
      if (err) return;

      console.log(payment);
    });
  });
});
```

Known as: “Callback Hell” / “Pyramid of Doom”

BEST PRACTICES

✓ Always check error first
✓ Return early on error
✓ Keep callbacks shallow
✓ Use named functions

CLEAN VERSION:

```
function handleUser(err, user) {
  if (err) return console.error(err);

  getOrders(user.id, handleOrders);
}
```

TRANSITION TO PROMISES

Modern Node.js replaces this with:

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

const data = await fs.readFile('file.txt');
```

✓ Cleaner
✓ Easier error handling

PROMISIFYING CALLBACKS

```
const util = require('util');
const readFile = util.promisify(fs.readFile);
```

Converts callback → Promise.

WHEN TO USE IT TODAY

Use Error-First Callbacks when:

✓ Working with legacy code
✓ Using low-level Node.js APIs
✓ Writing libraries that follow Node conventions

WHEN NOT TO USE IT

❌ New application code
❌ Complex async flows
❌ Modern frameworks

Prefer:

- Promises
- async/await

ERROR HANDLING STRATEGY

```
if (err) return callback(err);
```

Always:

- Stop execution
- Avoid silent failures

REAL-WORLD INSIGHT

Even today:

Many core systems still rely on this pattern internally.

Understanding it helps you:

- Debug legacy systems
- Understand Node internals
- Write better async code

MENTAL MODEL

Instead of: “Return value directly”

Think: “Return via callback, handle error first”

KEY TAKEAWAYS

- Error-first callbacks are a Node.js convention
- `(err, result)` is the standard signature
- Always handle error first
- Leads to callback hell if misused
- Mostly replaced by Promises and async/await

REACTOR PATTERN

THE CORE IDEA

“Handle multiple I/O operations using a **single thread** by reacting to events as they occur.”

The **Reactor Pattern** is the **foundation of how Node.js works internally**.

THE BIG INSIGHT

Node.js is NOT magic.

It’s basically:

```
Event Loop + Callback Queue + Non-blocking I/O
```

This is the **Reactor Pattern in action**.

THE PROBLEM IT SOLVES

Traditional blocking systems:

```
const data = fs.readFileSync('file.txt');
console.log(data);
```

Problems:

- Blocks the thread
- Cannot handle many requests efficiently

THE REACTOR SOLUTION

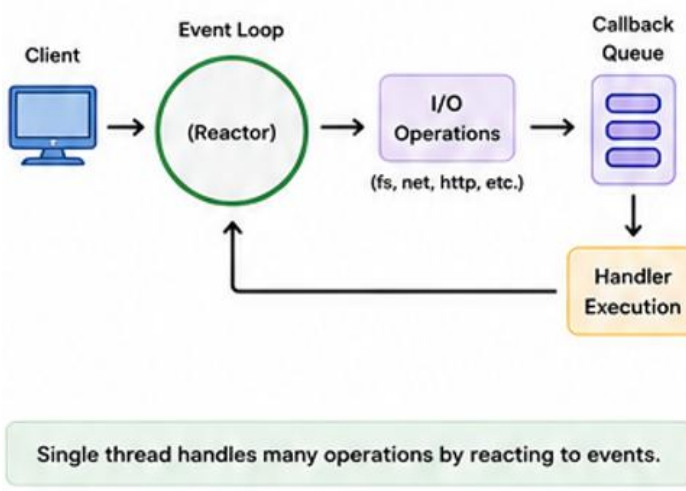
Don't wait → **react when ready**

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

While waiting:

- Node.js handles other tasks

VISUALIZATION



CORE COMPONENTS OF REACTOR PATTERN

1. EVENT DEMULTIPLEXER

Listens for events (I/O readiness)

In Node.js:

- Handled by **libuv**

2. EVENT LOOP

Continuously checks for events and dispatches them

3. HANDLERS (CALLBACKS)

Functions executed when events occur

FLOW:

```
Request → Event Loop → I/O → Callback Queue → Handler Execution
```

EXAMPLE FLOW

```
console.log('Start');

setTimeout(() => {
  console.log('Async Task');
}, 0);

console.log('End');
```

OUTPUT:

```
Start
End
Async Task
```

Because:

- Event loop defers async tasks