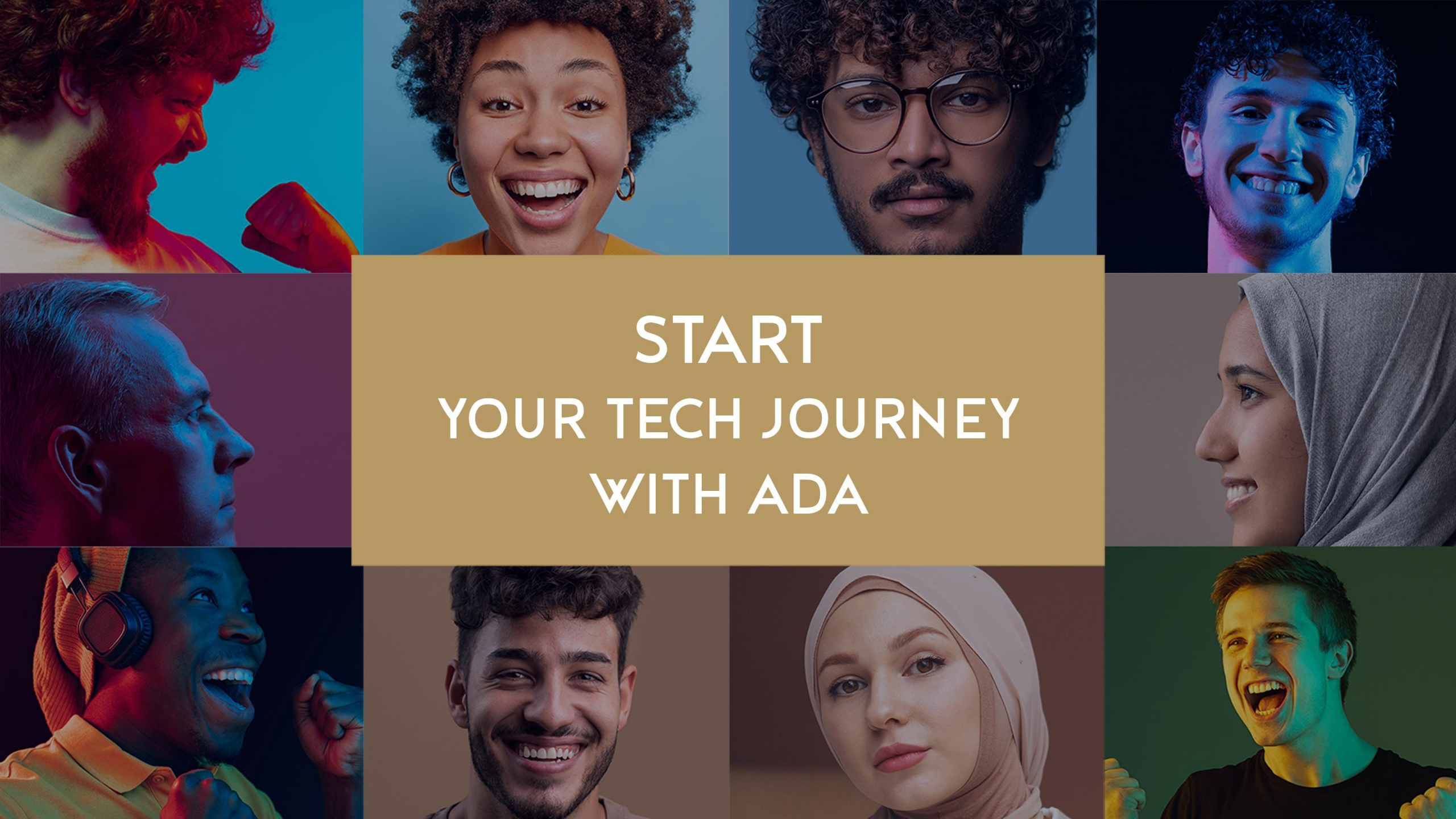

ACADEMY
OF DIGITAL ARTS
EGYPT
Google
cisco
Adobe
IBM
Microsoft
CompTIA


START
YOUR TECH JOURNEY
WITH ADA

Agenda

- What is Node.js? - Runtime environment and architecture
- Node.js runtime (V8 engine) - How JavaScript runs on the server
- Single-threaded, event-driven, non-blocking I/O concepts
- Event loop concept - The heart of Node.js performance
- CommonJS Modules - require() and module.exports
- Splitting code into multiple files
- Built-in modules (fs, path, os)
- Synchronous vs. Asynchronous execution in Node.js
- Callback functions in Node.js context

What is Node.js?

Definition:
Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine that allows you to run JavaScript on the server side.

Key Characteristics:

- JavaScript everywhere - Same language for frontend and backend
- Built on V8 engine - Fast JavaScript execution
- Event-driven - Responds to events and callbacks
- Non-blocking I/O - Doesn't wait for slow operations
- Single-threaded - One main thread with event loop
- Cross-platform - Runs on Windows, macOS, Linux

What is Node.js?

Node.js vs Browser JavaScript:

Browser JavaScript	Node.js
DOM manipulation	File system access
Window object	Global object
Limited file access	Full file system
Browser APIs	Node.js APIs
Client-side	Server-side
Security sandbox	Full system access

What is Node.js?

What Node.js is NOT:

❌ Not a programming language (JavaScript is)
❌ Not a framework (Express.js is a framework)
❌ Not a library
❌ Not multi-threaded like traditional servers

What Node.js IS:

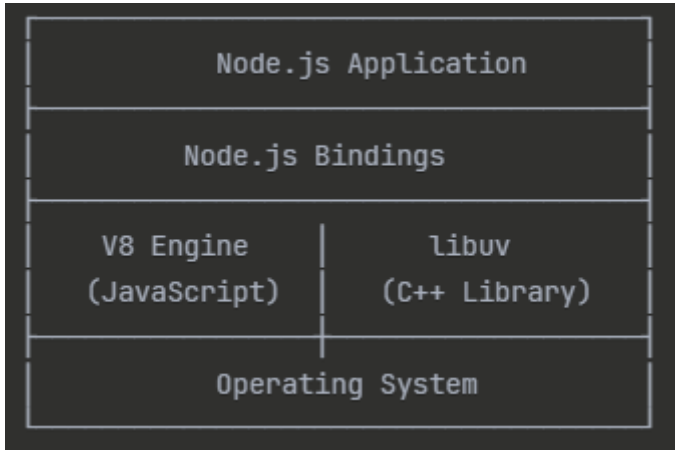
✅ JavaScript runtime environment
✅ Platform for building server applications
✅ Event-driven, non-blocking I/O model
✅ Built for scalable network applications

Node.js Runtime Architecture

V8 JavaScript Engine:

- Google's V8 engine - Same engine that powers Chrome
- Compiles JavaScript to machine code - Very fast execution
- Memory management - Automatic garbage collection
- Optimizations - Just-in-time compilation

Node.js Runtime Components:



Node.js Runtime Architecture

Core Components:

- V8 Engine - Executes JavaScript code
- libuv - C++ library for async I/O operations
- Node.js Bindings - Bridge between JavaScript and C++
- Node.js Standard Library - Built-in modules (fs, http, etc.)

How it works:

```
// Your JavaScript code
console.log("Hello, Node.js!");

// ↓ Processed by V8 Engine
// ↓ Compiled to machine code
// ↓ Executed by the system
// ↓ Output: Hello, Node.js!
```

Single-Threaded Event-Driven Architecture

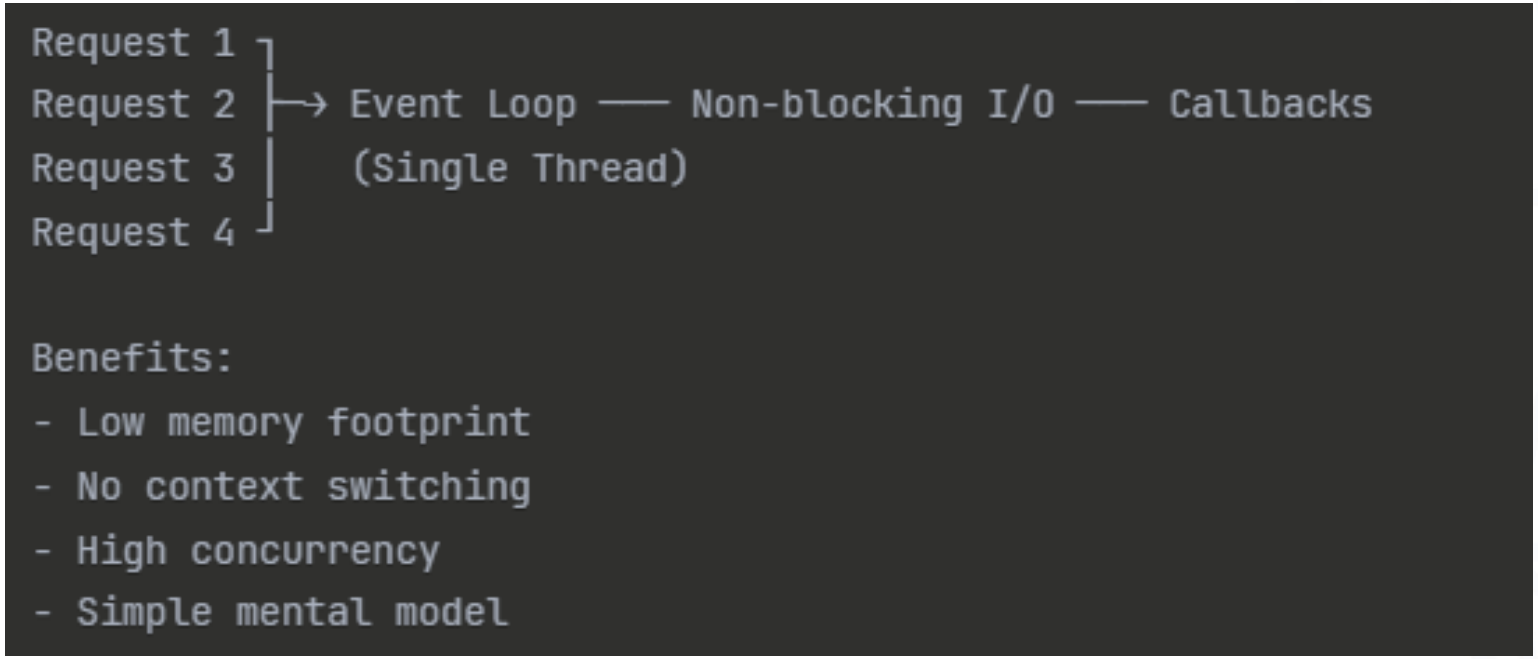
Traditional Multi-Threaded Model (PHP, Java):

```
Request 1 → Thread 1 ── Database ── Response 1
Request 2 → Thread 2 ── Database ── Response 2
Request 3 → Thread 3 ── Database ── Response 3
Request 4 → Thread 4 ── Database ── Response 4

Problems:
- Memory overhead (each thread ~2MB)
- Context switching overhead
- Thread management complexity
- Limited scalability
```

Single-Threaded Event-Driven Architecture

Node.js Single-Threaded Model:



Single-Threaded Event-Driven Architecture

Single-Threaded Characteristics:

```
// This is single-threaded
console.log("Start");

setTimeout(() => {
    console.log("Timeout callback");
}, 0);

console.log("End");

// Output:
// Start
// End
// Timeout callback
```

Single-Threaded Event-Driven Architecture

Event-Driven Programming:

```
// Events trigger callbacks
const EventEmitter = require('events');
const emitter = new EventEmitter();

// Register event listener
emitter.on('message', (data) => {
    console.log('Received:', data);
});

// Emit event
emitter.emit('message', 'Hello World!');
```

Single-Threaded Event-Driven Architecture

When Single-Threaded Works Well:

✅ I/O intensive applications (APIs, web servers)
✅ Real-time applications (chat, gaming)
✅ Data streaming applications
✅ Microservices architectures

When Single-Threaded Struggles:

❌ CPU intensive calculations
❌ Heavy computational tasks
❌ Blocking operations in main thread

Non-Blocking I/O

Blocking I/O (Synchronous):

```
// This blocks the entire thread
const fs = require('fs');

console.log("Before file read");

// This BLOCKS until file is read completely
const data = fs.readFileSync('large-file.txt', 'utf8');
console.log("File contents:", data.length);

console.log("After file read");

// Timeline:
// 1. "Before file read"
// 2. [WAITING... 2 seconds]
// 3. "File contents: 1000000"
// 4. "After file read"
```

Non-Blocking I/O

Non-Blocking I/O (Asynchronous):

```
// This doesn't block the thread
const fs = require('fs');

console.log("Before file read");

// This returns immediately, callback runs later
fs.readFile('large-file.txt', 'utf8', (err, data) => {
    if (err) {
        console.error("Error:", err);
    } else {
        console.log("File contents:", data.length);
    }
});

console.log("After file read");

// Timeline:
// 1. "Before file read"
// 2. "After file read"        ← Immediate!
// 3. [Later] "File contents: 1000000"
```

Non-Blocking I/O

I/O Operations in Node.js:

```
// All these are non-blocking by default:
fs.readFile('file.txt', callback);           // File operations
http.request('http://api.com', callback);    // Network requests
db.query('SELECT * FROM users', callback);   // Database queries
setTimeout(callback, 1000);                  // Timers
```

The Event Loop

What is the Event Loop?

The Event Loop is the mechanism that allows Node.js to perform non-blocking I/O operations despite being single-threaded.

Event Loop Phases:

The Event Loop

Simple Event Loop Example:

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

// Queued for timers phase
setTimeout(() => console.log("Timeout"), 0);

// Queued for check phase
setImmediate(() => console.log("Immediate"));

// Queued for next tick (highest priority)
process.nextTick(() => console.log("Next Tick"));

console.log("End");

// Output:
// Start
// End
// Next Tick
// Immediate
// Timeout
```

The Event Loop

Event Loop Rules:

- Process.nextTick() - Highest priority, runs before any phase
- Promises - High priority, runs after nextTick
- setImmediate() - Runs in check phase
- setTimeout() - Runs in timers phase
- I/O callbacks - Run when I/O operations complete

CommonJS Modules

Module System Evolution:

```
// Before modules - Everything in global scope
var userName = "Alice";
function greetUser() { ... }
var config = { ... };

// Problems:
// - Name collisions
// - Hard to maintain
// - No dependency management
// - Global pollution
```

CommonJS Solution:

```
// Each file is a module with its own scope
// math.js
const PI = 3.14159;

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

function subtract(a, b) {
    return a - b;
}

// Only export what you want to share
module.exports = {
    add,
    subtract,
    PI
};
```

CommonJS Modules

Different Export Patterns:

```
// Pattern 1: Export object
module.exports = {
    add: (a, b) => a + b,
    subtract: (a, b) => a - b
};
```

```
// Pattern 2: Export function
module.exports = function Calculator() {
    return {
        add: (a, b) => a + b,
        subtract: (a, b) => a - b
    };
};
```

```
// Pattern 3: Export class
module.exports = class Calculator {
    add(a, b) { return a + b; }
    subtract(a, b) { return a - b; }
};
```

```
// Pattern 4: Add to exports gradually
exports.add = (a, b) => a + b;
exports.subtract = (a, b) => a - b;
// Note: Don't reassign exports = {...} - it breaks the reference!
```

Built-in Node.js Modules

File System (fs) Module:

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

```
// Read file asynchronously
fs.readFile('data.txt', 'utf8', (err, data) => {
    if (err) {
        console.error('Error reading file:', err);
        return;
    }
    console.log('File contents:', data);
});
```

```
// Write file asynchronously
const content = "Hello, Node.js!";
fs.writeFile('output.txt', content, (err) => {
    if (err) {
        console.error('Error writing file:', err);
        return;
    }
    console.log('File written successfully');
});
```

```
// Check if file exists
fs.access('somefile.txt', fs.constants.F_OK, (err) => {
    if (err) {
        console.log('File does not exist');
    } else {
        console.log('File exists');
    }
});
```

```
// Get file stats
fs.stat('package.json', (err, stats) => {
    if (err) {
        console.error('Error getting file stats:', err);
        return;
    }

    console.log('File size:', stats.size);
    console.log('Is file:', stats.isFile());
    console.log('Is directory:', stats.isDirectory());
    console.log('Modified:', stats.mtime);
});
```

Built-in Node.js Modules

Path Module:

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

```
// Current file information
console.log('Current file:', __filename);
console.log('Current directory:', __dirname);
```

```
// Join paths (cross-platform)
const fullPath = path.join(__dirname, 'data', 'users.json');
console.log('Full path:', fullPath);
```

```
// Path operations
const filePath = '/users/alice/documents/report.pdf';

console.log('Directory:', path.dirname(filePath));    // /users/alice/documents
console.log('Filename:', path.basename(filePath));    // report.pdf
console.log('Extension:', path.extname(filePath));    // .pdf
console.log('Name only:', path.basename(filePath, '.pdf')); // report
```

```
// Resolve paths (absolute)
const absolutePath = path.resolve('data', 'config.json');
console.log('Absolute path:', absolutePath);
```

```
// Parse path into components
const pathInfo = path.parse('/users/alice/documents/report.pdf');
console.log('Path info:', pathInfo);
```

Synchronous vs Asynchronous in Node.js

Synchronous vs Asynchronous in Node.js

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

console.log("1: Starting");

try {
    // This BLOCKS until file is read
    const data = fs.readFileSync('large-file.txt', 'utf8');
    console.log("2: File read complete, size:", data.length);
} catch (error) {
    console.error("2: Error reading file:", error.message);
}

console.log("3: Continuing...");

// Timeline:
// 1: Starting
// [WAIT for file read...]
// 2: File read complete, size: 1000000
// 3: Continuing...
```

Synchronous vs Asynchronous in Node.js

Asynchronous Operations (Non-Blocking):

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

console.log("1: Starting");

// This returns immediately, callback runs later
fs.readFile('large-file.txt', 'utf8', (error, data) => {
    if (error) {
        console.error("File read error:", error.message);
    } else {
        console.log("File read complete, size:", data.length);
    }
});

console.log("2: Continuing...");

// More operations can happen here
setTimeout(() => {
    console.log("3: Timer callback");
}, 100);

console.log("4: End of main thread");

// Timeline:
// 1: Starting
// 2: Continuing...
// 4: End of main thread
// 3: Timer callback
// File read complete, size: 1000000
```

Synchronous vs Asynchronous in Node.js

When to Use Synchronous:

```
// ✅ GOOD: Application startup/initialization
const config = JSON.parse(fs.readFileSync('config.json', 'utf8'));

// ✅ GOOD: Simple scripts/utilities
const packageInfo = JSON.parse(fs.readFileSync('package.json', 'utf8'));
console.log(`Starting ${packageInfo.name} v${packageInfo.version}`);

// ✅ GOOD: Error conditions where you need to stop
if (!fs.existsSync('required-file.txt')) {
    console.error("Required file missing. Exiting.");
    process.exit(1);
}
```

Synchronous vs Asynchronous in Node.js

When to Use Asynchronous:

```
// ✅ GOOD: Web server request handling
app.get('/users/:id', async (req, res) => {
    // Non-blocking database query
    const user = await db.findUser(req.params.id);
    res.json(user);
});

// ✅ GOOD: Processing multiple files
const files = ['file1.txt', 'file2.txt', 'file3.txt'];

files.forEach(filename => {
    fs.readFile(filename, 'utf8', (err, data) => {
        if (err) {
            console.error(`Error reading ${filename}:`, err);
        } else {
            console.log(`Processed ${filename}: ${data.length} bytes`);
        }
    });
});

// ✅ GOOD: Any operation that might take time
function processLargeDataset(data, callback) {
    // Use setImmediate to avoid blocking
    setImmediate(() => {
        const result = data.map(item => heavyProcessing(item));
        callback(null, result);
    });
}
```

Synchronous vs Asynchronous in Node.js

Performance Comparison:

```
// BAD: Synchronous file reading in server
app.get('/data', (req, res) => {
    // This blocks ALL other requests!
    const data = fs.readFileSync('large-data.json', 'utf8');
    res.json(JSON.parse(data));
});

// GOOD: Asynchronous file reading in server
app.get('/data', (req, res) => {
    // This doesn't block other requests
    fs.readFile('large-data.json', 'utf8', (err, data) => {
        if (err) {
            res.status(500).json({error: 'Failed to read data'});
        } else {
            res.json(JSON.parse(data));
        }
    });
});
```

Callback Functions in Node.js

Standard Node.js Callback Pattern:

```
// Error-first callback pattern
function readUserData(userId, callback) {
    // First parameter: error (null if no error)
    // Second parameter: result data

    if (userId <= 0) {
        // Call callback with error
        callback(new Error("Invalid user ID"), null);
        return;
    }

    // Simulate async operation
    setTimeout(() => {
        const userData = {
            id: userId,
            name: "Alice",
            email: "alice@example.com"
        };

        // Call callback with success (no error)
        callback(null, userData);
    }, 1000);
}

// Usage
readUserData(123, (error, user) => {
    if (error) {
        console.error("Error:", error.message);
    } else {
        console.log("User:", user);
    }
});
```

Callback Functions in Node.js

File System Callbacks:

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

// Reading file with callback
fs.readFile('users.json', 'utf8', (err, data) => {
    if (err) {
        console.error("Failed to read file:", err.message);
        return;
    }

    try {
        const users = JSON.parse(data);
        console.log("Loaded users:", users.length);
    } catch (parseError) {
        console.error("Failed to parse JSON:", parseError.message);
    }
});

// Writing file with callback
const userData = {
    name: "Bob",
    email: "bob@example.com",
    created: new Date().toISOString()
};

fs.writeFile('new-user.json', JSON.stringify(userData, null, 2), (err) => {
    if (err) {
        console.error("Failed to write file:", err.message);
    } else {
        console.log("User data saved successfully");
    }
});
```

Callback Functions in Node.js

Creating Your Own Callback Functions:

```
// Database simulation with callbacks
function queryDatabase(sql, params, callback) {
    // Simulate network delay
    const delay = Math.random() * 1000 + 500; // 500-1500ms

    setTimeout(() => {
        // Simulate potential errors
        if (Math.random() < 0.1) { // 10% chance of error
            callback(new Error("Database connection failed"), null);
            return;
        }

        // Simulate successful query
        const mockResults = [
            {id: 1, name: "Alice", email: "alice@example.com"},
            {id: 2, name: "Bob", email: "bob@example.com"}
        ];

        callback(null, mockResults);
    }, delay);
}
```

```
// Usage with error handling
queryDatabase("SELECT * FROM users", [], (err, results) => {
    if (err) {
        console.error("Database query failed:", err.message);
        // Handle error - maybe retry, log, or show user message
        return;
    }

    console.log("Query successful:");
    results.forEach(user => {
        console.log(`- ${user.name} (${user.email})`);
    });
});
```

Callback Functions in Node.js

Callback Best Practices:

```
// ✅ GOOD: Consistent error-first pattern
function processData(data, callback) {
    if (!data) {
        callback(new Error("Data is required"), null);
        return;
    }

    // Process asynchronously
    setImmediate(() => {
        try {
            const result = data.toUpperCase();
            callback(null, result);
        } catch (error) {
            callback(error, null);
        }
    });
}
```

```
// ✅ GOOD: Always handle both error and success
processData("hello", (err, result) => {
    if (err) {
        console.error("Processing failed:", err.message);
    } else {
        console.log("Result:", result);
    }
});
```

```
// ❌ AVOID: Inconsistent callback patterns
function badFunction(data, callback) {
    if (!data) {
        callback("Error message"); // Should be Error object
        return;
    }

    callback(processedData); // Missing error parameter
}
```

Practice: Module Creation and Async Operations

Your Task:
Create two separate modules and demonstrate async vs sync code

Part 1: Create math.js module

- Create math.js with functions: add, subtract, multiply, divide
- Add input validation (check for numbers)
- Export functions using module.exports
- Include a constant PI = 3.14159

Part 2: Create app.js

- Import the math module using require()
- Use the math functions with different inputs
- Demonstrate async vs sync with setTimeout

Practice: Module Creation and Async Operations

Part 3: Async vs Sync Demo

- Create a function that simulates a slow operation
- Show the difference between blocking and non-blocking execution
- Use callbacks to handle async result

THANK YOU

ADAEGY