

# Node.js and Express.js Complete Tutorial Series

---

A comprehensive tutorial series covering Node.js fundamentals and Express.js web development, from basic concepts to advanced topics including testing, authentication, and deployment.

## Tutorial Contents

### Part 1: Node.js Fundamentals

1. **Node.js Fundamentals** - Introduction to Node.js, its architecture, and core concepts
2. **Node.js Installation and Basics** - Setting up Node.js development environment
3. **NPM Package Management** - Managing dependencies and packages
4. **Node.js Core Modules** - Built-in modules and their usage
5. **Creating HTTP Servers** - Building basic HTTP servers with Node.js

### Part 2: Express.js Framework

6. **Express Introduction** - Getting started with Express.js framework
7. **Request and Response Objects** - Understanding Express req and res objects
8. **Serving Static Files** - Handling static assets in Express
9. **Environment Variables with dotenv** - Managing configuration and secrets
10. **Express Router and Modular Routes** - Organizing routes and middleware

### Part 3: Advanced Express.js

11. **Template Engines (EJS & Pug)** - Server-side rendering with template engines
12. **RESTful API Design Principles** - Best practices for API design
13. **CRUD Operations in Express** - Building complete CRUD functionality
14. **Middleware in Express** - Custom and third-party middleware
15. **Database Integration (MongoDB & Mongoose)** - Working with databases

### Part 4: Security and Advanced Features

16. **Authentication and Authorization** - User authentication, JWT, sessions, and security
17. **File Upload Handling** - Managing file uploads and storage
18. **Testing Express Applications** - Comprehensive testing strategies and implementation

## Learning Objectives

By completing this tutorial series, you will:

- ☒ Understand Node.js fundamentals and asynchronous programming
- ☒ Master Express.js framework for web development
- ☒ Build RESTful APIs with proper design principles
- ☒ Implement authentication and authorization systems
- ☒ Work with databases using MongoDB and Mongoose
- ☒ Handle file uploads and static assets
- ☒ Write comprehensive tests for your applications

- ☒ Apply security best practices
- ☒ Structure and organize large-scale applications

## Prerequisites

- Basic knowledge of JavaScript (ES6+)
- Understanding of web development concepts
- Familiarity with command line/terminal
- Basic understanding of HTTP protocol

## Getting Started

1. **Start with the fundamentals:** Begin with [Node.js Fundamentals](#)
2. **Follow the sequence:** Each chapter builds upon previous concepts
3. **Practice along:** Code examples are provided throughout
4. **Build projects:** Apply concepts by building real applications

## What You'll Build

Throughout this series, you'll build:

- **Basic HTTP servers** with Node.js core modules
- **RESTful APIs** with Express.js
- **User authentication system** with JWT and sessions
- **File upload functionality** with validation and storage
- **Database-driven applications** with MongoDB
- **Comprehensive test suites** with Jest and Supertest
- **Production-ready applications** with security and best practices

## Technologies Covered

### Core Technologies

- **Node.js** - JavaScript runtime environment
- **Express.js** - Web application framework
- **MongoDB** - NoSQL database
- **Mongoose** - MongoDB object modeling

### Development Tools

- **NPM** - Package management
- **dotenv** - Environment variable management
- **Jest** - Testing framework
- **Supertest** - HTTP testing
- **Multer** - File upload handling
- **bcrypt** - Password hashing
- **jsonwebtoken** - JWT implementation

### Template Engines

- **EJS** - Embedded JavaScript templates
- **Pug** - Clean, whitespace-sensitive syntax

## How to Use This Tutorial

### For Beginners

1. Start from Chapter 1 and progress sequentially
2. Complete all code examples
3. Build the suggested projects
4. Review concepts before moving to the next chapter

### For Intermediate Developers

1. Review the table of contents
2. Jump to specific topics of interest
3. Use as a reference guide
4. Focus on advanced chapters (16-18)

### For Advanced Developers

1. Use as a comprehensive reference
2. Focus on best practices and patterns
3. Implement testing strategies
4. Apply security considerations

## Learning Path Recommendations

### Backend Developer Path

1. Node.js Fundamentals (Chapters 1-5)
2. Express.js Basics (Chapters 6-10)
3. Database Integration (Chapter 15)
4. Authentication (Chapter 16)
5. Testing (Chapter 18)

### Full-Stack Developer Path

1. Complete all chapters sequentially
2. Focus on template engines (Chapter 11)
3. Build complete applications
4. Implement both API and web interfaces

### API Developer Path

1. Node.js Fundamentals (Chapters 1-5)
2. Express.js and REST APIs (Chapters 6, 12-15)
3. Authentication and Security (Chapter 16)
4. File Handling (Chapter 17)
5. Testing (Chapter 18)

## Additional Resources

- **Git Logs** - Version control best practices
- **Official Documentation:**
  - [Node.js Documentation](#)
  - [Express.js Documentation](#)
  - [MongoDB Documentation](#)
  - [Mongoose Documentation](#)

## Contributing

This tutorial series is designed to be comprehensive and up-to-date. If you find any issues or have suggestions for improvements:

1. Review the content thoroughly
2. Check for outdated information
3. Suggest additional topics or examples
4. Report any errors or inconsistencies

## License

This tutorial series is provided for educational purposes. Feel free to use the code examples and concepts in your own projects.

## Updates and Maintenance

This tutorial series is regularly updated to reflect:

- Latest Node.js and Express.js versions
- Current best practices
- New security considerations
- Updated dependencies and tools

---

### Happy Learning!

Start your journey with [Node.js Fundamentals](#) and build amazing web applications with Node.js and Express.js!

# Node.js Fundamentals: Understanding the Runtime

---

## Overview

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine that allows you to run JavaScript on the server side. Unlike traditional server-side languages, Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient for building scalable network applications.

## Key Concepts

## What is Node.js?

Node.js is not a programming language or a framework – it's a runtime environment that executes JavaScript code outside of a web browser. It was created by Ryan Dahl in 2009 to enable JavaScript to be used for server-side development.

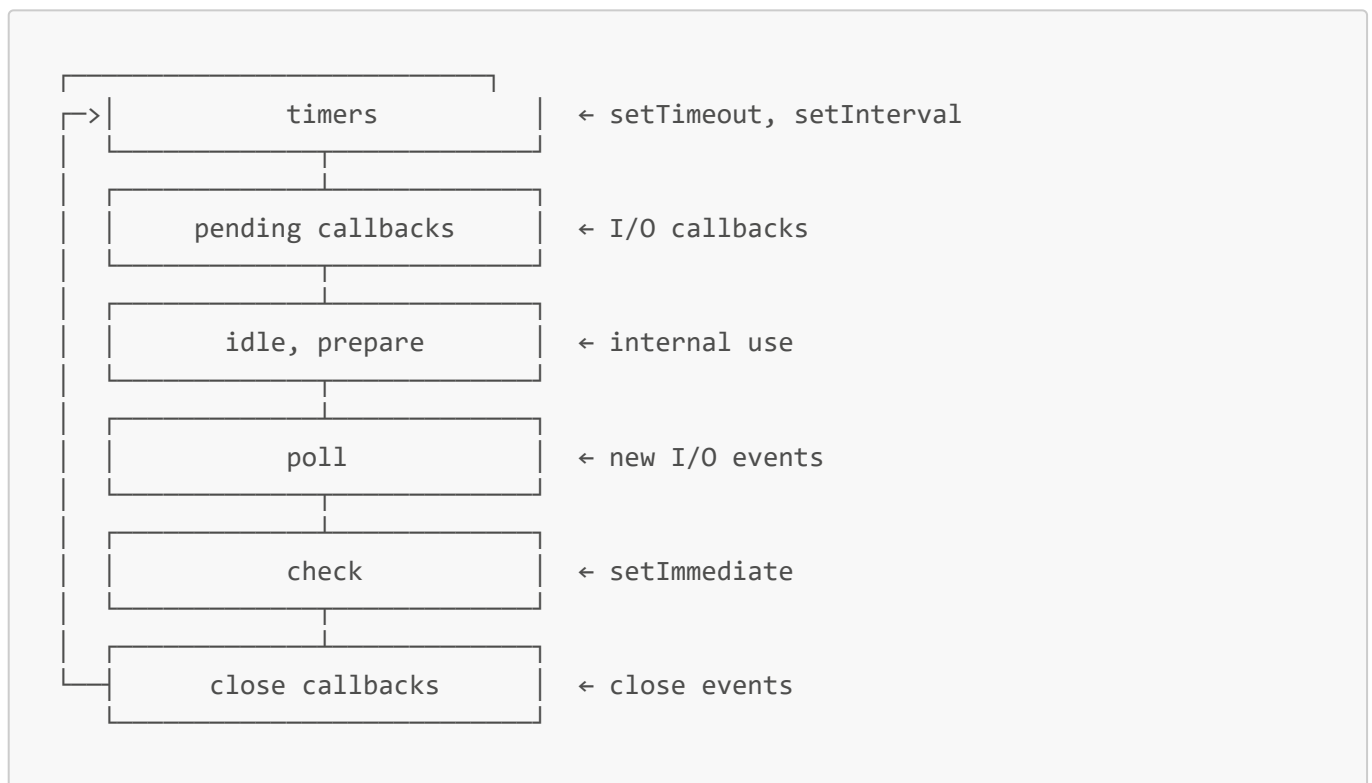
## The V8 Engine

Node.js uses Google's V8 JavaScript engine, the same engine that powers Google Chrome. V8 compiles JavaScript directly to native machine code, making it extremely fast. This engine:

- Compiles JavaScript to machine code
- Handles memory allocation and garbage collection
- Provides the JavaScript runtime environment

## The Event Loop

The event loop is the heart of Node.js's non-blocking I/O model. It's a single-threaded loop that handles all asynchronous operations.



## Non-blocking I/O

Traditional server models create a new thread for each request, which can be resource-intensive. Node.js uses a single thread with an event loop to handle multiple concurrent operations without blocking.

## Example Code

### Basic Node.js Script

Create a file called `hello.js`:

```
// hello.js
console.log("Hello, Node.js!");
console.log("Current directory:", process.cwd());
console.log("Node.js version:", process.version);
console.log("Platform:", process.platform);
```

Run it with:

```
node hello.js
```

## Demonstrating the Event Loop

```
// event-loop-demo.js
console.log("Start");

// Immediate execution
setImmediate(() => {
  console.log("setImmediate");
});

// Timer
setTimeout(() => {
  console.log("setTimeout");
}, 0);

// Process next tick (highest priority)
process.nextTick(() => {
  console.log("nextTick");
});

// Promise (microtask)
Promise.resolve().then(() => {
  console.log("Promise");
});

console.log("End");

// Output order:
// Start
// End
// nextTick
// Promise
// setTimeout
// setImmediate
```

## Blocking vs Non-blocking Example

```
// blocking-example.js
const fs = require("fs");

console.log("Start");

// Blocking (synchronous) - avoid in production
try {
  const data = fs.readFileSync("package.json", "utf8");
  console.log("File read synchronously");
} catch (err) {
  console.log("File not found (sync)");
}

// Non-blocking (asynchronous) - preferred
fs.readFile("package.json", "utf8", (err, data) => {
  if (err) {
    console.log("File not found (async)");
  } else {
    console.log("File read asynchronously");
  }
});

console.log("End");

// Output:
// Start
// File read synchronously (or error)
// End
// File read asynchronously (or error)
```

## Real-World Use Case

### CPU-Intensive vs I/O-Intensive Tasks

Node.js excels at I/O-intensive applications but struggles with CPU-intensive tasks:

```
// io-intensive.js - Good for Node.js
const fs = require("fs");
const http = require("http");

const server = http.createServer((req, res) => {
  // Multiple file operations can happen concurrently
  fs.readFile("data1.txt", (err, data1) => {
    fs.readFile("data2.txt", (err, data2) => {
      fs.readFile("data3.txt", (err, data3) => {
        res.end("All files processed");
      });
    });
  });
});
```

```
server.listen(3000);
```

```
// cpu-intensive.js - Not ideal for Node.js
function fibonacci(n) {
  if (n < 2) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

console.log("Start");
const result = fibonacci(40); // This blocks everything
console.log("Result:", result);
console.log("End");
```

## Best Practices

### 1. Always Use Asynchronous Operations

```
// ✗ Bad - blocks the event loop
const data = fs.readFileSync("file.txt");

// ✓ Good - non-blocking
fs.readFile("file.txt", (err, data) => {
  // Handle the data
});

// ✓ Even better - using promises/async-await
const data = await fs.promises.readFile("file.txt");
```

### 2. Handle Errors Properly

```
// ✗ Bad - unhandled errors can crash the app
fs.readFile("file.txt", (err, data) => {
  console.log(data.toString());
});

// ✓ Good - always handle errors
fs.readFile("file.txt", (err, data) => {
  if (err) {
    console.error("Error reading file:", err.message);
    return;
  }
  console.log(data.toString());
});
```



### 3. Use Environment Variables

```
// config.js
module.exports = {
  port: process.env.PORT || 3000,
  nodeEnv: process.env.NODE_ENV || "development",
  dbUrl: process.env.DATABASE_URL || "mongodb://localhost:27017/myapp",
};
```

### 4. Understand the Event Loop

- Don't block the event loop with synchronous operations
- Use `setImmediate()` for deferring execution
- Use `process.nextTick()` sparingly (it has the highest priority)

## Summary

Node.js revolutionizes server-side development by bringing JavaScript to the backend with its event-driven, non-blocking I/O model. Key takeaways:

- **V8 Engine:** Compiles JavaScript to machine code for high performance
- **Event Loop:** Single-threaded loop that handles asynchronous operations
- **Non-blocking I/O:** Allows handling multiple operations concurrently
- **Best Use Cases:** I/O-intensive applications like web servers, APIs, real-time apps
- **Avoid:** CPU-intensive tasks that can block the event loop

Understanding these fundamentals is crucial before diving into Express.js and building web applications. The event-driven nature of Node.js makes it perfect for building scalable network applications, but it requires a different mindset compared to traditional multi-threaded server environments.

Next, we'll explore how to install Node.js and start building our first applications.

## Node.js Installation and Running Basic Scripts

---

### Overview

Before you can start building applications with Node.js, you need to install it on your system and understand how to run JavaScript files. This chapter covers the installation process, version management, and running your first Node.js scripts.

## Key Concepts

### Node.js Installation Methods

There are several ways to install Node.js:

1. **Official Installer** - Download from [nodejs.org](https://nodejs.org)
2. **Package Managers** - Using npm, Chocolatey (Windows), Homebrew (macOS)

3. **Version Managers** - nvm (Node Version Manager)
4. **Docker** - Running Node.js in containers

## LTS vs Current Versions

- **LTS (Long Term Support)**: Stable, recommended for production
- **Current**: Latest features, may have breaking changes

## Node.js REPL

REPL (Read-Eval-Print Loop) is an interactive shell for testing JavaScript code quickly.

## Example Code

### Installation Steps

#### Method 1: Official Installer (Recommended for Beginners)

1. Visit [nodejs.org](https://nodejs.org)
2. Download the LTS version
3. Run the installer
4. Verify installation:

```
# Check Node.js version
node --version
# or
node -v

# Check npm version
npm --version
# or
npm -v
```

#### Method 2: Using nvm (Recommended for Developers)

##### Windows (nvm-windows):

```
# Download and install nvm-windows from GitHub
# Then install Node.js
nvm install lts
nvm use lts
```

##### macOS/Linux:

```
# Install nvm
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh | bash
```

```
# Restart terminal or run:
source ~/.bashrc

# Install latest LTS
nvm install --lts
nvm use --lts

# List installed versions
nvm list

# Switch between versions
nvm use 18.17.0
```

## Your First Node.js Script

Create a file called `first-script.js`:

```
// first-script.js
console.log("Welcome to Node.js!");
console.log("Today is:", new Date().toLocaleDateString());
console.log("Current working directory:", process.cwd());

// Access command line arguments
console.log("Command line arguments:", process.argv);

// Environment variables
console.log("Node environment:", process.env.NODE_ENV || "development");
```

Run it:

```
node first-script.js
```

## Working with Command Line Arguments

```
// args-demo.js
const args = process.argv.slice(2); // Remove 'node' and script name

if (args.length === 0) {
  console.log("No arguments provided!");
  console.log("Usage: node args-demo.js <name> <age>");
  process.exit(1);
}

const [name, age] = args;
console.log(`Hello ${name}, you are ${age} years old!`);

// Handle optional arguments
```

```
if (args[2]) {  
  console.log(`Additional info: ${args[2]}`);  
}
```

Run with arguments:

```
node args-demo.js John 25 "Software Developer"
```

## Using the Node.js REPL

Start the REPL by typing `node` without any arguments:

```
node
```

Then you can run JavaScript interactively:

```
> console.log('Hello from REPL!');  
Hello from REPL!  
undefined  
  
> const greeting = 'Hello World';  
undefined  
  
> greeting  
'Hello World'  
  
> Math.random()  
0.8394729834729834  
  
> .help // Show REPL commands  
> .exit // Exit REPL (or Ctrl+C twice)
```

## Useful REPL Commands

```
// In REPL:  
.help      // Show help  
.break     // Break out of multi-line expression  
.clear     // Clear the context  
.exit      // Exit REPL  
.save filename // Save session to file  
.load filename // Load file into session
```

## Creating a Simple Calculator

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

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

function multiply(a, b) {
  return a * b;
}

function divide(a, b) {
  if (b === 0) {
    throw new Error("Division by zero is not allowed");
  }
  return a / b;
}

// Get command line arguments
const [operation, num1, num2] = process.argv.slice(2);

if (!operation || !num1 || !num2) {
  console.log("Usage: node calculator.js <operation> <num1> <num2>");
  console.log("Operations: add, subtract, multiply, divide");
  process.exit(1);
}

const a = parseFloat(num1);
const b = parseFloat(num2);

if (isNaN(a) || isNaN(b)) {
  console.log("Please provide valid numbers");
  process.exit(1);
}

try {
  let result;

  switch (operation.toLowerCase()) {
    case "add":
      result = add(a, b);
      break;
    case "subtract":
      result = subtract(a, b);
      break;
    case "multiply":
      result = multiply(a, b);
      break;
    case "divide":
      result = divide(a, b);
```

```

        break;
    default:
        console.log("Unknown operation:", operation);
        process.exit(1);
    }

    console.log(`${a} ${operation} ${b} = ${result}`);
} catch (error) {
    console.error("Error:", error.message);
    process.exit(1);
}

```

Test the calculator:

```

node calculator.js add 10 5      # Output: 10 add 5 = 15
node calculator.js multiply 7 8  # Output: 7 multiply 8 = 56
node calculator.js divide 10 0   # Output: Error: Division by zero is not
allowed

```

## Real-World Use Case

### File Processing Script

Create a script that processes text files:

```

// file-processor.js
const fs = require("fs");
const path = require("path");

function processFile(filename) {
    // Check if file exists
    if (!fs.existsSync(filename)) {
        console.error(`File '${filename}' not found!`);
        return;
    }

    try {
        // Read file content
        const content = fs.readFileSync(filename, "utf8");

        // Process the content
        const lines = content.split("\n");
        const wordCount = content
            .split(/\s+/)
            .filter((word) => word.length > 0).length;
        const charCount = content.length;

        // Display statistics
        console.log(`\n📄 File: ${filename}`);
    }
}

```

```
console.log(`📊 Statistics:`);
console.log(`   Lines: ${lines.length}`);
console.log(`   Words: ${wordCount}`);
console.log(`   Characters: ${charCount}`);
console.log(`   Size: ${fs.statSync(filename).size} bytes`);

// Show first few lines
console.log(`\n📄 First 3 lines:`);
lines.slice(0, 3).forEach((line, index) => {
  console.log(`   ${index + 1}: ${line}`);
});
} catch (error) {
  console.error("Error processing file:", error.message);
}
}

// Get filename from command line
const filename = process.argv[2];

if (!filename) {
  console.log("Usage: node file-processor.js <filename>");
  process.exit(1);
}

processFile(filename);
```

Create a test file and run:

```
echo "Hello World\nThis is a test file\nWith multiple lines" > test.txt
node file-processor.js test.txt
```

## Best Practices

### 1. Version Management

```
# Always specify Node.js version in your project
echo "18.17.0" > .nvmrc

# Team members can then use:
nvm use # Automatically uses version from .nvmrc
```

### 2. Error Handling

```
// ❌ Bad - unhandled errors
const result = JSON.parse(userInput);

// ✅ Good - proper error handling
```

```
try {
  const result = JSON.parse(userInput);
  console.log("Parsed successfully:", result);
} catch (error) {
  console.error("Invalid JSON:", error.message);
  process.exit(1);
}
```

### 3. Exit Codes

```
// Use appropriate exit codes
process.exit(0); // Success
process.exit(1); // General error
process.exit(2); // Misuse of shell command
```

### 4. Environment Detection

```
// environment.js
const isDevelopment = process.env.NODE_ENV === "development";
const isProduction = process.env.NODE_ENV === "production";
const isTest = process.env.NODE_ENV === "test";

if (isDevelopment) {
  console.log("Running in development mode");
}

// Set environment when running
// NODE_ENV=production node app.js
```

### 5. Script Organization

```
// main.js - Entry point
function main() {
  try {
    // Your main logic here
    console.log("Application started successfully");
  } catch (error) {
    console.error("Application failed to start:", error.message);
    process.exit(1);
  }
}

// Only run if this file is executed directly
if (require.main === module) {
  main();
}
```



```
// Export for testing
module.exports = { main };
```

## Summary

Getting started with Node.js involves:

- **Installation:** Use official installer for beginners, nvm for developers
- **Version Management:** Always use LTS for production, consider nvm for multiple projects
- **Running Scripts:** Use `node filename.js` to execute JavaScript files
- **REPL:** Interactive shell for quick testing and experimentation
- **Command Line Arguments:** Access via `process.argv` for dynamic scripts
- **Error Handling:** Always handle errors gracefully and use appropriate exit codes

Key commands to remember:

```
node --version      # Check Node.js version
node script.js      # Run a script
node                # Start REPL
node -e "console.log('Hi')" # Execute inline code
```

Now that you have Node.js installed and understand the basics, you're ready to explore npm and start building more complex applications. The next chapter will cover npm fundamentals and package management.

# NPM: Node Package Manager Fundamentals

---

## Overview

npm (Node Package Manager) is the default package manager for Node.js and the world's largest software registry. It allows you to install, share, and manage dependencies for your Node.js projects. Understanding npm is crucial for modern JavaScript development.

## Key Concepts

What is npm?

npm serves three main purposes:

1. **Package Registry:** A massive database of JavaScript packages
2. **Command Line Tool:** For installing and managing packages
3. **Package Manager:** Handles dependencies and versions

## package.json

The `package.json` file is the heart of any Node.js project. It contains:

- Project metadata
- Dependencies
- Scripts
- Configuration

## Semantic Versioning (SemVer)

npm uses semantic versioning: **MAJOR.MINOR.PATCH**

- **MAJOR**: Breaking changes
- **MINOR**: New features (backward compatible)
- **PATCH**: Bug fixes

## Dependencies vs DevDependencies

- **dependencies**: Required for production
- **devDependencies**: Only needed for development

## Example Code

### Initializing a New Project

```
# Create a new directory
mkdir my-node-project
cd my-node-project

# Initialize npm (interactive)
npm init

# Initialize with defaults
npm init -y
# or
npm init --yes
```

### Sample package.json

```
{
  "name": "my-node-project",
  "version": "1.0.0",
  "description": "A sample Node.js project",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "node --watch index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": ["node", "javascript", "tutorial"],
  "author": "Your Name <your.email@example.com>",
  "license": "MIT",
```

```
"dependencies": {},  
"devDependencies": {}  
}
```

## Installing Packages

```
# Install a package and add to dependencies  
npm install express  
# or  
npm i express  
  
# Install multiple packages  
npm install express mongoose dotenv  
  
# Install as dev dependency  
npm install --save-dev nodemon  
# or  
npm i -D nodemon  
  
# Install globally  
npm install -g nodemon  
  
# Install specific version  
npm install express@4.18.0  
  
# Install from GitHub  
npm install user/repo
```

## Version Specifiers

```
{  
  "dependencies": {  
    "express": "^4.18.0", // Compatible version (4.x.x)  
    "mongoose": "~7.0.0", // Approximately equivalent (7.0.x)  
    "lodash": "4.17.21", // Exact version  
    "moment": ">=2.29.0", // Greater than or equal  
    "axios": "<1.0.0", // Less than  
    "dotenv": "*" // Any version (not recommended)  
  }  
}
```

## Package Management Commands

```
# List installed packages  
npm list  
npm ls
```

```
# List global packages
npm list -g --depth=0

# Check for outdated packages
npm outdated

# Update packages
npm update
npm update express

# Uninstall packages
npm uninstall express
npm remove express
npm rm express

# Uninstall global package
npm uninstall -g nodemon

# Clean npm cache
npm cache clean --force
```

## Working with Scripts

Create an `index.js` file:

```
// index.js
console.log("Hello from npm scripts!");
console.log("Environment:", process.env.NODE_ENV || "development");

// Simple HTTP server
const http = require("http");

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

const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Update `package.json` scripts:

```
{
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
```

```
"prod": "NODE_ENV=production node index.js",
"test": "echo \"Running tests...\" && node test.js",
"build": "echo \"Building project...\"",
"clean": "rm -rf node_modules package-lock.json",
"reinstall": "npm run clean && npm install",
"lint": "echo \"Linting code...\"",
"prestart": "echo \"Pre-start hook\"",
"poststart": "echo \"Post-start hook\""
}
}
```

Run scripts:

```
# Run scripts
npm start
npm run dev
npm test
npm run build

# List available scripts
npm run
```

## Creating a Complete Project Setup

```
# Initialize project
mkdir todo-app
cd todo-app
npm init -y

# Install dependencies
npm install express cors helmet morgan
npm install -D nodemon jest supertest
```

Update the `package.json`:

```
{
  "name": "todo-app",
  "version": "1.0.0",
  "description": "A simple todo application",
  "main": "server.js",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js",
    "test": "jest",
    "test:watch": "jest --watch",
    "lint": "echo \"Add linting here\"",
    "build": "echo \"Build process\""
  }
}
```

```
  },
  "keywords": ["todo", "express", "api"],
  "author": "Your Name",
  "license": "MIT",
  "dependencies": {
    "express": "^4.18.2",
    "cors": "^2.8.5",
    "helmet": "^7.0.0",
    "morgan": "^1.10.0"
  },
  "devDependencies": {
    "nodemon": "^3.0.1",
    "jest": "^29.6.2",
    "supertest": "^6.3.3"
  }
}
```

Create `server.js`:

```
// server.js
const express = require("express");
const cors = require("cors");
const helmet = require("helmet");
const morgan = require("morgan");

const app = express();
const PORT = process.env.PORT || 3000;

// Middleware
app.use(helmet());
app.use(cors());
app.use(morgan("combined"));
app.use(express.json());

// In-memory storage for todos
let todos = [
  { id: 1, text: "Learn Node.js", completed: false },
  { id: 2, text: "Build an API", completed: false },
];

// Routes
app.get("/api/todos", (req, res) => {
  res.json(todos);
});

app.post("/api/todos", (req, res) => {
  const { text } = req.body;

  if (!text) {
    return res.status(400).json({ error: "Text is required" });
  }
});
```

```
const newTodo = {
  id: todos.length + 1,
  text,
  completed: false,
};

todos.push(newTodo);
res.status(201).json(newTodo);
});

app.get("/health", (req, res) => {
  res.json({ status: "OK", timestamp: new Date().toISOString() });
});

app.listen(PORT, () => {
  console.log(`🚀 Server running on port ${PORT}`);
  console.log(`📖 API available at http://localhost:${PORT}/api/todos`);
});

module.exports = app;
```

## Real-World Use Case

### Package.json for a Production App

```
{
  "name": "production-api",
  "version": "2.1.0",
  "description": "Production-ready REST API",
  "main": "src/server.js",
  "engines": {
    "node": ">=18.0.0",
    "npm": ">=8.0.0"
  },
  "scripts": {
    "start": "node src/server.js",
    "dev": "nodemon src/server.js",
    "test": "jest --coverage",
    "test:watch": "jest --watch",
    "test:integration": "jest --testPathPattern=integration",
    "lint": "eslint src/",
    "lint:fix": "eslint src/ --fix",
    "format": "prettier --write src/",
    "build": "npm run lint && npm run test",
    "precommit": "npm run lint && npm run test",
    "docker:build": "docker build -t production-api .",
    "docker:run": "docker run -p 3000:3000 production-api"
  },
  "dependencies": {
    "express": "^4.18.2",
    "mongoose": "^7.4.0",
```

```
    "bcryptjs": "^2.4.3",
    "jsonwebtoken": "^9.0.1",
    "helmet": "^7.0.0",
    "cors": "^2.8.5",
    "morgan": "^1.10.0",
    "dotenv": "^16.3.1",
    "joi": "^17.9.2",
    "winston": "^3.10.0"
  },
  "devDependencies": {
    "nodemon": "^3.0.1",
    "jest": "^29.6.2",
    "supertest": "^6.3.3",
    "eslint": "^8.45.0",
    "prettier": "^3.0.0",
    "husky": "^8.0.3",
    "lint-staged": "^13.2.3"
  },
  "repository": {
    "type": "git",
    "url": "https://github.com/username/production-api.git"
  },
  "keywords": ["api", "express", "mongodb", "jwt"],
  "author": "Your Name <your.email@example.com>",
  "license": "MIT",
  "husky": {
    "hooks": {
      "pre-commit": "lint-staged"
    }
  },
  "lint-staged": {
    "*.js": ["eslint --fix", "prettier --write"]
  }
}
```

## NPM Configuration

Create `.npmrc` file for project-specific npm configuration:

```
# .npmrc
registry=https://registry.npmjs.org/
save-exact=true
engine-strict=true
fund=false
audit-level=moderate
```

## Security Best Practices



```
# Check for vulnerabilities
npm audit

# Fix vulnerabilities automatically
npm audit fix

# Force fix (may introduce breaking changes)
npm audit fix --force

# Install specific security updates
npm update --depth 3

# Check package info before installing
npm info express
npm view express versions --json
```

## Best Practices

### 1. Lock File Management

```
# Always commit package-lock.json
git add package-lock.json

# Use npm ci in production/CI
npm ci # Faster, reliable, reproducible builds
```

### 2. Version Management

```
# Bump version
npm version patch # 1.0.0 -> 1.0.1
npm version minor # 1.0.0 -> 1.1.0
npm version major # 1.0.0 -> 2.0.0

# Pre-release versions
npm version prerelease # 1.0.0 -> 1.0.1-0
```

### 3. Script Organization

```
{
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js",
    "test": "npm run test:unit && npm run test:integration",
    "test:unit": "jest --testPathPattern=unit",
    "test:integration": "jest --testPathPattern=integration",
```

```
"test:watch": "jest --watch",
"lint": "eslint .",
"lint:fix": "eslint . --fix",
"format": "prettier --write .",
"build": "npm run lint && npm run test",
"clean": "rm -rf node_modules dist",
"reset": "npm run clean && npm install"
}
}
```

#### 4. Environment-Specific Scripts

```
{
  "scripts": {
    "start": "node server.js",
    "start:dev": "NODE_ENV=development nodemon server.js",
    "start:prod": "NODE_ENV=production node server.js",
    "start:test": "NODE_ENV=test node server.js"
  }
}
```

#### 5. Useful npm Commands

```
# Show npm configuration
npm config list

# Set npm configuration
npm config set registry https://registry.npmjs.org/

# Show package information
npm info package-name

# Search for packages
npm search express

# Show dependency tree
npm ls --depth=0

# Find duplicate packages
npm ls --depth=0 | grep -E '^[^L]'
```

## Summary

npm is essential for Node.js development, providing:

- **Package Management:** Install, update, and remove dependencies
- **Version Control:** Semantic versioning and lock files for reproducible builds

- **Script Runner:** Automate common development tasks
- **Registry Access:** Access to millions of open-source packages

Key commands to remember:

```
npm init -y           # Initialize project
npm install package    # Install dependency
npm install -D package # Install dev dependency
npm run script-name    # Run npm script
npm audit             # Check security vulnerabilities
npm outdated          # Check for package updates
```

Mastering npm will significantly improve your development workflow and project management. Next, we'll explore Node.js core modules that provide built-in functionality for file system operations, HTTP servers, and more.

## Node.js Core Modules: Built-in Functionality

---

### Overview

Node.js comes with a rich set of built-in modules that provide essential functionality without requiring external dependencies. These core modules handle file system operations, HTTP requests, path manipulation, operating system interactions, and much more. Understanding these modules is crucial for effective Node.js development.

### Key Concepts

#### What are Core Modules?

Core modules are built into Node.js and can be imported using `require()` without installation. They provide:

- File system operations
- HTTP/HTTPS functionality
- Path and URL utilities
- Operating system information
- Cryptographic functions
- Stream processing
- And much more

#### CommonJS vs ES Modules

```
// CommonJS (traditional)
const fs = require("fs");

// ES Modules (modern)
import fs from "fs";
import { readFile } from "fs";
```

## Synchronous vs Asynchronous APIs

Most core modules provide both sync and async versions:

- Async: Non-blocking, callback-based or Promise-based
- Sync: Blocking, returns result directly

## Example Code

### File System (fs) Module

```
// fs-examples.js
const fs = require("fs");
const path = require("path");

// Reading files
// Asynchronous (preferred)
fs.readFile("data.txt", "utf8", (err, data) => {
  if (err) {
    console.error("Error reading file:", err.message);
    return;
  }
  console.log("File content:", data);
});

// Promise-based (modern approach)
const fsPromises = require("fs").promises;

async function readFileAsync() {
  try {
    const data = await fsPromises.readFile("data.txt", "utf8");
    console.log("File content (async/await):", data);
  } catch (error) {
    console.error("Error:", error.message);
  }
}

// Synchronous (use sparingly)
try {
  const data = fs.readFileSync("data.txt", "utf8");
  console.log("File content (sync):", data);
} catch (error) {
  console.error("Error:", error.message);
}

// Writing files
const content = "Hello, Node.js!\nThis is a test file.";

// Asynchronous write
fs.writeFile("output.txt", content, "utf8", (err) => {
```

```
    if (err) {
      console.error("Error writing file:", err);
      return;
    }
    console.log("File written successfully!");
  });

// Promise-based write
async function writeFileAsync() {
  try {
    await fsPromises.writeFile("output-async.txt", content, "utf8");
    console.log("File written successfully (async/await!)");
  } catch (error) {
    console.error("Error writing file:", error.message);
  }
}

// Appending to files
fs.appendFile("log.txt", `\n${new Date().toISOString()} - Log entry`, (err) => {
  if (err) {
    console.error("Error appending to file:", err);
    return;
  }
  console.log("Log entry added!");
});

// Working with directories
fs.readdir(".", (err, files) => {
  if (err) {
    console.error("Error reading directory:", err);
    return;
  }
  console.log("Files in current directory:", files);
});

// Creating directories
fs.mkdir("new-folder", { recursive: true }, (err) => {
  if (err) {
    console.error("Error creating directory:", err);
    return;
  }
  console.log("Directory created!");
});

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

  console.log("File stats:");
  console.log("  Size:", stats.size, "bytes");
  console.log("  Is file:", stats.isFile());
});
```

```
    console.log("  Is directory:", stats.isDirectory());
    console.log("  Created:", stats.birthtime);
    console.log("  Modified:", stats.mtime);
  });

  // 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");
    }
  });
});
```

## Path Module

```
// path-examples.js
const path = require("path");

// Path manipulation
const filePath = "/users/john/documents/file.txt";

console.log("Full path:", filePath);
console.log("Directory:", path.dirname(filePath)); // /users/john/documents
console.log("Filename:", path.basename(filePath)); // file.txt
console.log("Extension:", path.extname(filePath)); // .txt
console.log("Name without ext:", path.basename(filePath, ".txt")); // file

// Parsing paths
const parsed = path.parse(filePath);
console.log("Parsed path:", parsed);
// {
//   root: '/',
//   dir: '/users/john/documents',
//   base: 'file.txt',
//   ext: '.txt',
//   name: 'file'
// }

// Building paths
const newPath = path.format({
  dir: "/users/jane/projects",
  name: "app",
  ext: ".js",
});
console.log("Formatted path:", newPath); // /users/jane/projects/app.js

// Joining paths (cross-platform)
const joinedPath = path.join("/users", "john", "documents", "file.txt");
console.log("Joined path:", joinedPath);
```

```
// Resolving absolute paths
const absolutePath = path.resolve("documents", "file.txt");
console.log("Absolute path:", absolutePath);

// Relative paths
const relativePath = path.relative(
  "/users/john",
  "/users/john/documents/file.txt"
);
console.log("Relative path:", relativePath); // documents/file.txt

// Cross-platform path separators
console.log("Path separator:", path.sep); // \ on Windows, / on Unix
console.log("Path delimiter:", path.delimiter); // ; on Windows, : on Unix

// Normalize paths
const messyPath = "/users/john/../../jane/./documents//file.txt";
console.log("Normalized:", path.normalize(messyPath)); //
/users/jane/documents/file.txt
```

## HTTP Module

```
// http-examples.js
const http = require("http");
const url = require("url");
const querystring = require("querystring");

// Basic HTTP server
const server = http.createServer((req, res) => {
  // Parse URL
  const parsedUrl = url.parse(req.url, true);
  const pathname = parsedUrl.pathname;
  const query = parsedUrl.query;

  // Set CORS headers
  res.setHeader("Access-Control-Allow-Origin", "*");
  res.setHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
  res.setHeader("Access-Control-Allow-Headers", "Content-Type");

  // Handle different routes
  if (pathname === "/") {
    res.writeHead(200, { "Content-Type": "text/html" });
    res.end(`
      <h1>Welcome to Node.js HTTP Server</h1>
      <p>Current time: ${new Date().toISOString()}</p>
      <ul>
        <li><a href="/api/users">Users API</a></li>
        <li><a href="/api/health">Health Check</a></li>
        <li><a href="/api/info?name=John&age=30">Info with Query</a></li>
      </ul>
    `);
  }
});
```

```
} else if (pathname === "/api/users") {
  const users = [
    { id: 1, name: "John Doe", email: "john@example.com" },
    { id: 2, name: "Jane Smith", email: "jane@example.com" },
  ];

  res.writeHead(200, { "Content-Type": "application/json" });
  res.end(JSON.stringify(users, null, 2));
} else if (pathname === "/api/health") {
  const healthData = {
    status: "OK",
    timestamp: new Date().toISOString(),
    uptime: process.uptime(),
    memory: process.memoryUsage(),
  };

  res.writeHead(200, { "Content-Type": "application/json" });
  res.end(JSON.stringify(healthData, null, 2));
} else if (pathname === "/api/info") {
  const info = {
    message: "Hello from Node.js!",
    query: query,
    method: req.method,
    headers: req.headers,
    url: req.url,
  };

  res.writeHead(200, { "Content-Type": "application/json" });
  res.end(JSON.stringify(info, null, 2));
} else if (req.method === "POST" && pathname === "/api/data") {
  let body = "";

  req.on("data", (chunk) => {
    body += chunk.toString();
  });

  req.on("end", () => {
    try {
      const data = JSON.parse(body);
      const response = {
        message: "Data received successfully",
        receivedData: data,
        timestamp: new Date().toISOString(),
      };

      res.writeHead(200, { "Content-Type": "application/json" });
      res.end(JSON.stringify(response, null, 2));
    } catch (error) {
      res.writeHead(400, { "Content-Type": "application/json" });
      res.end(JSON.stringify({ error: "Invalid JSON" }));
    }
  });
} else {
  res.writeHead(404, { "Content-Type": "application/json" });
```



```

    res.end(JSON.stringify({ error: "Not Found" }));
  }
});

const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`🚀 Server running on http://localhost:${PORT}`);
});

// HTTP client example
function makeHttpRequest() {
  const options = {
    hostname: "jsonplaceholder.typicode.com",
    port: 443,
    path: "/posts/1",
    method: "GET",
    headers: {
      "User-Agent": "Node.js HTTP Client",
    },
  };

  const req = http.request(options, (res) => {
    let data = "";

    res.on("data", (chunk) => {
      data += chunk;
    });

    res.on("end", () => {
      console.log("Response:", JSON.parse(data));
    });
  });

  req.on("error", (error) => {
    console.error("Request error:", error);
  });

  req.end();
}

// Uncomment to test HTTP client
// makeHttpRequest();

```

## OS Module

```

// os-examples.js
const os = require("os");

// System information
console.log("Operating System Information:");
console.log("Platform:", os.platform()); // win32, darwin, linux

```

```
console.log("Architecture:", os.arch()); // x64, arm64, etc.
console.log("CPU Model:", os.cpus()[0].model);
console.log("CPU Cores:", os.cpus().length);
console.log(
  "Total Memory:",
  (os.totalmem() / 1024 / 1024 / 1024).toFixed(2),
  "GB"
);
console.log(
  "Free Memory:",
  (os.freemem() / 1024 / 1024 / 1024).toFixed(2),
  "GB"
);
console.log("Uptime:", (os.uptime() / 3600).toFixed(2), "hours");
console.log("Hostname:", os.hostname());
console.log("Home Directory:", os.homedir());
console.log("Temp Directory:", os.tmpdir());
console.log("Username:", os.userInfo().username);

// Network interfaces
console.log("\nNetwork Interfaces:");
const networkInterfaces = os.networkInterfaces();
Object.keys(networkInterfaces).forEach((interfaceName) => {
  const interfaces = networkInterfaces[interfaceName];
  interfaces.forEach((interface) => {
    if (interface.family === "IPv4" && !interface.internal) {
      console.log(`${interfaceName}: ${interface.address}`);
    }
  });
});

// Load average (Unix-like systems)
if (os.platform() !== "win32") {
  console.log("\nLoad Average:", os.loadavg());
}

// EOL (End of Line) character
console.log("\nEOL character code:", os.EOL.charCodeAt(0));
```

## URL Module

```
// url-examples.js
const url = require("url");
const { URL, URLSearchParams } = require("url");

// Legacy URL parsing
const urlString =
  "https://example.com:8080/path/to/resource?name=john&age=30#section1";
const parsedUrl = url.parse(urlString, true);

console.log("Legacy URL parsing:");
```

```
console.log("Protocol:", parsedUrl.protocol); // https:
console.log("Host:", parsedUrl.host); // example.com:8080
console.log("Hostname:", parsedUrl.hostname); // example.com
console.log("Port:", parsedUrl.port); // 8080
console.log("Pathname:", parsedUrl.pathname); // /path/to/resource
console.log("Query:", parsedUrl.query); // { name: 'john', age: '30' }
console.log("Hash:", parsedUrl.hash); // #section1

// Modern URL API
const modernUrl = new URL(urlString);

console.log("\nModern URL API:");
console.log("Origin:", modernUrl.origin); // https://example.com:8080
console.log("Protocol:", modernUrl.protocol); // https:
console.log("Host:", modernUrl.host); // example.com:8080
console.log("Hostname:", modernUrl.hostname); // example.com
console.log("Port:", modernUrl.port); // 8080
console.log("Pathname:", modernUrl.pathname); // /path/to/resource
console.log("Search:", modernUrl.search); // ?name=john&age=30
console.log("Hash:", modernUrl.hash); // #section1

// Working with search parameters
const params = new URLSearchParams(modernUrl.search);
console.log("\nSearch Parameters:");
console.log("Name:", params.get("name")); // john
console.log("Age:", params.get("age")); // 30

// Adding/modifying parameters
params.set("city", "New York");
params.append("hobby", "reading");
params.append("hobby", "coding");

console.log("Modified search:", params.toString());

// Building URLs
const baseUrl = "https://api.example.com";
const endpoint = "/users";
const queryParams = new URLSearchParams({
  page: "1",
  limit: "10",
  sort: "name",
});

const apiUrl = new URL(endpoint, baseUrl);
apiUrl.search = queryParams.toString();

console.log("\nBuilt API URL:", apiUrl.toString());
```

## Real-World Use Case

### File-based Logger with Core Modules

```
// logger.js
const fs = require("fs");
const path = require("path");
const os = require("os");

class FileLogger {
  constructor(logDir = "logs") {
    this.logDir = logDir;
    this.ensureLogDirectory();
  }

  ensureLogDirectory() {
    if (!fs.existsSync(this.logDir)) {
      fs.mkdirSync(this.logDir, { recursive: true });
    }
  }

  getLogFileName() {
    const date = new Date().toISOString().split("T")[0];
    return path.join(this.logDir, `app-${date}.log`);
  }

  formatLogEntry(level, message, meta = {}) {
    const timestamp = new Date().toISOString();
    const hostname = os.hostname();
    const pid = process.pid;

    const logEntry = {
      timestamp,
      level: level.toUpperCase(),
      message,
      hostname,
      pid,
      ...meta,
    };

    return JSON.stringify(logEntry) + os.EOL;
  }

  async writeLog(level, message, meta = {}) {
    const logEntry = this.formatLogEntry(level, message, meta);
    const logFile = this.getLogFileName();

    try {
      await fs.promises.appendFile(logFile, logEntry, "utf8");
    } catch (error) {
      console.error("Failed to write log:", error.message);
    }
  }

  info(message, meta = {}) {
    return this.writeLog("info", message, meta);
  }
}
```

```
error(message, meta = {}) {
  return this.writeLog("error", message, meta);
}

warn(message, meta = {}) {
  return this.writeLog("warn", message, meta);
}

debug(message, meta = {}) {
  return this.writeLog("debug", message, meta);
}

async getLogs(date = null) {
  const targetDate = date || new Date().toISOString().split("T")[0];
  const logFile = path.join(this.logDir, `app-${targetDate}.log`);

  try {
    const content = await fs.promises.readFile(logFile, "utf8");
    return content
      .split(os.EOL)
      .filter((line) => line.trim())
      .map((line) => JSON.parse(line));
  } catch (error) {
    if (error.code === "ENOENT") {
      return [];
    }
    throw error;
  }
}

async getLogStats() {
  try {
    const files = await fs.promises.readdir(this.logDir);
    const logFiles = files.filter((file) => file.endsWith(".log"));

    const stats = await Promise.all(
      logFiles.map(async (file) => {
        const filePath = path.join(this.logDir, file);
        const stat = await fs.promises.stat(filePath);
        return {
          file,
          size: stat.size,
          created: stat.birthtime,
          modified: stat.mtime,
        };
      })
    );

    return stats;
  } catch (error) {
    console.error("Failed to get log stats:", error.message);
    return [];
  }
}
```

```
}  
}  
  
// Usage example  
async function demonstrateLogger() {  
  const logger = new FileLogger();  
  
  // Log some entries  
  await logger.info("Application started", { version: "1.0.0" });  
  await logger.warn("This is a warning", { component: "auth" });  
  await logger.error("An error occurred", {  
    error: "Database connection failed",  
  });  
  
  // Get today's logs  
  const logs = await logger.getLogs();  
  console.log("Today's logs:", logs);  
  
  // Get log file statistics  
  const stats = await logger.getLogStats();  
  console.log("Log file stats:", stats);  
}  
  
// Export for use in other modules  
module.exports = FileLogger;  
  
// Run demonstration if this file is executed directly  
if (require.main === module) {  
  demonstrateLogger().catch(console.error);  
}
```

## Best Practices

### 1. Always Use Asynchronous APIs

```
// ✗ Bad - blocks the event loop  
const data = fs.readFileSync("large-file.txt");  
  
// ✓ Good - non-blocking  
fs.readFile("large-file.txt", (err, data) => {  
  // Handle result  
});  
  
// ✓ Even better - modern async/await  
const data = await fs.promises.readFile("large-file.txt");
```

### 2. Handle Errors Properly

```
// ✗ Bad - unhandled errors
fs.readFile("file.txt", (err, data) => {
  console.log(data.toString());
});

// ✓ Good - proper error handling
fs.readFile("file.txt", (err, data) => {
  if (err) {
    if (err.code === "ENOENT") {
      console.log("File not found");
    } else {
      console.error("Error reading file:", err.message);
    }
    return;
  }
  console.log(data.toString());
});
```

### 3. Use Path Module for Cross-Platform Compatibility

```
// ✗ Bad - platform-specific
const filePath = "logs/app.log";

// ✓ Good - cross-platform
const filePath = path.join("logs", "app.log");
```

### 4. Validate Input and Handle Edge Cases

```
function safeReadFile(filename) {
  if (!filename || typeof filename !== "string") {
    throw new Error("Filename must be a non-empty string");
  }

  const normalizedPath = path.normalize(filename);

  // Prevent directory traversal attacks
  if (normalizedPath.includes("..")) {
    throw new Error("Invalid file path");
  }

  return fs.promises.readFile(normalizedPath, "utf8");
}
```

### 5. Use Streams for Large Files

```
// For large files, use streams instead of reading everything into memory
const fs = require("fs");
const readline = require("readline");

async function processLargeFile(filename) {
  const fileStream = fs.createReadStream(filename);
  const rl = readline.createInterface({
    input: fileStream,
    crlfDelay: Infinity,
  });

  for await (const line of rl) {
    // Process each line without loading entire file into memory
    console.log(line);
  }
}
```

## Summary

Node.js core modules provide essential functionality for:

- **File System (fs):** Reading, writing, and manipulating files and directories
- **Path:** Cross-platform path manipulation and resolution
- **HTTP:** Creating servers and making HTTP requests
- **OS:** Accessing operating system information and utilities
- **URL:** Parsing and manipulating URLs

Key principles:

- Always prefer asynchronous APIs to avoid blocking the event loop
- Handle errors appropriately with proper error checking
- Use path module for cross-platform file path handling
- Validate inputs and handle edge cases
- Consider using streams for large data processing

These core modules form the foundation for building robust Node.js applications. Next, we'll explore how to create HTTP servers and build our first web applications.

## Creating HTTP Servers with Node.js

---

### Overview

Building HTTP servers is one of the most common use cases for Node.js. The built-in `http` module provides everything you need to create web servers, handle requests, serve files, and build APIs. This chapter covers creating HTTP servers from scratch, handling different HTTP methods, routing, and serving various types of content.

### Key Concepts



## HTTP Protocol Basics

HTTP (HyperText Transfer Protocol) is a request-response protocol:

- **Client** sends a request to the server
- **Server** processes the request and sends back a response
- Each request contains: method, URL, headers, and optional body
- Each response contains: status code, headers, and optional body

## HTTP Methods

- **GET**: Retrieve data
- **POST**: Create new data
- **PUT**: Update existing data
- **DELETE**: Remove data
- **PATCH**: Partial update
- **HEAD**: Get headers only
- **OPTIONS**: Get allowed methods

## Status Codes

- **2xx**: Success (200 OK, 201 Created)
- **3xx**: Redirection (301 Moved, 304 Not Modified)
- **4xx**: Client Error (400 Bad Request, 404 Not Found)
- **5xx**: Server Error (500 Internal Server Error)

## Example Code

### Basic HTTP Server

```
// basic-server.js
const http = require("http");

// Create server
const server = http.createServer((req, res) => {
  // Set response headers
  res.writeHead(200, {
    "Content-Type": "text/plain",
    "Access-Control-Allow-Origin": "*",
  });

  // Send response
  res.end("Hello, World! This is my first Node.js server.");
});

// Start server
const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`🚀 Server running on http://localhost:${PORT}`);
});
```

```
// Handle server errors
server.on("error", (error) => {
  console.error("Server error:", error.message);
});

// Graceful shutdown
process.on("SIGTERM", () => {
  console.log("Received SIGTERM, shutting down gracefully");
  server.close(() => {
    console.log("Server closed");
    process.exit(0);
  });
});
```

## Handling Different HTTP Methods

```
// method-handler.js
const http = require("http");
const url = require("url");

// In-memory data store
let users = [
  { id: 1, name: "John Doe", email: "john@example.com" },
  { id: 2, name: "Jane Smith", email: "jane@example.com" },
];

let nextId = 3;

const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url, true);
  const path = parsedUrl.pathname;
  const method = req.method;
  const query = parsedUrl.query;

  // Set common headers
  res.setHeader("Content-Type", "application/json");
  res.setHeader("Access-Control-Allow-Origin", "*");
  res.setHeader(
    "Access-Control-Allow-Methods",
    "GET, POST, PUT, DELETE, OPTIONS"
  );
  res.setHeader("Access-Control-Allow-Headers", "Content-Type");

  // Handle preflight requests
  if (method === "OPTIONS") {
    res.writeHead(200);
    res.end();
    return;
  }
});
```

```
// Route handling
if (path === "/api/users") {
  handleUsersRoute(req, res, method, query);
} else if (path.startsWith("/api/users/")) {
  const userId = parseInt(path.split("/")[3]);
  handleUserRoute(req, res, method, userId);
} else if (path === "/") {
  handleHomeRoute(req, res);
} else {
  handle404(req, res);
}
});

function handleUsersRoute(req, res, method, query) {
  switch (method) {
    case "GET":
      // Get all users with optional filtering
      let filteredUsers = users;

      if (query.name) {
        filteredUsers = users.filter((user) =>
          user.name.toLowerCase().includes(query.name.toLowerCase())
        );
      }

      res.writeHead(200);
      res.end(
        JSON.stringify(
          {
            success: true,
            data: filteredUsers,
            count: filteredUsers.length,
          },
          null,
          2
        )
      );
      break;

    case "POST":
      // Create new user
      let body = "";

      req.on("data", (chunk) => {
        body += chunk.toString();
      });

      req.on("end", () => {
        try {
          const userData = JSON.parse(body);

          // Validate required fields
          if (!userData.name || !userData.email) {
            res.writeHead(400);
          }
        }
      });
    }
  }
}
```

```
    res.end(
      JSON.stringify({
        success: false,
        error: "Name and email are required",
      })
    );
    return;
  }

  // Check if email already exists
  const existingUser = users.find(
    (user) => user.email === userData.email
  );
  if (existingUser) {
    res.writeHead(409);
    res.end(
      JSON.stringify({
        success: false,
        error: "Email already exists",
      })
    );
    return;
  }

  // Create new user
  const newUser = {
    id: nextId++,
    name: userData.name,
    email: userData.email,
  };

  users.push(newUser);

  res.writeHead(201);
  res.end(
    JSON.stringify(
      {
        success: true,
        data: newUser,
        message: "User created successfully",
      },
      null,
      2
    )
  );
} catch (error) {
  res.writeHead(400);
  res.end(
    JSON.stringify({
      success: false,
      error: "Invalid JSON data",
    })
  );
}
```

```
    });  
    break;  
  
    default:  
    res.writeHead(405);  
    res.end(  
      JSON.stringify({  
        success: false,  
        error: "Method not allowed",  
      })  
    );  
  }  
}  
  
function handleUserRoute(req, res, method, userId) {  
  const userIndex = users.findIndex((user) => user.id === userId);  
  
  switch (method) {  
    case "GET":  
      if (userIndex === -1) {  
        res.writeHead(404);  
        res.end(  
          JSON.stringify({  
            success: false,  
            error: "User not found",  
          })  
        );  
        return;  
      }  
  
      res.writeHead(200);  
      res.end(  
        JSON.stringify(  
          {  
            success: true,  
            data: users[userIndex],  
          },  
          null,  
          2  
        )  
      );  
      break;  
  
    case "PUT":  
      if (userIndex === -1) {  
        res.writeHead(404);  
        res.end(  
          JSON.stringify({  
            success: false,  
            error: "User not found",  
          })  
        );  
        return;  
      }  
    }
```

```
let body = "";
req.on("data", (chunk) => {
  body += chunk.toString();
});

req.on("end", () => {
  try {
    const userData = JSON.parse(body);

    // Update user
    users[userIndex] = {
      ...users[userIndex],
      ...userData,
      id: userId, // Ensure ID doesn't change
    };

    res.writeHead(200);
    res.end(
      JSON.stringify(
        {
          success: true,
          data: users[userIndex],
          message: "User updated successfully",
        },
        null,
        2
      )
    );
  } catch (error) {
    res.writeHead(400);
    res.end(
      JSON.stringify({
        success: false,
        error: "Invalid JSON data",
      })
    );
  }
});
break;

case "DELETE":
  if (userIndex === -1) {
    res.writeHead(404);
    res.end(
      JSON.stringify({
        success: false,
        error: "User not found",
      })
    );
    return;
  }

  const deletedUser = users.splice(userIndex, 1)[0];
```

```

    res.writeHead(200);
    res.end(
      JSON.stringify(
        {
          success: true,
          data: deletedUser,
          message: "User deleted successfully",
        },
        null,
        2
      )
    );
    break;

default:
  res.writeHead(405);
  res.end(
    JSON.stringify({
      success: false,
      error: "Method not allowed",
    })
  );
}
}

function handleHomeRoute(req, res) {
  res.setHeader("Content-Type", "text/html");
  res.writeHead(200);
  res.end(`
    <!DOCTYPE html>
    <html>
    <head>
      <title>Node.js API Server</title>
      <style>
        body { font-family: Arial, sans-serif; margin: 40px; }
        .endpoint { background: #f5f5f5; padding: 10px; margin: 10px 0;
border-radius: 5px; }
        .method { font-weight: bold; color: #007acc; }
      </style>
    </head>
    <body>
      <h1>🚀 Node.js API Server</h1>
      <p>Welcome to the Node.js HTTP server! Here are the available
endpoints:</p>

      <h2>📋 API Endpoints</h2>

      <div class="endpoint">
        <span class="method">GET</span> /api/users - Get all users
        <br><small>Query params: ?name=searchTerm</small>
      </div>

      <div class="endpoint">

```

```

        <span class="method">POST</span> /api/users - Create a new user
        <br><small>Body: {"name": "John Doe", "email": "john@example.com"}
    </small>
</div>

<div class="endpoint">
    <span class="method">GET</span> /api/users/:id - Get user by ID
</div>

<div class="endpoint">
    <span class="method">PUT</span> /api/users/:id - Update user by ID
    <br><small>Body: {"name": "Updated Name", "email":
"updated@example.com"}</small>
</div>

<div class="endpoint">
    <span class="method">DELETE</span> /api/users/:id - Delete user by
ID
</div>

<h2><img alt="pencil icon" data-bbox="258 388 278 403"/> Test the API</h2>
<p>You can test these endpoints using:</p>
<ul>
    <li>Browser (for GET requests)</li>
    <li>Postman</li>
    <li>Thunder Client (VS Code extension)</li>
    <li>curl commands</li>
</ul>

<h3>Example curl commands:</h3>
<pre>
# Get all users
curl http://localhost:3000/api/users

# Create a user
curl -X POST http://localhost:3000/api/users \
  -H "Content-Type: application/json" \
  -d '{"name":"Alice Johnson","email":"alice@example.com"}'

# Get user by ID
curl http://localhost:3000/api/users/1

# Update user
curl -X PUT http://localhost:3000/api/users/1 \
  -H "Content-Type: application/json" \
  -d '{"name":"Alice Smith"}'

# Delete user
curl -X DELETE http://localhost:3000/api/users/1
    </pre>
</body>
</html>
`);
}

```



```
function handle404(req, res) {
  res.writeHead(404);
  res.end(
    JSON.stringify(
      {
        success: false,
        error: "Endpoint not found",
        path: req.url,
        method: req.method,
      },
      null,
      2
    )
  );
}

const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`🚀 Server running on http://localhost:${PORT}`);
  console.log(`📖 API documentation available at http://localhost:${PORT}`);
});
```

## File Server

```
// file-server.js
const http = require("http");
const fs = require("fs");
const path = require("path");
const url = require("url");

// MIME types for different file extensions
const mimeTypes = {
  ".html": "text/html",
  ".css": "text/css",
  ".js": "text/javascript",
  ".json": "application/json",
  ".png": "image/png",
  ".jpg": "image/jpeg",
  ".jpeg": "image/jpeg",
  ".gif": "image/gif",
  ".svg": "image/svg+xml",
  ".ico": "image/x-icon",
  ".txt": "text/plain",
  ".pdf": "application/pdf",
};

const server = http.createServer((req, res) => {
  const parsedUrl = url.parse(req.url);
  let pathname = parsedUrl.pathname;
```

```
// Security: prevent directory traversal
pathname = path.normalize(pathname);
if (pathname.includes("..")) {
  res.writeHead(403);
  res.end("Forbidden: Directory traversal not allowed");
  return;
}

// Default to index.html for root path
if (pathname === "/") {
  pathname = "/index.html";
}

// Construct file path
const filePath = path.join(__dirname, "public", pathname);

// Get file extension and MIME type
const ext = path.extname(filePath).toLowerCase();
const mimeType = mimeTypes[ext] || "application/octet-stream";

// Check if file exists
fs.access(filePath, fs.constants.F_OK, (err) => {
  if (err) {
    // File not found
    serve404(res);
    return;
  }

  // Get file stats
  fs.stat(filePath, (err, stats) => {
    if (err) {
      serve500(res, err);
      return;
    }

    // Don't serve directories
    if (stats.isDirectory()) {
      serve403(res, "Directory listing not allowed");
      return;
    }

    // Set headers
    res.setHeader("Content-Type", mimeType);
    res.setHeader("Content-Length", stats.size);
    res.setHeader("Last-Modified", stats.mtime.toUTCString());
    res.setHeader("Cache-Control", "public, max-age=3600"); // Cache for 1 hour

    // Check if client has cached version
    const ifModifiedSince = req.headers["if-modified-since"];
    if (ifModifiedSince && new Date(ifModifiedSince) >= stats.mtime) {
      res.writeHead(304); // Not Modified
      res.end();
      return;
    }
  })
}
```

```
// Stream file to response
const readStream = fs.createReadStream(filePath);

readStream.on("error", (err) => {
  serve500(res, err);
});

res.writeHead(200);
readStream.pipe(res);
});
});
});

function serve404(res) {
  res.writeHead(404, { "Content-Type": "text/html" });
  res.end(`
    <!DOCTYPE html>
    <html>
    <head>
      <title>404 - Not Found</title>
      <style>
        body { font-family: Arial, sans-serif; text-align: center; margin-
top: 100px; }
        h1 { color: #e74c3c; }
      </style>
    </head>
    <body>
      <h1>404 - Page Not Found</h1>
      <p>The requested resource could not be found on this server.</p>
      <a href="/">Go back to home</a>
    </body>
    </html>
  `);
}

function serve403(res, message) {
  res.writeHead(403, { "Content-Type": "text/html" });
  res.end(`
    <!DOCTYPE html>
    <html>
    <head>
      <title>403 - Forbidden</title>
    </head>
    <body>
      <h1>403 - Forbidden</h1>
      <p>${message}</p>
    </body>
    </html>
  `);
}

function serve500(res, error) {
  console.error("Server error:", error);
```

```
res.writeHead(500, { "Content-Type": "text/html" });
res.end(`
  <!DOCTYPE html>
  <html>
  <head>
    <title>500 - Internal Server Error</title>
  </head>
  <body>
    <h1>500 - Internal Server Error</h1>
    <p>Something went wrong on the server.</p>
  </body>
  </html>
`);
}

// Create public directory and sample files
const publicDir = path.join(__dirname, "public");
if (!fs.existsSync(publicDir)) {
  fs.mkdirSync(publicDir);

  // Create sample index.html
  const indexHtml = `
<!DOCTYPE html>
<html>
<head>
  <title>Node.js File Server</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <h1>Welcome to Node.js File Server</h1>
  <p>This is a static file server built with Node.js!</p>
  <script src="script.js"></script>
</body>
</html>
`;

  const css = `
body {
  font-family: Arial, sans-serif;
  margin: 40px;
  background-color: #f5f5f5;
}

h1 {
  color: #333;
  text-align: center;
}

p {
  text-align: center;
  font-size: 18px;
}
`;
```

```
const js = `
console.log('Hello from Node.js file server!');
document.addEventListener('DOMContentLoaded', function() {
  console.log('Page loaded successfully!');
});
`;

fs.writeFileSync(path.join(publicDir, "index.html"), indexHtml);
fs.writeFileSync(path.join(publicDir, "styles.css"), css);
fs.writeFileSync(path.join(publicDir, "script.js"), js);
}

const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`🚀 File server running on http://localhost:${PORT}`);
  console.log(`📁 Serving files from: ${publicDir}`);
});
```

## Real-World Use Case

### RESTful API Server with Logging

```
// api-server.js
const http = require("http");
const url = require("url");
const fs = require("fs");
const path = require("path");

// Simple logging middleware
class Logger {
  static log(req, res, startTime) {
    const duration = Date.now() - startTime;
    const logEntry = {
      timestamp: new Date().toISOString(),
      method: req.method,
      url: req.url,
      statusCode: res.statusCode,
      duration: `${duration}ms`,
      userAgent: req.headers["user-agent"] || "Unknown",
    };
  }

  console.log(
    `${logEntry.method} ${logEntry.url} ${logEntry.statusCode} ${logEntry.duration}`
  );

  // Write to log file
  const logLine = JSON.stringify(logEntry) + "\n";
  fs.appendFile("access.log", logLine, (err) => {
    if (err) console.error("Failed to write log:", err);
  });
}
```

```
    }  
  }  
  
  // Simple router  
  class Router {  
    constructor() {  
      this.routes = new Map();  
    }  
  
    addRoute(method, path, handler) {  
      const key = `${method.toUpperCase()}:${path}`;  
      this.routes.set(key, handler);  
    }  
  
    get(path, handler) {  
      this.addRoute("GET", path, handler);  
    }  
  
    post(path, handler) {  
      this.addRoute("POST", path, handler);  
    }  
  
    put(path, handler) {  
      this.addRoute("PUT", path, handler);  
    }  
  
    delete(path, handler) {  
      this.addRoute("DELETE", path, handler);  
    }  
  
    handle(req, res) {  
      const parsedUrl = url.parse(req.url, true);  
      const key = `${req.method}:${parsedUrl.pathname}`;  
  
      const handler = this.routes.get(key);  
      if (handler) {  
        handler(req, res, parsedUrl.query);  
      } else {  
        // Try pattern matching for dynamic routes  
        const dynamicHandler = this.findDynamicRoute(  
          req.method,  
          parsedUrl.pathname  
        );  
        if (dynamicHandler) {  
          dynamicHandler.handler(req, res, dynamicHandler.params);  
        } else {  
          this.send404(res);  
        }  
      }  
    }  
  }  
  
  findDynamicRoute(method, pathname) {  
    for (const [key, handler] of this.routes) {  
      const [routeMethod, routePath] = key.split(":");
```

```
    if (routeMethod !== method) continue;

    const routeParts = routePath.split("/");
    const pathParts = pathname.split("/");

    if (routeParts.length !== pathParts.length) continue;

    const params = {};
    let matches = true;

    for (let i = 0; i < routeParts.length; i++) {
      if (routeParts[i].startsWith(":")) {
        const paramName = routeParts[i].slice(1);
        params[paramName] = pathParts[i];
      } else if (routeParts[i] !== pathParts[i]) {
        matches = false;
        break;
      }
    }

    if (matches) {
      return { handler, params };
    }
  }

  return null;
}

send404(res) {
  res.writeHead(404, { "Content-Type": "application/json" });
  res.end(JSON.stringify({ error: "Route not found" }));
}

// Response helpers
class Response {
  static json(res, data, statusCode = 200) {
    res.writeHead(statusCode, { "Content-Type": "application/json" });
    res.end(JSON.stringify(data, null, 2));
  }

  static error(res, message, statusCode = 500) {
    Response.json(res, { error: message }, statusCode);
  }
}

// Request body parser
class RequestParser {
  static parseBody(req) {
    return new Promise((resolve, reject) => {
      let body = "";

      req.on("data", (chunk) => {
```

```
        body += chunk.toString();
    });

    req.on("end", () => {
        try {
            const data = body ? JSON.parse(body) : {};
            resolve(data);
        } catch (error) {
            reject(new Error("Invalid JSON"));
        }
    });

    req.on("error", reject);
});
}
}

// Data store (in production, use a real database)
let products = [
    { id: 1, name: "Laptop", price: 999.99, category: "Electronics" },
    { id: 2, name: "Book", price: 19.99, category: "Education" },
    { id: 3, name: "Coffee Mug", price: 9.99, category: "Kitchen" },
];
let nextId = 4;

// Create router and define routes
const router = new Router();

// Health check
router.get("/health", (req, res) => {
    Response.json(res, {
        status: "OK",
        timestamp: new Date().toISOString(),
        uptime: process.uptime(),
    });
});

// Get all products
router.get("/api/products", (req, res, query) => {
    let filteredProducts = products;

    // Filter by category
    if (query.category) {
        filteredProducts = products.filter(
            (p) => p.category.toLowerCase() === query.category.toLowerCase()
        );
    }

    // Filter by price range
    if (query.minPrice) {
        const minPrice = parseFloat(query.minPrice);
        filteredProducts = filteredProducts.filter((p) => p.price >= minPrice);
    }
});
```



```
    if (query.maxPrice) {
      const maxPrice = parseFloat(query.maxPrice);
      filteredProducts = filteredProducts.filter((p) => p.price <= maxPrice);
    }

    Response.json(res, {
      products: filteredProducts,
      count: filteredProducts.length,
    });
  });

// Get product by ID
router.get("/api/products/:id", (req, res, params) => {
  const id = parseInt(params.id);
  const product = products.find((p) => p.id === id);

  if (!product) {
    Response.error(res, "Product not found", 404);
    return;
  }

  Response.json(res, { product });
});

// Create product
router.post("/api/products", async (req, res) => {
  try {
    const data = await RequestParser.parseBody(req);

    // Validate required fields
    if (!data.name || !data.price) {
      Response.error(res, "Name and price are required", 400);
      return;
    }

    const newProduct = {
      id: nextId++,
      name: data.name,
      price: parseFloat(data.price),
      category: data.category || "Uncategorized",
    };

    products.push(newProduct);

    Response.json(
      res,
      {
        message: "Product created successfully",
        product: newProduct,
      },
      201
    );
  } catch (error) {
    Response.error(res, error.message, 400);
  }
});
```

```
    }
  });

  // Update product
  router.put("/api/products/:id", async (req, res, params) => {
    try {
      const id = parseInt(params.id);
      const productIndex = products.findIndex((p) => p.id === id);

      if (productIndex === -1) {
        Response.error(res, "Product not found", 404);
        return;
      }

      const data = await RequestParser.parseBody(req);

      // Update product
      products[productIndex] = {
        ...products[productIndex],
        ...data,
        id, // Ensure ID doesn't change
      };

      Response.json(res, {
        message: "Product updated successfully",
        product: products[productIndex],
      });
    } catch (error) {
      Response.error(res, error.message, 400);
    }
  });

  // Delete product
  router.delete("/api/products/:id", (req, res, params) => {
    const id = parseInt(params.id);
    const productIndex = products.findIndex((p) => p.id === id);

    if (productIndex === -1) {
      Response.error(res, "Product not found", 404);
      return;
    }

    const deletedProduct = products.splice(productIndex, 1)[0];

    Response.json(res, {
      message: "Product deleted successfully",
      product: deletedProduct,
    });
  });

  // Create server
  const server = http.createServer((req, res) => {
    const startTime = Date.now();
```

```
// Set CORS headers
res.setHeader("Access-Control-Allow-Origin", "*");
res.setHeader(
  "Access-Control-Allow-Methods",
  "GET, POST, PUT, DELETE, OPTIONS"
);
res.setHeader("Access-Control-Allow-Headers", "Content-Type");

// Handle preflight requests
if (req.method === "OPTIONS") {
  res.writeHead(200);
  res.end();
  return;
}

// Handle request
router.handle(req, res);

// Log request after response is sent
res.on("finish", () => {
  Logger.log(req, res, startTime);
});
});

const PORT = process.env.PORT || 3000;
server.listen(PORT, () => {
  console.log(`🚀 API Server running on http://localhost:${PORT}`);
  console.log("📋 Available endpoints:");
  console.log("  GET    /health");
  console.log("  GET    /api/products");
  console.log("  GET    /api/products/:id");
  console.log("  POST   /api/products");
  console.log("  PUT    /api/products/:id");
  console.log("  DELETE /api/products/:id");
});
```

## Best Practices

### 1. Always Handle Errors

```
server.on("error", (error) => {
  console.error("Server error:", error);
});

server.on("clientError", (err, socket) => {
  socket.end("HTTP/1.1 400 Bad Request\r\n\r\n");
});
```

### 2. Set Appropriate Headers

```
// Security headers
res.setHeader("X-Content-Type-Options", "nosniff");
res.setHeader("X-Frame-Options", "DENY");
res.setHeader("X-XSS-Protection", "1; mode=block");

// CORS headers
res.setHeader("Access-Control-Allow-Origin", "*");
res.setHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
```

### 3. Validate Input

```
function validateProductData(data) {
  const errors = [];

  if (!data.name || typeof data.name !== "string") {
    errors.push("Name is required and must be a string");
  }

  if (!data.price || isNaN(parseFloat(data.price))) {
    errors.push("Price is required and must be a number");
  }

  return errors;
}
```

### 4. Use Streams for Large Data

```
// For large responses, use streams
const readStream = fs.createReadStream("large-file.json");
res.setHeader("Content-Type", "application/json");
readStream.pipe(res);
```

### 5. Implement Graceful Shutdown

```
process.on("SIGTERM", () => {
  console.log("Received SIGTERM, shutting down gracefully");
  server.close(() => {
    console.log("Server closed");
    process.exit(0);
  });
});
```

## Summary

Creating HTTP servers with Node.js involves:

- **Basic Server:** Using `http.createServer()` to handle requests and responses
- **Routing:** Parsing URLs and handling different endpoints
- **HTTP Methods:** Supporting GET, POST, PUT, DELETE operations
- **Request Parsing:** Handling request bodies and query parameters
- **Response Formatting:** Sending JSON, HTML, and file responses
- **Error Handling:** Proper error responses and server error handling
- **Security:** Setting appropriate headers and validating input

Key concepts:

- Always handle errors gracefully
- Set appropriate HTTP status codes
- Use streams for large data
- Implement proper CORS handling
- Log requests for debugging and monitoring

While building HTTP servers from scratch is educational, in production you'll typically use Express.js, which provides a much more convenient and feature-rich framework. Next, we'll explore Express.js and see how it simplifies web server development.

# Introduction to Express.js: Web Framework for Node.js

---

## Overview

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It simplifies the process of building web servers and APIs by providing a thin layer of fundamental web application features, without obscuring Node.js features you know and love.

## Key Concepts

What is Express.js?

Express.js is:

- A **web framework** for Node.js
- **Minimalist** and **unopinionated**
- Built on top of Node.js HTTP module
- Provides **routing**, **middleware**, and **templating** capabilities
- The most popular Node.js framework

## Core Features

- **Routing:** Define routes for different HTTP methods and URLs
- **Middleware:** Functions that execute during the request-response cycle
- **Template Engines:** Support for various view engines (EJS, Pug, Handlebars)
- **Static Files:** Serve static assets like CSS, JavaScript, images
- **Error Handling:** Built-in error handling mechanisms

## Express vs Raw Node.js

Feature	Raw Node.js	Express.js
Setup	Complex	Simple
Routing	Manual parsing	Built-in router
Middleware	Custom implementation	Rich ecosystem
Request parsing	Manual	Built-in parsers
Static files	Manual handling	<code>express.static()</code>
Template engines	Manual integration	Built-in support

## Example Code

### Installing Express

```
# Initialize a new project
npm init -y

# Install Express
npm install express

# Install development dependencies
npm install -D nodemon
```

Update `package.json` scripts:

```
{
  "scripts": {
    "start": "node app.js",
    "dev": "nodemon app.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  }
}
```

### Basic Express Server

```
// app.js
const express = require("express");

// Create Express application
const app = express();
const PORT = process.env.PORT || 3000;

// Basic route
```

```
app.get("/", (req, res) => {
  res.send("Hello, Express.js!");
});

// JSON response
app.get("/api/status", (req, res) => {
  res.json({
    status: "OK",
    message: "Express server is running",
    timestamp: new Date().toISOString(),
  });
});

// Start server
app.listen(PORT, () => {
  console.log(`🚀 Express server running on http://localhost:${PORT}`);
});
```

## Basic Routing

```
// routes-basic.js
const express = require("express");
const app = express();

// GET route
app.get("/", (req, res) => {
  res.send("<h1>Welcome to Express!</h1>");
});

// POST route
app.post("/users", (req, res) => {
  res.json({ message: "User created" });
});

// PUT route
app.put("/users/:id", (req, res) => {
  const userId = req.params.id;
  res.json({ message: `User ${userId} updated` });
});

// DELETE route
app.delete("/users/:id", (req, res) => {
  const userId = req.params.id;
  res.json({ message: `User ${userId} deleted` });
});

// Route with multiple HTTP methods
app
  .route("/products")
  .get((req, res) => {
    res.json({ message: "Get all products" });
  });
```

```
    })
    .post((req, res) => {
      res.json({ message: "Create product" });
    });

// Route parameters
app.get("/users/:id", (req, res) => {
  const userId = req.params.id;
  res.json({
    message: `Getting user ${userId}`,
    params: req.params,
  });
});

// Multiple route parameters
app.get("/users/:userId/posts/:postId", (req, res) => {
  const { userId, postId } = req.params;
  res.json({
    message: `Getting post ${postId} from user ${userId}`,
    params: req.params,
  });
});

// Query parameters
app.get("/search", (req, res) => {
  const { q, page, limit } = req.query;
  res.json({
    message: "Search results",
    query: q,
    page: page || 1,
    limit: limit || 10,
    allQuery: req.query,
  });
});

// Wildcard routes
app.get("/files/*", (req, res) => {
  const filePath = req.params[0];
  res.json({
    message: `Accessing file: ${filePath}`,
    fullPath: req.path,
  });
});

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

## Middleware Basics



```
// middleware-basics.js
const express = require("express");
const app = express();

// Built-in middleware for parsing JSON
app.use(express.json());

// Built-in middleware for parsing URL-encoded data
app.use(express.urlencoded({ extended: true }));

// Custom logging middleware
app.use((req, res, next) => {
  const timestamp = new Date().toISOString();
  console.log(`[${timestamp}] ${req.method} ${req.path}`);
  next(); // Pass control to the next middleware
});

// Authentication middleware (example)
const requireAuth = (req, res, next) => {
  const authHeader = req.headers.authorization;

  if (!authHeader) {
    return res.status(401).json({ error: "Authorization header required" });
  }

  // Simple token check (in real app, verify JWT)
  if (authHeader !== "Bearer valid-token") {
    return res.status(401).json({ error: "Invalid token" });
  }

  // Add user info to request object
  req.user = { id: 1, name: "John Doe" };
  next();
};

// Middleware for specific routes
app.use("/api/protected", requireAuth);

// Request timing middleware
app.use((req, res, next) => {
  req.startTime = Date.now();

  // Override res.json to add timing
  const originalJson = res.json;
  res.json = function (data) {
    const duration = Date.now() - req.startTime;
    console.log(`Request completed in ${duration}ms`);
    return originalJson.call(this, data);
  };

  next();
});
```

```
// Routes
app.get("/", (req, res) => {
  res.json({ message: "Public endpoint" });
});

app.get("/api/protected/profile", (req, res) => {
  res.json({
    message: "Protected endpoint",
    user: req.user,
  });
});

app.post("/api/data", (req, res) => {
  console.log("Request body:", req.body);
  res.json({
    message: "Data received",
    received: req.body,
  });
});

// Error handling middleware (must be last)
app.use((err, req, res, next) => {
  console.error("Error:", err.message);
  res.status(500).json({
    error: "Internal server error",
    message: err.message,
  });
});

// 404 handler (must be after all routes)
app.use("*", (req, res) => {
  res.status(404).json({
    error: "Route not found",
    path: req.path,
    method: req.method,
  });
});

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

## Complete CRUD API Example

```
// crud-api.js
const express = require("express");
const app = express();

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

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

// Logging middleware
app.use((req, res, next) => {
  console.log(`${new Date().toISOString()} - ${req.method} ${req.path}`);
  next();
});

// In-memory data store (use database in production)
let books = [
  {
    id: 1,
    title: "The Great Gatsby",
    author: "F. Scott Fitzgerald",
    year: 1925,
  },
  { id: 2, title: "To Kill a Mockingbird", author: "Harper Lee", year: 1960 },
  { id: 3, title: "1984", author: "George Orwell", year: 1949 },
];

let nextId = 4;

// Validation middleware
const validateBook = (req, res, next) => {
  const { title, author, year } = req.body;
  const errors = [];

  if (!title || typeof title !== "string" || title.trim().length === 0) {
    errors.push("Title is required and must be a non-empty string");
  }

  if (!author || typeof author !== "string" || author.trim().length === 0) {
    errors.push("Author is required and must be a non-empty string");
  }

  if (
    !year ||
    !Number.isInteger(year) ||
    year < 0 ||
    year > new Date().getFullYear()
  ) {
    errors.push("Year must be a valid integer between 0 and current year");
  }

  if (errors.length > 0) {
    return res.status(400).json({
      success: false,
      errors,
    });
  }

  next();
};
```

```
// Routes

// Home page
app.get("/", (req, res) => {
  res.send(`
    <h1>📖 Books API</h1>
    <p>Welcome to the Books API built with Express.js!</p>
    <h2>Available Endpoints:</h2>
    <ul>
      <li><strong>GET</strong> /api/books - Get all books</li>
      <li><strong>GET</strong> /api/books/:id - Get book by ID</li>
      <li><strong>POST</strong> /api/books - Create a new book</li>
      <li><strong>PUT</strong> /api/books/:id - Update book by ID</li>
      <li><strong>DELETE</strong> /api/books/:id - Delete book by ID</li>
    </ul>
    <h2>Example Usage:</h2>
    <pre>
# Get all books
curl http://localhost:3000/api/books

# Create a book
curl -X POST http://localhost:3000/api/books \
  -H "Content-Type: application/json" \
  -d '{"title":"Brave New World","author":"Aldous Huxley","year":1932}'
    </pre>
  `);
});

// GET /api/books - Get all books
app.get("/api/books", (req, res) => {
  const { author, year, search } = req.query;
  let filteredBooks = books;

  // Filter by author
  if (author) {
    filteredBooks = filteredBooks.filter((book) =>
      book.author.toLowerCase().includes(author.toLowerCase())
    );
  }

  // Filter by year
  if (year) {
    filteredBooks = filteredBooks.filter(
      (book) => book.year === parseInt(year)
    );
  }

  // Search in title or author
  if (search) {
    filteredBooks = filteredBooks.filter(
      (book) =>
        book.title.toLowerCase().includes(search.toLowerCase()) ||
        book.author.toLowerCase().includes(search.toLowerCase())
    );
  }
});
```

```
}

res.json({
  success: true,
  count: filteredBooks.length,
  data: filteredBooks,
});
});

// GET /api/books/:id - Get book by ID
app.get("/api/books/:id", (req, res) => {
  const id = parseInt(req.params.id);

  if (isNaN(id)) {
    return res.status(400).json({
      success: false,
      error: "Invalid book ID",
    });
  }

  const book = books.find((b) => b.id === id);

  if (!book) {
    return res.status(404).json({
      success: false,
      error: "Book not found",
    });
  }

  res.json({
    success: true,
    data: book,
  });
});

// POST /api/books - Create new book
app.post("/api/books", validateBook, (req, res) => {
  const { title, author, year } = req.body;

  // Check if book already exists
  const existingBook = books.find(
    (b) =>
      b.title.toLowerCase() === title.toLowerCase() &&
      b.author.toLowerCase() === author.toLowerCase()
  );

  if (existingBook) {
    return res.status(409).json({
      success: false,
      error: "Book with this title and author already exists",
    });
  }

  const newBook = {
```

```
    id: nextId++,
    title: title.trim(),
    author: author.trim(),
    year,
  });

  books.push(newBook);

  res.status(201).json({
    success: true,
    message: "Book created successfully",
    data: newBook,
  });
});

// PUT /api/books/:id - Update book
app.put("/api/books/:id", validateBook, (req, res) => {
  const id = parseInt(req.params.id);

  if (isNaN(id)) {
    return res.status(400).json({
      success: false,
      error: "Invalid book ID",
    });
  }

  const bookIndex = books.findIndex((b) => b.id === id);

  if (bookIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "Book not found",
    });
  }

  const { title, author, year } = req.body;

  // Update book
  books[bookIndex] = {
    id,
    title: title.trim(),
    author: author.trim(),
    year,
  };

  res.json({
    success: true,
    message: "Book updated successfully",
    data: books[bookIndex],
  });
});

// DELETE /api/books/:id - Delete book
app.delete("/api/books/:id", (req, res) => {
```

```
const id = parseInt(req.params.id);

if (isNaN(id)) {
  return res.status(400).json({
    success: false,
    error: "Invalid book ID",
  });
}

const bookIndex = books.findIndex((b) => b.id === id);

if (bookIndex === -1) {
  return res.status(404).json({
    success: false,
    error: "Book not found",
  });
}

const deletedBook = books.splice(bookIndex, 1)[0];

res.json({
  success: true,
  message: "Book deleted successfully",
  data: deletedBook,
});
});

// Health check endpoint
app.get("/health", (req, res) => {
  res.json({
    status: "OK",
    timestamp: new Date().toISOString(),
    uptime: process.uptime(),
    memory: process.memoryUsage(),
    booksCount: books.length,
  });
});

// Error handling middleware
app.use((err, req, res, next) => {
  console.error("Error:", err.stack);
  res.status(500).json({
    success: false,
    error: "Internal server error",
  });
});

// 404 handler
app.use("*", (req, res) => {
  res.status(404).json({
    success: false,
    error: "Endpoint not found",
    path: req.originalUrl,
    method: req.method,
  });
});
```

```
});  
});  
  
const PORT = process.env.PORT || 3000;  
app.listen(PORT, () => {  
  console.log(`🚀 Books API server running on http://localhost:${PORT}`);  
  console.log(`📖 API documentation available at http://localhost:${PORT}`);  
});
```

## Real-World Use Case

### Express Server with Environment Configuration

```
// server.js  
const express = require("express");  
const path = require("path");  
  
// Environment configuration  
const config = {  
  port: process.env.PORT || 3000,  
  nodeEnv: process.env.NODE_ENV || "development",  
  apiPrefix: process.env.API_PREFIX || "/api/v1",  
};  
  
const app = express();  
  
// Middleware setup based on environment  
if (config.nodeEnv === "development") {  
  // Development-specific middleware  
  app.use((req, res, next) => {  
    console.log(`[DEV] ${req.method} ${req.path}`);  
    next();  
  });  
}  
  
// Global middleware  
app.use(express.json({ limit: "10mb" }));  
app.use(express.urlencoded({ extended: true, limit: "10mb" }));  
  
// Security headers  
app.use((req, res, next) => {  
  res.setHeader("X-Content-Type-Options", "nosniff");  
  res.setHeader("X-Frame-Options", "DENY");  
  res.setHeader("X-XSS-Protection", "1; mode=block");  
  next();  
});  
  
// API routes  
const apiRouter = express.Router();  
  
apiRouter.get("/status", (req, res) => {
```



```
    res.json({
      status: "OK",
      environment: config.nodeEnv,
      timestamp: new Date().toISOString(),
      version: "1.0.0",
    });
  });
});

apiRouter.get("/info", (req, res) => {
  res.json({
    app: "Express.js API",
    version: "1.0.0",
    environment: config.nodeEnv,
    node: process.version,
    platform: process.platform,
    uptime: process.uptime(),
  });
});

// Mount API routes
app.use(config.apiPrefix, apiRouter);

// Serve static files in production
if (config.nodeEnv === "production") {
  app.use(express.static(path.join(__dirname, "public")));

  // Catch-all handler for SPA
  app.get("*", (req, res) => {
    res.sendFile(path.join(__dirname, "public", "index.html"));
  });
}

// Development route
if (config.nodeEnv === "development") {
  app.get("/", (req, res) => {
    res.send(`
      <h1>Express.js Development Server</h1>
      <p>Environment: ${config.nodeEnv}</p>
      <p>API Base URL: <a href="${config.apiPrefix}">${config.apiPrefix}</a>
    </p>
      <ul>
        <li><a href="${config.apiPrefix}/status">Status</a></li>
        <li><a href="${config.apiPrefix}/info">Info</a></li>
      </ul>
    `);
  });
}

// Error handling
app.use((err, req, res, next) => {
  console.error("Error:", err.stack);

  const errorResponse = {
    error: "Internal server error",
  };
```

```
};

// Include stack trace in development
if (config.nodeEnv === "development") {
  errorResponse.stack = err.stack;
}

res.status(500).json(errorResponse);
});

// 404 handler
app.use("*", (req, res) => {
  res.status(404).json({
    error: "Not found",
    path: req.originalUrl,
  });
});

// Graceful shutdown
process.on("SIGTERM", () => {
  console.log("Received SIGTERM, shutting down gracefully");
  server.close(() => {
    console.log("Server closed");
    process.exit(0);
  });
});

const server = app.listen(config.port, () => {
  console.log(`🚀 Express server running on http://localhost:${config.port}`);
  console.log(`📁 Environment: ${config.nodeEnv}`);
  console.log(`🔗 API Base URL: ${config.apiPrefix}`);
});

module.exports = app;
```

## Best Practices

### 1. Project Structure

```
project/
├── app.js           # Main application file
├── routes/          # Route definitions
│   ├── index.js
│   ├── users.js
│   └── products.js
├── middleware/      # Custom middleware
│   ├── auth.js
│   └── validation.js
├── controllers/     # Route handlers
│   ├── userController.js
│   └── productController.js
```

```
|— models/          # Data models
|— config/          # Configuration files
|— public/          # Static files
|— views/           # Template files
```

## 2. Environment Variables

```
// Use environment variables for configuration
const config = {
  port: process.env.PORT || 3000,
  dbUrl: process.env.DATABASE_URL || "mongodb://localhost:27017/myapp",
  jwtSecret: process.env.JWT_SECRET || "fallback-secret",
  nodeEnv: process.env.NODE_ENV || "development",
};
```

## 3. Error Handling

```
// Async error wrapper
const asyncHandler = (fn) => (req, res, next) => {
  Promise.resolve(fn(req, res, next)).catch(next);
};

// Usage
app.get(
  "/api/users",
  asyncHandler(async (req, res) => {
    const users = await User.find();
    res.json(users);
  })
);
```

## 4. Input Validation

```
// Validation middleware
const validateUser = (req, res, next) => {
  const { name, email } = req.body;

  if (!name || name.length < 2) {
    return res
      .status(400)
      .json({ error: "Name must be at least 2 characters" });
  }

  if (!email || !email.includes("@")) {
    return res.status(400).json({ error: "Valid email is required" });
  }
};
```

```
    next();  
  };
```

## 5. Security Headers

```
// Security middleware  
app.use((req, res, next) => {  
  res.setHeader("X-Content-Type-Options", "nosniff");  
  res.setHeader("X-Frame-Options", "DENY");  
  res.setHeader("X-XSS-Protection", "1; mode=block");  
  res.setHeader("Strict-Transport-Security", "max-age=31536000");  
  next();  
});
```

## Summary

Express.js simplifies Node.js web development by providing:

- **Simple Setup:** Minimal boilerplate code to get started
- **Powerful Routing:** Easy route definition with parameters and query strings
- **Middleware System:** Modular request processing pipeline
- **Built-in Parsers:** JSON and URL-encoded body parsing
- **Static File Serving:** Easy static asset serving
- **Error Handling:** Centralized error handling mechanisms

Key advantages over raw Node.js:

- Less boilerplate code
- Better organization and structure
- Rich ecosystem of middleware
- Simplified request/response handling
- Built-in routing system

Express.js strikes the perfect balance between simplicity and functionality, making it the go-to choice for Node.js web applications. Next, we'll dive deeper into handling request and response objects, exploring all the methods and properties available for building robust web applications.

# Handling Request and Response Objects in Express.js

---

## Overview

The request (**req**) and response (**res**) objects are the heart of Express.js applications. They provide access to HTTP request data and methods to send responses back to clients. Understanding these objects thoroughly is crucial for building robust web applications and APIs.

## Key Concepts

## Request Object (req)

The request object represents the HTTP request and contains properties for:

- Request parameters, query strings, and body
- HTTP headers and cookies
- Request method and URL information
- User agent and IP address
- File uploads and form data

## Response Object (res)

The response object represents the HTTP response and provides methods to:

- Send different types of responses (JSON, HTML, files)
- Set HTTP status codes and headers
- Handle cookies and redirects
- Stream data and handle errors

## Request-Response Cycle

1. Client sends HTTP request
2. Express creates `req` and `res` objects
3. Middleware and route handlers process the request
4. Response is sent back to client
5. Connection is closed (unless keep-alive)

## Example Code

### Request Object Properties and Methods

```
// request-examples.js
const express = require("express");
const app = express();

// Middleware to parse JSON and URL-encoded data
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Comprehensive request information endpoint
app.all("/request-info", (req, res) => {
  const requestInfo = {
    // Basic request information
    method: req.method,
    url: req.url,
    originalUrl: req.originalUrl,
    path: req.path,
    protocol: req.protocol,
    secure: req.secure,

    // Headers
```

```
headers: req.headers,
userAgent: req.get("User-Agent"),
contentType: req.get("Content-Type"),
authorization: req.get("Authorization"),

// Network information
ip: req.ip,
ips: req.ips,
hostname: req.hostname,

// Parameters and query
params: req.params,
query: req.query,
body: req.body,

// Request properties
fresh: req.fresh,
stale: req.stale,
xhr: req.xhr, // Is AJAX request

// Cookies (if cookie-parser middleware is used)
cookies: req.cookies || "No cookie parser middleware",
signedCookies: req.signedCookies || "No cookie parser middleware",
};

res.json(requestInfo);
});

// Route parameters examples
app.get("/users/:userId", (req, res) => {
  const userId = req.params.userId;

  res.json({
    message: `Getting user ${userId}`,
    params: req.params,
    paramType: typeof userId, // Always string
  });
});

// Multiple parameters
app.get("/users/:userId/posts/:postId", (req, res) => {
  const { userId, postId } = req.params;

  res.json({
    message: `Getting post ${postId} from user ${userId}`,
    userId: parseInt(userId), // Convert to number if needed
    postId: parseInt(postId),
    allParams: req.params,
  });
});

// Query parameters
app.get("/search", (req, res) => {
  const {
```

```
    q, // Search query
    page = 1, // Default value
    limit = 10, // Default value
    sort, // Optional
    filter, // Optional
  } = req.query;

// Convert string parameters to appropriate types
const pageNum = parseInt(page);
const limitNum = parseInt(limit);

// Validate parameters
if (isNaN(pageNum) || pageNum < 1) {
  return res.status(400).json({ error: "Page must be a positive integer" });
}

if (isNaN(limitNum) || limitNum < 1 || limitNum > 100) {
  return res.status(400).json({ error: "Limit must be between 1 and 100" });
}

res.json({
  searchQuery: q,
  pagination: {
    page: pageNum,
    limit: limitNum,
    offset: (pageNum - 1) * limitNum,
  },
  sorting: sort,
  filters: filter,
  allQuery: req.query,
});
});

// Request body handling
app.post("/data", (req, res) => {
  console.log("Request body:", req.body);
  console.log("Content-Type:", req.get("Content-Type"));

  // Validate request body
  if (!req.body || Object.keys(req.body).length === 0) {
    return res.status(400).json({ error: "Request body is required" });
  }

  res.json({
    message: "Data received successfully",
    receivedData: req.body,
    dataType: typeof req.body,
    contentLength: req.get("Content-Length"),
  });
});

// Header inspection
app.get("/headers", (req, res) => {
  const importantHeaders = {
```

```

    userAgent: req.get("User-Agent"),
    accept: req.get("Accept"),
    acceptLanguage: req.get("Accept-Language"),
    acceptEncoding: req.get("Accept-Encoding"),
    connection: req.get("Connection"),
    host: req.get("Host"),
    referer: req.get("Referer"),
    origin: req.get("Origin"),
  };

  res.json({
    importantHeaders,
    allHeaders: req.headers,
    headerCount: Object.keys(req.headers).length,
  });
});

// Content negotiation
app.get("/content-negotiation", (req, res) => {
  const acceptsJson = req.accepts("json");
  const acceptsHtml = req.accepts("html");
  const acceptsXml = req.accepts("xml");

  res.json({
    clientAccepts: {
      json: !!acceptsJson,
      html: !!acceptsHtml,
      xml: !!acceptsXml,
    },
    acceptHeader: req.get("Accept"),
    preferredType: req.accepts(["json", "html", "xml"]),
  });
});

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

```

## Response Object Methods

```

// response-examples.js
const express = require("express");
const path = require("path");
const fs = require("fs");
const app = express();

app.use(express.json());

// Basic response methods
app.get("/response-types", (req, res) => {

```



```
const type = req.query.type;

switch (type) {
  case "json":
    res.json({ message: "JSON response", timestamp: new Date() });
    break;

  case "text":
    res.send("Plain text response");
    break;

  case "html":
    res.send("<h1>HTML Response</h1><p>This is HTML content</p>");
    break;

  case "status":
    res.status(201).json({ message: "Created successfully" });
    break;

  default:
    res.json({
      message: "Specify type parameter",
      availableTypes: ["json", "text", "html", "status"],
    });
}
});

// Status codes
app.get("/status/:code", (req, res) => {
  const statusCode = parseInt(req.params.code);

  if (isNaN(statusCode) || statusCode < 100 || statusCode > 599) {
    return res.status(400).json({ error: "Invalid status code" });
  }

  res.status(statusCode).json({
    statusCode,
    message: getStatusMessage(statusCode),
  });
});

function getStatusMessage(code) {
  const messages = {
    200: "OK",
    201: "Created",
    400: "Bad Request",
    401: "Unauthorized",
    403: "Forbidden",
    404: "Not Found",
    500: "Internal Server Error",
  };
  return messages[code] || "Unknown Status";
}
```

```
// Headers manipulation
app.get("/headers-demo", (req, res) => {
  // Set individual headers
  res.set("X-Custom-Header", "Custom Value");
  res.set("X-API-Version", "1.0.0");

  // Set multiple headers at once
  res.set({
    "X-Response-Time": Date.now(),
    "X-Server": "Express.js",
    "Cache-Control": "no-cache",
  });

  // Get header value
  const customHeader = res.get("X-Custom-Header");

  res.json({
    message: "Headers set successfully",
    customHeaderValue: customHeader,
    allHeaders: res.getHeaders(),
  });
});

// Content-Type examples
app.get("/content-types/:type", (req, res) => {
  const type = req.params.type;

  switch (type) {
    case "json":
      res.type("json").send(JSON.stringify({ message: "JSON content" }));
      break;

    case "xml":
      res
        .type("xml")
        .send('<?xml version="1.0"?><message>XML content</message>');
      break;

    case "plain":
      res.type("txt").send("Plain text content");
      break;

    case "html":
      res.type("html").send("<h1>HTML Content</h1>");
      break;

    default:
      res.json({ error: "Unknown content type" });
  }
});

// Redirects
app.get("/redirect-demo/:type", (req, res) => {
  const type = req.params.type;
```

```
switch (type) {
  case "permanent":
    res.redirect(301, "/redirected");
    break;

  case "temporary":
    res.redirect(302, "/redirected");
    break;

  case "back":
    res.redirect("back"); // Redirect to referer or '/'
    break;

  default:
    res.redirect("/redirected"); // Default 302
}
});

app.get("/redirected", (req, res) => {
  res.json({ message: "You have been redirected!" });
});

// File downloads
app.get("/download/:filename", (req, res) => {
  const filename = req.params.filename;
  const filePath = path.join(__dirname, "downloads", filename);

  // Check if file exists
  if (!fs.existsSync(filePath)) {
    return res.status(404).json({ error: "File not found" });
  }

  // Download with original filename
  res.download(filePath, (err) => {
    if (err) {
      console.error("Download error:", err);
      if (!res.headersSent) {
        res.status(500).json({ error: "Download failed" });
      }
    }
  });
});

// File sending
app.get("/file/:filename", (req, res) => {
  const filename = req.params.filename;
  const filePath = path.join(__dirname, "files", filename);

  res.sendFile(filePath, (err) => {
    if (err) {
      console.error("Send file error:", err);
      if (!res.headersSent) {
        res.status(404).json({ error: "File not found" });
      }
    }
  });
});
```

```
    }
  }
});
});

// Streaming responses
app.get("/stream", (req, res) => {
  res.writeHead(200, {
    "Content-Type": "text/plain",
    "Transfer-Encoding": "chunked",
  });

  let counter = 0;
  const interval = setInterval(() => {
    counter++;
    res.write(`Chunk ${counter}: ${new Date().toISOString()}\n`);

    if (counter >= 10) {
      clearInterval(interval);
      res.end("Stream completed\n");
    }
  }, 1000);

  // Handle client disconnect
  req.on("close", () => {
    clearInterval(interval);
    console.log("Client disconnected");
  });
});

// JSON responses with different structures
app.get("/api/response-formats/:format", (req, res) => {
  const format = req.params.format;
  const sampleData = [
    { id: 1, name: "John Doe", email: "john@example.com" },
    { id: 2, name: "Jane Smith", email: "jane@example.com" },
  ];

  switch (format) {
    case "simple":
      res.json(sampleData);
      break;

    case "wrapped":
      res.json({
        success: true,
        data: sampleData,
        count: sampleData.length,
      });
      break;

    case "paginated":
      res.json({
        success: true,
```

```
        data: sampleData,
        pagination: {
            page: 1,
            limit: 10,
            total: sampleData.length,
            totalPages: 1,
        },
    });
    break;

    case "meta":
        res.json({
            data: sampleData,
            meta: {
                count: sampleData.length,
                timestamp: new Date().toISOString(),
                version: "1.0.0",
            },
        });
        break;

    default:
        res.status(400).json({
            error: "Invalid format",
            availableFormats: ["simple", "wrapped", "paginated", "meta"],
        });
    }
});

// Error responses
app.get("/error/:type", (req, res) => {
    const type = req.params.type;

    switch (type) {
        case "validation":
            res.status(400).json({
                error: "Validation Error",
                details: [
                    { field: "email", message: "Email is required" },
                    {
                        field: "password",
                        message: "Password must be at least 8 characters",
                    },
                ],
            });
            break;

        case "unauthorized":
            res.status(401).json({
                error: "Unauthorized",
                message: "Authentication required",
            });
            break;
    }
});
```

```
    case "forbidden":
      res.status(403).json({
        error: "Forbidden",
        message: "Insufficient permissions",
      });
      break;

    case "notfound":
      res.status(404).json({
        error: "Not Found",
        message: "Resource not found",
      });
      break;

    case "server":
      res.status(500).json({
        error: "Internal Server Error",
        message: "Something went wrong",
      });
      break;

    default:
      res.status(400).json({
        error: "Invalid error type",
        availableTypes: [
          "validation",
          "unauthorized",
          "forbidden",
          "notfound",
          "server",
        ],
      });
  }
});

// Create sample directories and files
const downloadsDir = path.join(__dirname, "downloads");
const filesDir = path.join(__dirname, "files");

if (!fs.existsSync(downloadsDir)) {
  fs.mkdirSync(downloadsDir, { recursive: true });
  fs.writeFileSync(
    path.join(downloadsDir, "sample.txt"),
    "This is a sample download file."
  );
}

if (!fs.existsSync(filesDir)) {
  fs.mkdirSync(filesDir, { recursive: true });
  fs.writeFileSync(
    path.join(filesDir, "sample.json"),
    JSON.stringify({ message: "Sample JSON file" }, null, 2)
  );
}
```

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

## Advanced Request/Response Handling

```
// advanced-req-res.js
const express = require("express");
const multer = require("multer"); // For file uploads
const app = express();

// Configure multer for file uploads
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, "uploads/");
  },
  filename: (req, file, cb) => {
    const uniqueSuffix = Date.now() + "-" + Math.round(Math.random() * 1e9);
    cb(
      null,
      file.fieldname + "-" + uniqueSuffix + path.extname(file.originalname)
    );
  },
});

const upload = multer({
  storage,
  limits: {
    fileSize: 5 * 1024 * 1024, // 5MB limit
  },
  fileFilter: (req, file, cb) => {
    // Accept only images
    if (file.mimetype.startsWith("image/")) {
      cb(null, true);
    } else {
      cb(new Error("Only image files are allowed"), false);
    }
  },
});

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

// Request validation middleware
const validateRequest = (schema) => {
  return (req, res, next) => {
    const errors = [];

    // Validate required fields
```

```
    if (schema.required) {
      schema.required.forEach((field) => {
        if (!req.body[field]) {
          errors.push(`${field} is required`);
        }
      });
    }

    // Validate field types
    if (schema.types) {
      Object.keys(schema.types).forEach((field) => {
        if (req.body[field] && typeof req.body[field] !== schema.types[field]) {
          errors.push(`${field} must be of type ${schema.types[field]}`);
        }
      });
    }

    if (errors.length > 0) {
      return res.status(400).json({
        success: false,
        errors,
      });
    }

    next();
  };
};

// Request logging middleware
const requestLogger = (req, res, next) => {
  const start = Date.now();

  // Log request
  console.log(`[${new Date().toISOString()}] ${req.method} ${req.path}`);

  // Override res.json to log response
  const originalJson = res.json;
  res.json = function (data) {
    const duration = Date.now() - start;
    console.log(`[${new Date().toISOString()}] Response sent in ${duration}ms`);
    return originalJson.call(this, data);
  };

  next();
};

app.use(requestLogger);

// File upload endpoint
app.post("/upload", upload.single("image"), (req, res) => {
  if (!req.file) {
    return res.status(400).json({
      success: false,
      error: "No file uploaded",
    });
  }
});
```



```
    });
  }

  res.json({
    success: true,
    message: "File uploaded successfully",
    file: {
      originalName: req.file.originalname,
      filename: req.file.filename,
      size: req.file.size,
      mimetype: req.file.mimetype,
      path: req.file.path,
    },
  });
});

// Multiple file upload
app.post("/upload-multiple", upload.array("images", 5), (req, res) => {
  if (!req.files || req.files.length === 0) {
    return res.status(400).json({
      success: false,
      error: "No files uploaded",
    });
  }

  const fileInfo = req.files.map((file) => ({
    originalName: file.originalname,
    filename: file.filename,
    size: file.size,
    mimetype: file.mimetype,
  }));

  res.json({
    success: true,
    message: `${req.files.length} files uploaded successfully`,
    files: fileInfo,
  });
});

// Request with validation
const userSchema = {
  required: ["name", "email"],
  types: {
    name: "string",
    email: "string",
    age: "number",
  },
};

app.post("/users", validateRequest(userSchema), (req, res) => {
  const { name, email, age } = req.body;

  // Additional validation
  if (!email.includes("@")) {
```

```
    return res.status(400).json({
      success: false,
      error: "Invalid email format",
    });
  }

  if (age && (age < 0 || age > 150)) {
    return res.status(400).json({
      success: false,
      error: "Age must be between 0 and 150",
    });
  }

  // Simulate user creation
  const newUser = {
    id: Date.now(),
    name,
    email,
    age: age || null,
    createdAt: new Date().toISOString(),
  };

  res.status(201).json({
    success: true,
    message: "User created successfully",
    user: newUser,
  });
});

// Content negotiation
app.get("/api/data", (req, res) => {
  const data = {
    id: 1,
    name: "Sample Data",
    timestamp: new Date().toISOString(),
  };

  // Check what the client accepts
  if (req.accepts("json")) {
    res.json(data);
  } else if (req.accepts("xml")) {
    const xml = `<?xml version="1.0"?>
<data>
  <id>${data.id}</id>
  <name>${data.name}</name>
  <timestamp>${data.timestamp}</timestamp>
</data>`;
    res.type("xml").send(xml);
  } else if (req.accepts("html")) {
    const html = `
    <div>
      <h2>Data</h2>
      <p><strong>ID:</strong> ${data.id}</p>
      <p><strong>Name:</strong> ${data.name}</p>
    `;
  }
});
```

```
        <p><strong>Timestamp:</strong> ${data.timestamp}</p>
      </div>`;
    res.type("html").send(html);
  } else {
    res.status(406).json({
      error: "Not Acceptable",
      supportedTypes: ["application/json", "application/xml", "text/html"],
    });
  }
});

// Response caching
app.get("/api/cached-data", (req, res) => {
  // Set cache headers
  res.set({
    "Cache-Control": "public, max-age=3600", // Cache for 1 hour
    ETag: '"123456"',
    "Last-Modified": new Date().toUTCString(),
  });

  // Check if client has cached version
  if (req.get("If-None-Match") === '"123456"') {
    return res.status(304).end(); // Not Modified
  }

  res.json({
    message: "This response is cached",
    timestamp: new Date().toISOString(),
    data: "Some expensive computation result",
  });
});

// Error handling for file uploads
app.use((error, req, res, next) => {
  if (error instanceof multer.MulterError) {
    if (error.code === "LIMIT_FILE_SIZE") {
      return res.status(400).json({
        success: false,
        error: "File too large. Maximum size is 5MB",
      });
    }
  }

  if (error.message === "Only image files are allowed") {
    return res.status(400).json({
      success: false,
      error: error.message,
    });
  }

  next(error);
});

// Create uploads directory
```

```
const fs = require("fs");
const path = require("path");
const uploadsDir = path.join(__dirname, "uploads");
if (!fs.existsSync(uploadsDir)) {
  fs.mkdirSync(uploadsDir, { recursive: true });
}

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

## Real-World Use Case

### API with Comprehensive Request/Response Handling

```
// api-server-complete.js
const express = require("express");
const app = express();

// Middleware
app.use(express.json({ limit: "10mb" }));
app.use(express.urlencoded({ extended: true, limit: "10mb" }));

// Request ID middleware
app.use((req, res, next) => {
  req.id = Math.random().toString(36).substr(2, 9);
  res.set("X-Request-ID", req.id);
  next();
});

// API response wrapper
const apiResponse = {
  success: (res, data, message = "Success", statusCode = 200) => {
    res.status(statusCode).json({
      success: true,
      message,
      data,
      timestamp: new Date().toISOString(),
      requestId: res.get("X-Request-ID"),
    });
  },
  error: (res, message = "Error", statusCode = 500, errors = null) => {
    res.status(statusCode).json({
      success: false,
      message,
      errors,
      timestamp: new Date().toISOString(),
      requestId: res.get("X-Request-ID"),
    });
  }
};
```

```
    },

    paginated: (res, data, pagination, message = "Success") => {
      res.json({
        success: true,
        message,
        data,
        pagination,
        timestamp: new Date().toISOString(),
        requestId: res.get("X-Request-ID"),
      });
    },
  },
};

// Sample data
let products = [
  {
    id: 1,
    name: "Laptop",
    price: 999.99,
    category: "Electronics",
    inStock: true,
  },
  { id: 2, name: "Book", price: 19.99, category: "Education", inStock: true },
  {
    id: 3,
    name: "Coffee Mug",
    price: 9.99,
    category: "Kitchen",
    inStock: false,
  },
];
let nextId = 4;

// Get products with filtering, sorting, and pagination
app.get("/api/products", (req, res) => {
  try {
    let {
      page = 1,
      limit = 10,
      category,
      minPrice,
      maxPrice,
      inStock,
      sort = "id",
      order = "asc",
      search,
    } = req.query;

    // Convert and validate parameters
    page = parseInt(page);
    limit = parseInt(limit);

    if (isNaN(page) || page < 1) page = 1;
```

```
if (isNaN(limit) || limit < 1 || limit > 100) limit = 10;

let filteredProducts = [...products];

// Apply filters
if (category) {
  filteredProducts = filteredProducts.filter(
    (p) => p.category.toLowerCase() === category.toLowerCase()
  );
}

if (minPrice) {
  const min = parseFloat(minPrice);
  if (!isNaN(min)) {
    filteredProducts = filteredProducts.filter((p) => p.price >= min);
  }
}

if (maxPrice) {
  const max = parseFloat(maxPrice);
  if (!isNaN(max)) {
    filteredProducts = filteredProducts.filter((p) => p.price <= max);
  }
}

if (inStock !== undefined) {
  const stockFilter = inStock === "true";
  filteredProducts = filteredProducts.filter(
    (p) => p.inStock === stockFilter
  );
}

if (search) {
  filteredProducts = filteredProducts.filter((p) =>
    p.name.toLowerCase().includes(search.toLowerCase())
  );
}

// Apply sorting
if (
  sort &&
  filteredProducts.length > 0 &&
  filteredProducts[0].hasOwnProperty(sort)
) {
  filteredProducts.sort((a, b) => {
    let aVal = a[sort];
    let bVal = b[sort];

    if (typeof aVal === "string") {
      aVal = aVal.toLowerCase();
      bVal = bVal.toLowerCase();
    }

    if (order === "desc") {
```

```
        return bVal > aVal ? 1 : -1;
    } else {
        return aVal > bVal ? 1 : -1;
    }
  });
}

// Apply pagination
const total = filteredProducts.length;
const totalPages = Math.ceil(total / limit);
const offset = (page - 1) * limit;
const paginatedProducts = filteredProducts.slice(offset, offset + limit);

const pagination = {
  page,
  limit,
  total,
  totalPages,
  hasNext: page < totalPages,
  hasPrev: page > 1,
};

apiResponse.paginated(
  res,
  paginatedProducts,
  pagination,
  "Products retrieved successfully"
);
} catch (error) {
  console.error("Error in GET /api/products:", error);
  apiResponse.error(res, "Failed to retrieve products", 500);
}
});

// Get single product
app.get("/api/products/:id", (req, res) => {
  try {
    const id = parseInt(req.params.id);

    if (isNaN(id)) {
      return apiResponse.error(res, "Invalid product ID", 400);
    }

    const product = products.find((p) => p.id === id);

    if (!product) {
      return apiResponse.error(res, "Product not found", 404);
    }

    apiResponse.success(res, product, "Product retrieved successfully");
  } catch (error) {
    console.error("Error in GET /api/products/:id:", error);
    apiResponse.error(res, "Failed to retrieve product", 500);
  }
});
```

```
});

// Create product
app.post("/api/products", (req, res) => {
  try {
    const { name, price, category, inStock = true } = req.body;

    // Validation
    const errors = [];

    if (!name || typeof name !== "string" || name.trim().length === 0) {
      errors.push("Name is required and must be a non-empty string");
    }

    if (!price || isNaN(parseFloat(price)) || parseFloat(price) <= 0) {
      errors.push("Price is required and must be a positive number");
    }

    if (
      !category ||
      typeof category !== "string" ||
      category.trim().length === 0
    ) {
      errors.push("Category is required and must be a non-empty string");
    }

    if (typeof inStock !== "boolean") {
      errors.push("inStock must be a boolean value");
    }

    if (errors.length > 0) {
      return apiResponse.error(res, "Validation failed", 400, errors);
    }

    // Check for duplicate
    const existingProduct = products.find(
      (p) =>
        p.name.toLowerCase() === name.toLowerCase() &&
        p.category.toLowerCase() === category.toLowerCase()
    );

    if (existingProduct) {
      return apiResponse.error(
        res,
        "Product with this name and category already exists",
        409
      );
    }

    // Create product
    const newProduct = {
      id: nextId++,
      name: name.trim(),
      price: parseFloat(price),
```



```
        category: category.trim(),
        inStock,
    };

    products.push(newProduct);

    apiResponse.success(res, newProduct, "Product created successfully", 201);
} catch (error) {
    console.error("Error in POST /api/products:", error);
    apiResponse.error(res, "Failed to create product", 500);
}
});

// Update product
app.put("/api/products/:id", (req, res) => {
    try {
        const id = parseInt(req.params.id);

        if (isNaN(id)) {
            return apiResponse.error(res, "Invalid product ID", 400);
        }

        const productIndex = products.findIndex((p) => p.id === id);

        if (productIndex === -1) {
            return apiResponse.error(res, "Product not found", 404);
        }

        const { name, price, category, inStock } = req.body;
        const errors = [];

        // Validate only provided fields
        if (
            name !== undefined &&
            (typeof name !== "string" || name.trim().length === 0)
        ) {
            errors.push("Name must be a non-empty string");
        }

        if (
            price !== undefined &&
            (isNaN(parseFloat(price)) || parseFloat(price) <= 0)
        ) {
            errors.push("Price must be a positive number");
        }

        if (
            category !== undefined &&
            (typeof category !== "string" || category.trim().length === 0)
        ) {
            errors.push("Category must be a non-empty string");
        }

        if (inStock !== undefined && typeof inStock !== "boolean") {

```

```
        errors.push("inStock must be a boolean value");
    }

    if (errors.length > 0) {
        return apiResponse.error(res, "Validation failed", 400, errors);
    }

    // Update product
    const updatedProduct = {
        ...products[productIndex],
        ...(name !== undefined && { name: name.trim() }),
        ...(price !== undefined && { price: parseFloat(price) }),
        ...(category !== undefined && { category: category.trim() }),
        ...(inStock !== undefined && { inStock }),
    };

    products[productIndex] = updatedProduct;

    apiResponse.success(res, updatedProduct, "Product updated successfully");
} catch (error) {
    console.error("Error in PUT /api/products/:id:", error);
    apiResponse.error(res, "Failed to update product", 500);
}
});

// Delete product
app.delete("/api/products/:id", (req, res) => {
    try {
        const id = parseInt(req.params.id);

        if (isNaN(id)) {
            return apiResponse.error(res, "Invalid product ID", 400);
        }

        const productIndex = products.findIndex((p) => p.id === id);

        if (productIndex === -1) {
            return apiResponse.error(res, "Product not found", 404);
        }

        const deletedProduct = products.splice(productIndex, 1)[0];

        apiResponse.success(res, deletedProduct, "Product deleted successfully");
    } catch (error) {
        console.error("Error in DELETE /api/products/:id:", error);
        apiResponse.error(res, "Failed to delete product", 500);
    }
});

// Global error handler
app.use((err, req, res, next) => {
    console.error("Unhandled error:", err.stack);
    apiResponse.error(res, "Internal server error", 500);
});
```

```
// 404 handler
app.use("*", (req, res) => {
  apiResponse.error(res, "Endpoint not found", 404);
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`🚀 API server running on http://localhost:${PORT}`);
});
```

## Best Practices

### 1. Always Validate Input

```
const validateInput = (req, res, next) => {
  const { email, password } = req.body;

  if (!email || !email.includes("@")) {
    return res.status(400).json({ error: "Valid email required" });
  }

  if (!password || password.length < 8) {
    return res
      .status(400)
      .json({ error: "Password must be at least 8 characters" });
  }

  next();
};
```

### 2. Use Consistent Response Format

```
const responseFormat = {
  success: (data, message = "Success") => ({
    success: true,
    message,
    data,
    timestamp: new Date().toISOString(),
  }),
  error: (message, errors = null) => ({
    success: false,
    message,
    errors,
    timestamp: new Date().toISOString(),
  }),
};
```

### 3. Handle Async Errors

```
const asyncHandler = (fn) => (req, res, next) => {
  Promise.resolve(fn(req, res, next)).catch(next);
};

app.get(
  "/api/users",
  asyncHandler(async (req, res) => {
    const users = await User.find();
    res.json(users);
  })
);
```

### 4. Set Security Headers

```
app.use((req, res, next) => {
  res.set({
    "X-Content-Type-Options": "nosniff",
    "X-Frame-Options": "DENY",
    "X-XSS-Protection": "1; mode=block",
  });
  next();
});
```

### 5. Log Requests and Responses

```
const logger = (req, res, next) => {
  const start = Date.now();

  res.on("finish", () => {
    const duration = Date.now() - start;
    console.log(`${req.method} ${req.path} ${res.statusCode} ${duration}ms`);
  });

  next();
};
```

## Summary

Mastering request and response objects in Express.js involves:

#### Request Object (req):

- Access route parameters with `req.params`

- Handle query strings with `req.query`
- Parse request body with `req.body`
- Read headers with `req.get()` or `req.headers`
- Get client information with `req.ip`, `req.hostname`

**Response Object (res):**

- Send JSON with `res.json()`
- Set status codes with `res.status()`
- Set headers with `res.set()`
- Handle redirects with `res.redirect()`
- Send files with `res.sendFile()` or `res.download()`

**Best Practices:**

- Always validate input data
- Use consistent response formats
- Handle errors gracefully
- Set appropriate HTTP status codes
- Implement proper logging
- Set security headers

Understanding these objects thoroughly enables you to build robust, secure, and user-friendly web applications and APIs. Next, we'll explore serving static files and building complete web applications with Express.js.

## Serving Static Files in Express.js

---

### Overview

Serving static files is a fundamental requirement for web applications. Static files include CSS stylesheets, JavaScript files, images, fonts, and other assets that don't change based on user requests. Express.js provides built-in middleware to serve static files efficiently, along with advanced features for optimization and security.

### Key Concepts

#### Static Files vs Dynamic Content

**Static Files:**

- CSS stylesheets
- JavaScript files
- Images (PNG, JPG, SVG, etc.)
- Fonts (TTF, WOFF, etc.)
- Documents (PDF, TXT, etc.)
- Audio/Video files

**Dynamic Content:**

- HTML generated from templates
- API responses
- User-specific data
- Real-time content

## Express Static Middleware

Express provides `express.static()` middleware that:

- Serves files from a specified directory
- Sets appropriate MIME types
- Handles caching headers
- Supports range requests
- Provides security features

## Virtual Paths

Virtual paths allow you to:

- Mount static files at different URL paths
- Organize files logically
- Hide actual directory structure
- Implement multiple static directories

## Example Code

### Basic Static File Serving

```
// basic-static.js
const express = require("express");
const path = require("path");
const app = express();

// Serve static files from 'public' directory
app.use(express.static("public"));

// Alternative: using absolute path
app.use(express.static(path.join(__dirname, "public")));

// Basic route
app.get("/", (req, res) => {
  res.send(`
    <!DOCTYPE html>
    <html>
    <head>
      <title>Static Files Demo</title>
      <link rel="stylesheet" href="/css/style.css">
    </head>
    <body>
      <h1>Welcome to Static Files Demo</h1>
      
  `);
});
```

```
        <script src="/js/app.js"></script>
      </body>
    </html>
  `);
});

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

## Virtual Path Mounting

```
// virtual-paths.js
const express = require("express");
const path = require("path");
const app = express();

// Mount static files with virtual paths
app.use("/static", express.static("public"));
app.use("/assets", express.static("assets"));
app.use("/uploads", express.static("uploads"));

// Multiple directories for same virtual path
app.use("/shared", express.static("public"));
app.use("/shared", express.static("assets"));

// Serve files from different directories
app.use("/css", express.static(path.join(__dirname, "styles")));
app.use("/js", express.static(path.join(__dirname, "scripts")));
app.use("/images", express.static(path.join(__dirname, "media", "images")));

// API route
app.get("/api/info", (req, res) => {
  res.json({
    message: "Static files are served at:",
    paths: {
      general: "/static/*",
      assets: "/assets/*",
      uploads: "/uploads/*",
      styles: "/css/*",
      scripts: "/js/*",
      images: "/images/*",
    },
  });
});

// Demo page
app.get("/", (req, res) => {
  res.send(`
    <!DOCTYPE html>
```

```

    <html>
    <head>
      <title>Virtual Paths Demo</title>
      <link rel="stylesheet" href="/css/main.css">
      <link rel="stylesheet" href="/static/css/theme.css">
    </head>
    <body>
      <h1>Virtual Paths Demo</h1>
      <div>
        <h2>Images from different paths:</h2>
        
        
        
      </div>
      <script src="/js/main.js"></script>
      <script src="/static/js/utils.js"></script>
    </body>
  </html>
  `);
});

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

```

## Advanced Static File Configuration

```

// advanced-static.js
const express = require("express");
const path = require("path");
const fs = require("fs");
const app = express();

// Static middleware with options
app.use(
  "/public",
  express.static("public", {
    // Set cache control headers
    maxAge: "1d", // Cache for 1 day

    // Set custom headers
    setHeaders: (res, path, stat) => {
      // Set security headers
      res.set("X-Content-Type-Options", "nosniff");

      // Set different cache times for different file types
      if (path.endsWith(".css") || path.endsWith(".js")) {
        res.set("Cache-Control", "public, max-age=31536000"); // 1 year
      }
    }
  })
);

```



```
    } else if (path.endsWith(".html")) {
      res.set("Cache-Control", "public, max-age=3600"); // 1 hour
    }

    // Add custom header with file info
    res.set("X-File-Size", stat.size);
    res.set("X-Last-Modified", stat.mtime.toISOString());
  },

  // Custom index file
  index: ["index.html", "index.htm", "default.html"],

  // Disable directory listing
  dotfiles: "ignore", // ignore, allow, deny

  // Enable/disable etag
  etag: true,

  // Enable/disable last-modified header
  lastModified: true,

  // Redirect to trailing slash
  redirect: true,
}))
);

// Conditional static serving based on environment
if (process.env.NODE_ENV === "development") {
  // In development, serve files without caching
  app.use(
    "/dev-assets",
    express.static("dev-assets", {
      maxAge: 0,
      etag: false,
      lastModified: false,
    })
  );
}

// Serve different files based on user agent
app.use("/adaptive", (req, res, next) => {
  const userAgent = req.get("User-Agent") || "";

  if (userAgent.includes("Mobile")) {
    express.static("mobile-assets")(req, res, next);
  } else {
    express.static("desktop-assets")(req, res, next);
  }
});

// Custom static file handler with authentication
app.use(
  "/protected",
  (req, res, next) => {
```

```
// Simple authentication check
const token = req.get("Authorization");

if (!token || token !== "Bearer secret-token") {
  return res.status(401).json({ error: "Unauthorized" });
}

next();
},
express.static("protected-files")
);

// File download with custom names
app.get("/download/:filename", (req, res) => {
  const filename = req.params.filename;
  const filePath = path.join(__dirname, "downloads", filename);

  // Check if file exists
  if (!fs.existsSync(filePath)) {
    return res.status(404).json({ error: "File not found" });
  }

  // Get file stats
  const stats = fs.statSync(filePath);

  // Set download headers
  res.set({
    "Content-Disposition": `attachment; filename="${filename}"`,
    "Content-Type": "application/octet-stream",
    "Content-Length": stats.size,
  });

  // Stream the file
  const fileStream = fs.createReadStream(filePath);
  fileStream.pipe(res);

  // Handle stream errors
  fileStream.on("error", (err) => {
    console.error("File stream error:", err);
    if (!res.headersSent) {
      res.status(500).json({ error: "File read error" });
    }
  });
});

// File information endpoint
app.get("/file-info/:filename", (req, res) => {
  const filename = req.params.filename;
  const filePath = path.join(__dirname, "public", filename);

  fs.stat(filePath, (err, stats) => {
    if (err) {
      return res.status(404).json({ error: "File not found" });
    }
  });
});
```

```
    res.json({
      filename,
      size: stats.size,
      created: stats.birthtime,
      modified: stats.mtime,
     .isFile: stats.isFile(),
      isDirectory: stats.isDirectory(),
      extension: path.extname(filename),
      mimeType: getMimeType(filename),
    });
  });
});

// Helper function to get MIME type
function getMimeType(filename) {
  const ext = path.extname(filename).toLowerCase();
  const mimeTypes = {
    ".html": "text/html",
    ".css": "text/css",
    ".js": "application/javascript",
    ".json": "application/json",
    ".png": "image/png",
    ".jpg": "image/jpeg",
    ".jpeg": "image/jpeg",
    ".gif": "image/gif",
    ".svg": "image/svg+xml",
    ".pdf": "application/pdf",
    ".txt": "text/plain",
  };
  return mimeTypes[ext] || "application/octet-stream";
}

// Serve SPA (Single Page Application)
app.get("*", (req, res) => {
  // For SPA, serve index.html for all non-API routes
  if (!req.path.startsWith("/api/")) {
    res.sendFile(path.join(__dirname, "public", "index.html"));
  } else {
    res.status(404).json({ error: "API endpoint not found" });
  }
});

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

## File Upload and Static Serving

```
// upload-static.js
const express = require('express');
const multer = require('multer');
const path = require('path');
const fs = require('fs');
const app = express();

// Create upload directories
const uploadDirs = ['uploads/images', 'uploads/documents', 'uploads/temp'];
uploadDirs.forEach(dir => {
  if (!fs.existsSync(dir)) {
    fs.mkdirSync(dir, { recursive: true });
  }
});

// Configure multer for file uploads
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    let uploadPath = 'uploads/temp';

    if (file.mimetype.startsWith('image/')) {
      uploadPath = 'uploads/images';
    } else if (file.mimetype === 'application/pdf' ||
      file.mimetype.includes('document')) {
      uploadPath = 'uploads/documents';
    }

    cb(null, uploadPath);
  },
  filename: (req, file, cb) => {
    const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1E9);
    const ext = path.extname(file.originalname);
    const name = path.basename(file.originalname, ext);
    cb(null, `${name}-${uniqueSuffix}${ext}`);
  }
});

const upload = multer({
  storage,
  limits: {
    fileSize: 10 * 1024 * 1024 // 10MB limit
  },
  fileFilter: (req, file, cb) => {
    // Allow specific file types
    const allowedTypes = [
      'image/jpeg',
      'image/png',
      'image/gif',
      'image/svg+xml',
      'application/pdf',
      'text/plain',
      'application/msword',
      'application/vnd.openxmlformats-
```

```
officedocument.wordprocessingml.document'
    ];

    if (allowedTypes.includes(file.mimetype)) {
        cb(null, true);
    } else {
        cb(new Error('File type not allowed'), false);
    }
}
});

// Serve uploaded files with different access levels
app.use('/uploads/images', express.static('uploads/images', {
    maxAge: '7d',
    setHeaders: (res, path) => {
        res.set('X-Content-Type-Options', 'nosniff');
        if (path.endsWith('.svg')) {
            res.set('Content-Type', 'image/svg+xml');
        }
    }
}));

// Protected document serving
app.use('/uploads/documents', (req, res, next) => {
    // Simple authentication for documents
    const auth = req.get('Authorization');
    if (!auth || !auth.startsWith('Bearer ')) {
        return res.status(401).json({ error: 'Authentication required' });
    }
    next();
}, express.static('uploads/documents'));

// Temporary files with short cache
app.use('/uploads/temp', express.static('uploads/temp', {
    maxAge: '1h'
}));

// File upload endpoint
app.post('/upload', upload.single('file'), (req, res) => {
    if (!req.file) {
        return res.status(400).json({ error: 'No file uploaded' });
    }

    const fileInfo = {
        originalName: req.file.originalname,
        filename: req.file.filename,
        size: req.file.size,
        mimetype: req.file.mimetype,
        path: req.file.path,
        url: `/${req.file.path.replace(/\\/g, '/')}` // Convert Windows paths
    };

    res.json({
        success: true,

```

```
        message: 'File uploaded successfully',
        file: fileInfo
    });
});

// Multiple file upload
app.post('/upload-multiple', upload.array('files', 5), (req, res) => {
    if (!req.files || req.files.length === 0) {
        return res.status(400).json({ error: 'No files uploaded' });
    }

    const fileInfo = req.files.map(file => ({
        originalName: file.originalname,
        filename: file.filename,
        size: file.size,
        mimetype: file.mimetype,
        url: `/${file.path.replace(/\\/g, '/')}`
    }));

    res.json({
        success: true,
        message: `${req.files.length} files uploaded successfully`,
        files: fileInfo
    });
});

// File gallery endpoint
app.get('/api/gallery', (req, res) => {
    const imagesDir = path.join(__dirname, 'uploads/images');

    fs.readdir(imagesDir, (err, files) => {
        if (err) {
            return res.status(500).json({ error: 'Failed to read directory' });
        }

        const imageFiles = files
            .filter(file => /\. (jpg|jpeg|png|gif|svg)$/i.test(file))
            .map(file => {
                const filePath = path.join(imagesDir, file);
                const stats = fs.statSync(filePath);

                return {
                    filename: file,
                    url: `/uploads/images/${file}`,
                    size: stats.size,
                    modified: stats.mtime
                };
            })
            .sort((a, b) => new Date(b.modified) - new Date(a.modified));

        res.json({
            success: true,
            images: imageFiles,
            count: imageFiles.length
        });
    });
});
```

```
    });
  });
});

// File deletion endpoint
app.delete('/api/files/:filename', (req, res) => {
  const filename = req.params.filename;

  // Search in all upload directories
  const searchDirs = ['uploads/images', 'uploads/documents', 'uploads/temp'];
  let fileFound = false;

  for (const dir of searchDirs) {
    const filePath = path.join(__dirname, dir, filename);

    if (fs.existsSync(filePath)) {
      fs.unlinkSync(filePath);
      fileFound = true;

      res.json({
        success: true,
        message: 'File deleted successfully',
        filename,
        directory: dir
      });
      break;
    }
  }

  if (!fileFound) {
    res.status(404).json({ error: 'File not found' });
  }
});

// Upload form page
app.get('/', (req, res) => {
  res.send(`
    <!DOCTYPE html>
    <html>
    <head>
      <title>File Upload Demo</title>
      <style>
        body { font-family: Arial, sans-serif; max-width: 800px; margin: 0
auto; padding: 20px; }
        .upload-form { border: 1px solid #ddd; padding: 20px; margin: 20px
0; }
        .file-list { margin: 20px 0; }
        .file-item { padding: 10px; border: 1px solid #eee; margin: 5px 0;
}
        img { max-width: 200px; max-height: 200px; }
      </style>
    </head>
    <body>
      <h1>File Upload and Static Serving Demo</h1>
  `);
});
```

```

<div class="upload-form">
  <h2>Single File Upload</h2>
  <form action="/upload" method="post" enctype="multipart/form-
data">
    <input type="file" name="file" required>
    <button type="submit">Upload</button>
  </form>
</div>

<div class="upload-form">
  <h2>Multiple File Upload</h2>
  <form action="/upload-multiple" method="post"
enctype="multipart/form-data">
    <input type="file" name="files" multiple required>
    <button type="submit">Upload Files</button>
  </form>
</div>

<div class="file-list">
  <h2>Actions</h2>
  <button onclick="loadGallery()">Load Image Gallery</button>
  <div id="gallery"></div>
</div>

<script>
  async function loadGallery() {
    try {
      const response = await fetch('/api/gallery');
      const data = await response.json();

      const gallery = document.getElementById('gallery');
      gallery.innerHTML = '<h3>Image Gallery</h3>';

      data.images.forEach(image => {
        const div = document.createElement('div');
        div.className = 'file-item';
        div.innerHTML = `
          
          <p><strong>${image.filename}</strong></p>
          <p>Size: ${image.size / 1024}.toFixed(2)} KB</p>
          <p>Modified: ${new
Date(image.modified).toLocaleString()}</p>
          <button
onclick="deleteFile('${image.filename}')">Delete</button>
        `;
        gallery.appendChild(div);
      });
    } catch (error) {
      console.error('Error loading gallery:', error);
    }
  }

  async function deleteFile(filename) {

```



```

        if (confirm('Are you sure you want to delete this file?')) {
            try {
                const response = await fetch(`/api/files/${filename}`,
{
                    method: 'DELETE'
                });

                if (response.ok) {
                    alert('File deleted successfully');
                    loadGallery();
                } else {
                    alert('Failed to delete file');
                }
            } catch (error) {
                console.error('Error deleting file:', error);
            }
        }
    }
    </script>
</body>
</html>
`);
});

// Error handling for multer
app.use((error, req, res, next) => {
    if (error instanceof multer.MulterError) {
        if (error.code === 'LIMIT_FILE_SIZE') {
            return res.status(400).json({ error: 'File too large' });
        }

        if (error.message === 'File type not allowed') {
            return res.status(400).json({ error: error.message });
        }

        next(error);
    }
});

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

```

## Real-World Use Case

### Complete Static File Server with CDN-like Features

```

// cdn-static-server.js
const express = require("express");
const path = require("path");

```

```
const fs = require("fs");
const crypto = require("crypto");
const compression = require("compression");
const app = express();

// Enable gzip compression
app.use(compression());

// Security headers middleware
app.use((req, res, next) => {
  res.set({
    "X-Content-Type-Options": "nosniff",
    "X-Frame-Options": "DENY",
    "X-XSS-Protection": "1; mode=block",
    "Referrer-Policy": "strict-origin-when-cross-origin",
  });
  next();
});

// CORS for static assets
app.use("/assets", (req, res, next) => {
  res.set({
    "Access-Control-Allow-Origin": "*",
    "Access-Control-Allow-Methods": "GET, HEAD, OPTIONS",
    "Access-Control-Allow-Headers":
      "Origin, X-Requested-With, Content-Type, Accept",
  });

  if (req.method === "OPTIONS") {
    return res.sendStatus(200);
  }

  next();
});

// Custom static middleware with versioning
const versionedStatic = (directory, options = {}) => {
  return (req, res, next) => {
    const filePath = path.join(directory, req.path);

    // Check if file exists
    if (!fs.existsSync(filePath)) {
      return next();
    }

    const stats = fs.statSync(filePath);

    // Generate ETag based on file content
    const fileContent = fs.readFileSync(filePath);
    const etag = crypto.createHash("md5").update(fileContent).digest("hex");

    // Check if client has cached version
    const clientETag = req.get("If-None-Match");
    if (clientETag === etag) {
```

```
    return res.status(304).end();
  }

  // Set cache headers based on file type
  const ext = path.extname(filePath).toLowerCase();
  let maxAge = "1d"; // Default 1 day

  if ([".css", ".js"].includes(ext)) {
    maxAge = "1y"; // CSS/JS cached for 1 year
  } else if ([".png", ".jpg", ".jpeg", ".gif", ".svg"].includes(ext)) {
    maxAge = "30d"; // Images cached for 30 days
  } else if ([".html", ".htm"].includes(ext)) {
    maxAge = "1h"; // HTML cached for 1 hour
  }

  // Set response headers
  res.set({
    "Cache-Control": `public, max-age=${parseMaxAge(maxAge)}`,
    ETag: etag,
    "Last-Modified": stats.mtime.toUTCString(),
    "Content-Length": stats.size,
  });

  // Set content type
  const mimeType = getMimeType(ext);
  if (mimeType) {
    res.set("Content-Type", mimeType);
  }

  // Stream the file
  const stream = fs.createReadStream(filePath);
  stream.pipe(res);

  stream.on("error", (err) => {
    console.error("File stream error:", err);
    if (!res.headersSent) {
      res.status(500).end();
    }
  });
};

// Helper function to parse max-age
function parseMaxAge(maxAge) {
  const units = {
    s: 1,
    m: 60,
    h: 3600,
    d: 86400,
    w: 604800,
    y: 31536000,
  };
  const match = maxAge.match(/^(\\d+)([smhdwy])$/);
```

```
    if (match) {
      return parseInt(match[1]) * units[match[2]];
    }
    return 86400; // Default 1 day
  }

// Helper function for MIME types
function getMimeType(ext) {
  const mimeTypes = {
    ".html": "text/html; charset=utf-8",
    ".css": "text/css; charset=utf-8",
    ".js": "application/javascript; charset=utf-8",
    ".json": "application/json; charset=utf-8",
    ".png": "image/png",
    ".jpg": "image/jpeg",
    ".jpeg": "image/jpeg",
    ".gif": "image/gif",
    ".svg": "image/svg+xml",
    ".ico": "image/x-icon",
    ".pdf": "application/pdf",
    ".txt": "text/plain; charset=utf-8",
    ".woff": "font/woff",
    ".woff2": "font/woff2",
    ".ttf": "font/ttf",
    ".eot": "application/vnd.ms-fontobject",
  };
  return mimeTypes[ext];
}

// Use versioned static middleware
app.use("/assets", versionedStatic(path.join(__dirname, "assets")));
app.use("/static", versionedStatic(path.join(__dirname, "public")));

// Image optimization endpoint
app.get("/images/:size/:filename", (req, res) => {
  const { size, filename } = req.params;
  const originalPath = path.join(__dirname, "images", filename);

  // Validate size parameter
  const validSizes = ["small", "medium", "large", "thumbnail"];
  if (!validSizes.includes(size)) {
    return res.status(400).json({ error: "Invalid size parameter" });
  }

  // Check if original file exists
  if (!fs.existsSync(originalPath)) {
    return res.status(404).json({ error: "Image not found" });
  }

  // For demo purposes, just serve the original
  // In production, you would resize the image here
  res.set({
    "Cache-Control": "public, max-age=2592000", // 30 days
    "X-Image-Size": size,
  });
});
```

```
});

res.sendFile(originalPath);
});

// Asset manifest endpoint
app.get("/api/manifest", (req, res) => {
  const assetsDir = path.join(__dirname, "assets");
  const manifest = {};

  function scanDirectory(dir, prefix = "") {
    const files = fs.readdirSync(dir);

    files.forEach((file) => {
      const filePath = path.join(dir, file);
      const stats = fs.statSync(filePath);

      if (stats.isDirectory()) {
        scanDirectory(filePath, `${prefix}${file}/`);
      } else {
        const relativePath = `${prefix}${file}`;
        const content = fs.readFileSync(filePath);
        const hash = crypto
          .createHash("md5")
          .update(content)
          .digest("hex")
          .substring(0, 8);

        manifest[relativePath] = {
          path: `/assets/${relativePath}`,
          hash,
          size: stats.size,
          modified: stats.mtime.toISOString(),
        };
      }
    });
  }

  if (fs.existsSync(assetsDir)) {
    scanDirectory(assetsDir);
  }

  res.json({
    manifest,
    generated: new Date().toISOString(),
    count: Object.keys(manifest).length,
  });
});

// Health check endpoint
app.get("/health", (req, res) => {
  const directories = ["assets", "public", "images"];
  const status = {};
```

```
directories.forEach((dir) => {
  const dirPath = path.join(__dirname, dir);
  status[dir] = {
    exists: fs.existsSync(dirPath),
    readable: fs.existsSync(dirPath) ? fs.constants.R_OK : false,
  };
});

res.json({
  status: "healthy",
  timestamp: new Date().toISOString(),
  directories: status,
  uptime: process.uptime(),
});
});

// Create sample directories and files
const sampleDirs = [
  "assets/css",
  "assets/js",
  "assets/images",
  "public",
  "images",
];
sampleDirs.forEach((dir) => {
  if (!fs.existsSync(dir)) {
    fs.mkdirSync(dir, { recursive: true });
  }
});

// Create sample files
const sampleFiles = {
  "assets/css/style.css": "body { font-family: Arial, sans-serif; }",
  "assets/js/app.js": 'console.log("App loaded");',
  "public/index.html":
    "<!DOCTYPE html><html><head><title>Demo</title></head><body><h1>Hello  
World</h1></body></html>",
};

Object.entries(sampleFiles).forEach(([filePath, content]) => {
  if (!fs.existsSync(filePath)) {
    fs.writeFileSync(filePath, content);
  }
});

// Default route
app.get("/", (req, res) => {
  res.send(`
    <!DOCTYPE html>
    <html>
    <head>
      <title>CDN-like Static Server</title>
      <link rel="stylesheet" href="/assets/css/style.css">
    </head>
  `);
});
```

```

        <body>
          <h1>CDN-like Static File Server</h1>
          <div>
            <h2>Available Endpoints:</h2>
            <ul>
              <li><a href="/assets/css/style.css">/assets/css/style.css</a>
</li>
              <li><a href="/assets/js/app.js">/assets/js/app.js</a></li>
              <li><a href="/static/index.html">/static/index.html</a></li>
              <li><a href="/api/manifest">/api/manifest</a></li>
              <li><a href="/health">/health</a></li>
            </ul>
          </div>
          <script src="/assets/js/app.js"></script>
        </body>
      </html>
    `);
  });

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`🚀 CDN-like static server running on http://localhost:${PORT}`);
});

```

## Best Practices

### 1. Organize Static Files Properly

```

project/
├── public/
│   ├── css/
│   ├── js/
│   ├── images/
│   ├── fonts/
│   └── favicon.ico
├── uploads/
│   ├── images/
│   └── documents/
└── assets/
    ├── vendor/
    └── build/

```

### 2. Set Appropriate Cache Headers

```

const cacheSettings = {
  // Long cache for versioned assets
  versioned: "public, max-age=31536000, immutable",

  // Medium cache for images

```

```
images: "public, max-age=2592000",

// Short cache for HTML
html: "public, max-age=3600",

// No cache for development
development: "no-cache, no-store, must-revalidate",
};
```

### 3. Implement Security Headers

```
app.use("/static", (req, res, next) => {
  res.set({
    "X-Content-Type-Options": "nosniff",
    "X-Frame-Options": "DENY",
    "Content-Security-Policy": "default-src 'self'",
  });
  next();
});
```

### 4. Use Compression

```
const compression = require("compression");
app.use(
  compression({
    filter: (req, res) => {
      // Don't compress responses if this request has a 'x-no-compression' header
      if (req.headers["x-no-compression"]) {
        return false;
      }
      // Use compression filter function
      return compression.filter(req, res);
    },
  })
);
```

### 5. Handle File Upload Security

```
const upload = multer({
  storage: multer.diskStorage({
    destination: "uploads/",
    filename: (req, file, cb) => {
      // Sanitize filename
      const sanitized = file.originalname.replace(/^[^a-zA-Z0-9.-]/g, "_");
      cb(null, `${Date.now()}-${sanitized}`);
    },
  })
});
```



```
    }},
    limits: {
      fileSize: 5 * 1024 * 1024, // 5MB
      files: 5,
    },
    fileFilter: (req, file, cb) => {
      // Whitelist allowed file types
      const allowedTypes = /jpeg|jpg|png|gif|pdf/;
      const extname = allowedTypes.test(
        path.extname(file.originalname).toLowerCase()
      );
      const mimetype = allowedTypes.test(file.mimetype);

      if (mimetype && extname) {
        return cb(null, true);
      } else {
        cb(new Error("Invalid file type"));
      }
    },
  });
```

## Summary

Serving static files in Express.js involves:

### Basic Static Serving:

- Use `express.static()` middleware
- Organize files in logical directories
- Set virtual paths for clean URLs

### Advanced Features:

- Custom cache headers for different file types
- Security headers and CORS configuration
- File upload handling with validation
- Content compression and optimization

### Performance Optimization:

- Implement proper caching strategies
- Use ETags for cache validation
- Enable gzip compression
- Set appropriate max-age values

### Security Considerations:

- Validate file types and sizes
- Sanitize file names
- Set security headers
- Implement access controls

**Production Features:**

- Asset versioning and manifests
- CDN-like caching behavior
- Health checks and monitoring
- Error handling and logging

Mastering static file serving is essential for building performant web applications. Next, we'll explore environment variables and configuration management with dotenv.

## Environment Variables with dotenv

---

### Overview

Environment variables are key-value pairs that exist outside your application code and provide configuration data to your application at runtime. They are essential for managing different configurations across development, testing, and production environments without hardcoding sensitive information like API keys, database URLs, or server ports.

The **dotenv** package is a popular Node.js library that loads environment variables from a **.env** file into **process.env**, making it easy to manage configuration in development.

### Key Concepts

#### What are Environment Variables?

Environment variables are:

- Configuration values stored outside your code
- Accessible through **process.env** in Node.js
- Different for each environment (dev, staging, production)
- Secure way to store sensitive information
- Platform-independent configuration method

#### Why Use Environment Variables?

1. **Security**: Keep sensitive data out of source code
2. **Flexibility**: Different configs for different environments
3. **Portability**: Same code works across environments
4. **Best Practice**: Industry standard for configuration
5. **CI/CD**: Easy deployment configuration

#### The dotenv Package

**dotenv** allows you to:

- Load variables from **.env** files
- Override system environment variables
- Support multiple environment files

- Provide default values
- Parse different data types

## Example Code

### Basic dotenv Setup

```
// basic-dotenv.js
// Load dotenv as early as possible
require("dotenv").config();

const express = require("express");
const app = express();

// Access environment variables
const PORT = process.env.PORT || 3000;
const NODE_ENV = process.env.NODE_ENV || "development";
const API_KEY = process.env.API_KEY;
const DATABASE_URL = process.env.DATABASE_URL;

// Middleware
app.use(express.json());

// Environment info endpoint
app.get("/env-info", (req, res) => {
  res.json({
    environment: NODE_ENV,
    port: PORT,
    hasApiKey: !!API_KEY,
    hasDatabaseUrl: !!DATABASE_URL,
    nodeVersion: process.version,
    platform: process.platform,
    // Never expose sensitive values directly
    apiKeyLength: API_KEY ? API_KEY.length : 0,
  });
});

// Configuration endpoint
app.get("/config", (req, res) => {
  const config = {
    server: {
      port: PORT,
      environment: NODE_ENV,
      debug: process.env.DEBUG === "true",
    },
    features: {
      enableLogging: process.env.ENABLE_LOGGING !== "false",
      maxUploadSize: process.env.MAX_UPLOAD_SIZE || "10mb",
      rateLimitWindow: parseInt(process.env.RATE_LIMIT_WINDOW) || 900000,
    },
    external: {
      hasApiKey: !!API_KEY,
    }
  };
  res.json(config);
});
```

```

        hasDatabase: !!DATABASE_URL,
        redisUrl: process.env.REDIS_URL ? "configured" : "not configured",
    },
};

res.json(config);
});

// Health check with environment validation
app.get("/health", (req, res) => {
    const requiredVars = ["API_KEY", "DATABASE_URL"];
    const missing = requiredVars.filter((varName) => !process.env[varName]);

    const health = {
        status: missing.length === 0 ? "healthy" : "unhealthy",
        timestamp: new Date().toISOString(),
        environment: NODE_ENV,
        missingVariables: missing,
        uptime: process.uptime(),
    };

    const statusCode = health.status === "healthy" ? 200 : 503;
    res.status(statusCode).json(health);
});

app.listen(PORT, () => {
    console.log(`🚀 Server running on port ${PORT}`);
    console.log(`📁 Environment: ${NODE_ENV}`);
    console.log(`🔑 API Key: ${API_KEY ? "Loaded" : "Missing"}`);
    console.log(`🗄 Database: ${DATABASE_URL ? "Configured" : "Missing"}`);
});

```

## Advanced dotenv Configuration

```

// advanced-dotenv.js
const path = require("path");

// Custom dotenv configuration
require("dotenv").config({
    path: path.join(__dirname, ".env"), // Custom path
    debug: process.env.DEBUG_DOTENV === "true", // Debug mode
    override: false, // Don't override existing env vars
});

// Load environment-specific files
const NODE_ENV = process.env.NODE_ENV || "development";
const envFiles = [
    `\.env.${NODE_ENV}.local`,
    `\.env.${NODE_ENV}`,
    ".env.local",
    ".env",
];

```

```
];

// Load multiple env files in order of priority
envFiles.forEach((file) => {
  require("dotenv").config({
    path: path.join(__dirname, file),
    override: false,
  });
});

const express = require("express");
const app = express();

// Configuration class
class Config {
  constructor() {
    this.server = {
      port: this.getNumber("PORT", 3000),
      host: this.getString("HOST", "localhost"),
      environment: this.getString("NODE_ENV", "development"),
    };

    this.database = {
      url: this.getString("DATABASE_URL"),
      maxConnections: this.getNumber("DB_MAX_CONNECTIONS", 10),
      timeout: this.getNumber("DB_TIMEOUT", 30000),
      ssl: this.getBoolean("DB_SSL", false),
    };

    this.redis = {
      url: this.getString("REDIS_URL"),
      ttl: this.getNumber("REDIS_TTL", 3600),
    };

    this.auth = {
      jwtSecret: this.getString("JWT_SECRET"),
      jwtExpiry: this.getString("JWT_EXPIRY", "24h"),
      bcryptRounds: this.getNumber("BCRYPT_ROUNDS", 12),
    };

    this.external = {
      apiKey: this.getString("API_KEY"),
      apiUrl: this.getString("API_URL", "https://api.example.com"),
      webhookSecret: this.getString("WEBHOOK_SECRET"),
    };

    this.features = {
      enableLogging: this.getBoolean("ENABLE_LOGGING", true),
      enableMetrics: this.getBoolean("ENABLE_METRICS", false),
      enableCache: this.getBoolean("ENABLE_CACHE", true),
      maxUploadSize: this.getString("MAX_UPLOAD_SIZE", "10mb"),
    };

    this.security = {
```

```
    corsOrigin: this.getArray("CORS_ORIGIN", ["http://localhost:3000"]),
    rateLimitWindow: this.getNumber("RATE_LIMIT_WINDOW", 900000),
    rateLimitMax: this.getNumber("RATE_LIMIT_MAX", 100),
  };

  // Validate required variables
  this.validate();
}

getString(key, defaultValue = undefined) {
  const value = process.env[key];
  if (value === undefined && defaultValue === undefined) {
    throw new Error(`Required environment variable ${key} is not set`);
  }
  return value || defaultValue;
}

getNumber(key, defaultValue = undefined) {
  const value = process.env[key];
  if (value === undefined) {
    if (defaultValue === undefined) {
      throw new Error(`Required environment variable ${key} is not set`);
    }
    return defaultValue;
  }

  const parsed = parseInt(value, 10);
  if (isNaN(parsed)) {
    throw new Error(
      `Environment variable ${key} must be a number, got: ${value}`
    );
  }
  return parsed;
}

getBoolean(key, defaultValue = undefined) {
  const value = process.env[key];
  if (value === undefined) {
    if (defaultValue === undefined) {
      throw new Error(`Required environment variable ${key} is not set`);
    }
    return defaultValue;
  }

  return value.toLowerCase() === "true";
}

getArray(key, defaultValue = undefined) {
  const value = process.env[key];
  if (value === undefined) {
    if (defaultValue === undefined) {
      throw new Error(`Required environment variable ${key} is not set`);
    }
    return defaultValue;
  }
}
```

```
    }

    return value.split(",").map((item) => item.trim());
  }

  validate() {
    const requiredVars = {
      production: ["DATABASE_URL", "JWT_SECRET", "API_KEY"],
      development: ["DATABASE_URL"],
      test: [],
    };

    const required = requiredVars[this.server.environment] || [];
    const missing = required.filter((varName) => !process.env[varName]);

    if (missing.length > 0) {
      throw new Error(
        `Missing required environment variables: ${missing.join(", ")}`
      );
    }
  }

  // Get sanitized config for logging (no secrets)
  getSanitized() {
    return {
      server: this.server,
      database: {
        ...this.database,
        url: this.database.url ? "[CONFIGURED]" : "[NOT SET]",
      },
      redis: {
        ...this.redis,
        url: this.redis.url ? "[CONFIGURED]" : "[NOT SET]",
      },
      auth: {
        jwtExpiry: this.auth.jwtExpiry,
        bcryptRounds: this.auth.bcryptRounds,
        jwtSecret: this.auth.jwtSecret ? "[CONFIGURED]" : "[NOT SET]",
      },
      external: {
        apiUrl: this.external.apiUrl,
        apiKey: this.external.apiKey ? "[CONFIGURED]" : "[NOT SET]",
        webhookSecret: this.external.webhookSecret
          ? "[CONFIGURED]"
          : "[NOT SET]",
      },
      features: this.features,
      security: this.security,
    };
  }
}

// Initialize configuration
const config = new Config();
```

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

// Configuration endpoint
app.get("/api/config", (req, res) => {
  res.json({
    success: true,
    config: config.getSanitized(),
  });
});

// Environment variables endpoint
app.get("/api/env", (req, res) => {
  // Only show non-sensitive environment info
  const envInfo = {
    nodeVersion: process.version,
    platform: process.platform,
    architecture: process.arch,
    environment: config.server.environment,
    uptime: process.uptime(),
    memoryUsage: process.memoryUsage(),
    loadedEnvFiles: envFiles.filter((file) => {
      try {
        require("fs").accessSync(path.join(__dirname, file));
        return true;
      } catch {
        return false;
      }
    }),
  };

  res.json({
    success: true,
    environment: envInfo,
  });
});

// Feature flags endpoint
app.get("/api/features", (req, res) => {
  res.json({
    success: true,
    features: config.features,
  });
});

// Validation endpoint
app.get("/api/validate", (req, res) => {
  try {
    // Re-validate configuration
    config.validate();

    res.json({
      success: true,
      message: "Configuration is valid",
    });
  } catch {
    // Handle validation error
  }
});
```



```

        environment: config.server.environment,
    });
} catch (error) {
    res.status(500).json({
        success: false,
        error: error.message,
    });
}
});

app.listen(config.server.port, config.server.host, () => {
    console.log(
        `🚀 Server running on ${config.server.host}:${config.server.port}`
    );
    console.log(`📊 Environment: ${config.server.environment}`);
    console.log("📄 Configuration loaded successfully");

    if (config.server.environment === "development") {
        console.log(
            "🔗 Sanitized config:",
            JSON.stringify(config.getSanitized(), null, 2)
        );
    }
});

// Export config for use in other modules
module.exports = config;

```

## Environment-Specific Configuration

```

// config-manager.js
const fs = require("fs");
const path = require("path");

class ConfigManager {
    constructor() {
        this.loadEnvironmentFiles();
        this.config = this.buildConfig();
    }

    loadEnvironmentFiles() {
        const NODE_ENV = process.env.NODE_ENV || "development";

        // Priority order: most specific to least specific
        const envFiles = [
            `.${env}.${NODE_ENV}.local`,
            `.${env}.local`,
            `.${env}.${NODE_ENV}`,
            ".env",
        ];
    }

```

```
console.log(`Loading environment files for: ${NODE_ENV}`);

envFiles.forEach((file) => {
  const filePath = path.join(process.cwd(), file);

  if (fs.existsSync(filePath)) {
    console.log(`✅ Loading ${file}`);
    require("dotenv").config({
      path: filePath,
      override: false, // Don't override already set variables
    });
  } else {
    console.log(`❌ Skipping ${file} (not found)`);
  }
});

buildConfig() {
  const env = process.env.NODE_ENV || "development";

  const baseConfig = {
    app: {
      name: process.env.APP_NAME || "My App",
      version: process.env.APP_VERSION || "1.0.0",
      environment: env,
      debug: this.parseBoolean(process.env.DEBUG, env === "development"),
    },

    server: {
      port: this.parseNumber(process.env.PORT, 3000),
      host: process.env.HOST || "0.0.0.0",
      timeout: this.parseNumber(process.env.SERVER_TIMEOUT, 30000),
      keepAliveTimeout: this.parseNumber(
        process.env.KEEP_ALIVE_TIMEOUT,
        5000
      ),
    },

    database: {
      url: process.env.DATABASE_URL,
      host: process.env.DB_HOST || "localhost",
      port: this.parseNumber(process.env.DB_PORT, 5432),
      name: process.env.DB_NAME || "myapp",
      username: process.env.DB_USERNAME || "postgres",
      password: process.env.DB_PASSWORD,
      ssl: this.parseBoolean(process.env.DB_SSL, env === "production"),
      pool: {
        min: this.parseNumber(process.env.DB_POOL_MIN, 2),
        max: this.parseNumber(process.env.DB_POOL_MAX, 10),
        idle: this.parseNumber(process.env.DB_POOL_IDLE, 10000),
      },
    },

    redis: {
```

```
url: process.env.REDIS_URL,
host: process.env.REDIS_HOST || "localhost",
port: this.parseNumber(process.env.REDIS_PORT, 6379),
password: process.env.REDIS_PASSWORD,
db: this.parseNumber(process.env.REDIS_DB, 0),
ttl: this.parseNumber(process.env.REDIS_TTL, 3600),
},

auth: {
  jwtSecret: process.env.JWT_SECRET,
  jwtExpiry: process.env.JWT_EXPIRY || "24h",
  refreshTokenExpiry: process.env.REFRESH_TOKEN_EXPIRY || "7d",
  bcryptRounds: this.parseNumber(process.env.BCRYPT_ROUNDS, 12),
  sessionSecret: process.env.SESSION_SECRET,
  cookieMaxAge: this.parseNumber(process.env.COOKIE_MAX_AGE, 86400000),
},

email: {
  provider: process.env.EMAIL_PROVIDER || "smtp",
  host: process.env.EMAIL_HOST,
  port: this.parseNumber(process.env.EMAIL_PORT, 587),
  secure: this.parseBoolean(process.env.EMAIL_SECURE, false),
  username: process.env.EMAIL_USERNAME,
  password: process.env.EMAIL_PASSWORD,
  from: process.env.EMAIL_FROM || "noreply@example.com",
},

storage: {
  provider: process.env.STORAGE_PROVIDER || "local",
  bucket: process.env.STORAGE_BUCKET,
  region: process.env.STORAGE_REGION,
  accessKey: process.env.STORAGE_ACCESS_KEY,
  secretKey: process.env.STORAGE_SECRET_KEY,
  uploadPath: process.env.UPLOAD_PATH || "./uploads",
  maxFileSize: process.env.MAX_FILE_SIZE || "10mb",
},

logging: {
  level:
    process.env.LOG_LEVEL || (env === "production" ? "info" : "debug"),
  file: this.parseBoolean(process.env.LOG_TO_FILE, env === "production"),
  console: this.parseBoolean(process.env.LOG_TO_CONSOLE, true),
  logPath: process.env.LOG_PATH || "./logs",
},

security: {
  corsOrigin: this.parseArray(process.env.CORS_ORIGIN, [
    "http://localhost:3000",
  ]),
  rateLimitWindow: this.parseNumber(
    process.env.RATE_LIMIT_WINDOW,
    900000
  ),
  rateLimitMax: this.parseNumber(process.env.RATE_LIMIT_MAX, 100),
```

```
helmetEnabled: this.parseBoolean(process.env.HELMET_ENABLED, true),
csrfEnabled: this.parseBoolean(
  process.env.CSRF_ENABLED,
  env === "production"
),
},
},

monitoring: {
  enabled: this.parseBoolean(
    process.env.MONITORING_ENABLED,
    env === "production"
  ),
  metricsPort: this.parseNumber(process.env.METRICS_PORT, 9090),
  healthCheckPath: process.env.HEALTH_CHECK_PATH || "/health",
},
};

// Environment-specific overrides
return this.applyEnvironmentOverrides(baseConfig, env);
}

applyEnvironmentOverrides(config, env) {
  const overrides = {
    development: {
      database: {
        ssl: false,
        pool: { max: 5 },
      },
      logging: {
        level: "debug",
        console: true,
        file: false,
      },
      security: {
        corsOrigin: ["*"],
        csrfEnabled: false,
      },
    },
    test: {
      database: {
        name: config.database.name + "_test",
        pool: { max: 2 },
      },
      logging: {
        level: "error",
        console: false,
        file: false,
      },
      auth: {
        bcryptRounds: 1, // Faster for tests
      },
    },
  },
```

```
    production: {
      server: {
        host: "0.0.0.0",
      },
      database: {
        ssl: true,
      },
      logging: {
        level: "info",
        console: false,
        file: true,
      },
      security: {
        csrfEnabled: true,
        helmetEnabled: true,
      },
      monitoring: {
        enabled: true,
      },
    },
  },
};

const envOverrides = overrides[env] || {};
return this.deepMerge(config, envOverrides);
}

parseBoolean(value, defaultValue = false) {
  if (value === undefined) return defaultValue;
  return value.toLowerCase() === "true";
}

parseNumber(value, defaultValue = 0) {
  if (value === undefined) return defaultValue;
  const parsed = parseInt(value, 10);
  return isNaN(parsed) ? defaultValue : parsed;
}

parseArray(value, defaultValue = []) {
  if (value === undefined) return defaultValue;
  return value
    .split(",")
    .map((item) => item.trim())
    .filter(Boolean);
}

deepMerge(target, source) {
  const result = { ...target };

  for (const key in source) {
    if (
      source[key] &&
      typeof source[key] === "object" &&
      !Array.isArray(source[key])
    ) {

```

```
        result[key] = this.deepMerge(target[key] || {}, source[key]);
    } else {
        result[key] = source[key];
    }
}

return result;
}

validate() {
    const requiredByEnv = {
        production: ["JWT_SECRET", "DATABASE_URL", "SESSION_SECRET"],
        development: ["DATABASE_URL"],
        test: [],
    };

    const required = requiredByEnv[this.config.app.environment] || [];
    const missing = required.filter((varName) => !process.env[varName]);

    if (missing.length > 0) {
        throw new Error(
            `Missing required environment variables for ${
                this.config.app.environment
            }
            -: ${missing.join(", ")}`
        );
    }

    // Additional validation
    if (this.config.server.port < 1 || this.config.server.port > 65535) {
        throw new Error("PORT must be between 1 and 65535");
    }

    if (
        this.config.auth.bcryptRounds < 1 ||
        this.config.auth.bcryptRounds > 20
    ) {
        throw new Error("BCRYPT_ROUNDS must be between 1 and 20");
    }
}

getSanitized() {
    const sanitized = JSON.parse(JSON.stringify(this.config));

    // Remove sensitive data
    const sensitiveKeys = [
        "password",
        "secret",
        "key",
        "token",
        "auth",
        "credential",
    ];

    const sanitize = (obj) => {
```

```

    for (const key in obj) {
      if (typeof obj[key] === "object" && obj[key] !== null) {
        sanitize(obj[key]);
      } else if (
        typeof obj[key] === "string" &&
        sensitiveKeys.some((sensitive) =>
          key.toLowerCase().includes(sensitive)
        )
      ) {
        obj[key] = obj[key] ? "[CONFIGURED]" : "[NOT SET]";
      }
    }
  };

  sanitize(sanitized);
  return sanitized;
}

get(path) {
  return path.split(".").reduce((obj, key) => obj && obj[key], this.config);
}
}

// Usage example
const configManager = new ConfigManager();
configManager.validate();

module.exports = configManager.config;
module.exports.manager = configManager;

```

## Sample .env Files

```

# .env (base configuration)
APP_NAME=My Express App
APP_VERSION=1.0.0
NODE_ENV=development

# Server Configuration
PORT=3000
HOST=localhost

# Database Configuration
DATABASE_URL=postgresql://username:password@localhost:5432/myapp
DB_POOL_MIN=2
DB_POOL_MAX=10

# Redis Configuration
REDIS_URL=redis://localhost:6379
REDIS_TTL=3600

# Authentication

```

```
JWT_SECRET=your-super-secret-jwt-key
JWT_EXPIRY=24h
BCRYPT_ROUNDS=12
SESSION_SECRET=your-session-secret

# Email Configuration
EMAIL_HOST=smtp.gmail.com
EMAIL_PORT=587
EMAIL_USERNAME=your-email@gmail.com
EMAIL_PASSWORD=your-app-password
EMAIL_FROM=noreply@yourapp.com

# Storage Configuration
STORAGE_PROVIDER=local
UPLOAD_PATH=./uploads
MAX_FILE_SIZE=10mb

# Security
CORS_ORIGIN=http://localhost:3000,http://localhost:3001
RATE_LIMIT_WINDOW=900000
RATE_LIMIT_MAX=100

# Logging
LOG_LEVEL=debug
LOG_TO_CONSOLE=true
LOG_TO_FILE=false

# Features
ENABLE_LOGGING=true
ENABLE_METRICS=false
DEBUG=true
```

```
# .env.development (development overrides)
NODE_ENV=development
DEBUG=true
LOG_LEVEL=debug

# Development database
DATABASE_URL=postgresql://dev:dev@localhost:5432/myapp_dev
DB_SSL=false

# Disable security features for development
CSRF_ENABLED=false
CORS_ORIGIN=*

# Development email (use local service)
EMAIL_HOST=localhost
EMAIL_PORT=1025
```



```
# .env.production (production overrides)
NODE_ENV=production
DEBUG=false
LOG_LEVEL=info

# Production server
PORT=8080
HOST=0.0.0.0

# Production database with SSL
DB_SSL=true
DB_POOL_MAX=20

# Enable all security features
CSRF_ENABLED=true
HELMET_ENABLED=true

# Production logging
LOG_TO_CONSOLE=false
LOG_TO_FILE=true

# Monitoring
MONITORING_ENABLED=true
METRICS_PORT=9090
```

```
# .env.test (test environment)
NODE_ENV=test
DEBUG=false
LOG_LEVEL=error

# Test database
DATABASE_URL=postgresql://test:test@localhost:5432/myapp_test
DB_POOL_MAX=2

# Fast bcrypt for tests
BCRYPT_ROUNDS=1

# Disable logging in tests
LOG_TO_CONSOLE=false
LOG_TO_FILE=false

# Test-specific settings
JWT_EXPIRY=1h
RATE_LIMIT_MAX=1000
```

## Real-World Use Case

### Complete Application with Environment Configuration

```
// app.js
const express = require("express");
const helmet = require("helmet");
const cors = require("cors");
const rateLimit = require("express-rate-limit");
const config = require("../config/config");
const logger = require("../utils/logger");

const app = express();

// Security middleware
if (config.security.helmetEnabled) {
  app.use(helmet());
}

// CORS configuration
app.use(
  cors({
    origin: config.security.corsOrigin,
    credentials: true,
  })
);

// Rate limiting
if (config.security.rateLimitMax > 0) {
  const limiter = rateLimit({
    windowMs: config.security.rateLimitWindow,
    max: config.security.rateLimitMax,
    message: {
      error: "Too many requests",
      retryAfter: Math.ceil(config.security.rateLimitWindow / 1000),
    },
  });
  app.use("/api/", limiter);
}

// Body parsing
app.use(express.json({ limit: config.storage.maxFileSize }));
app.use(
  express.urlencoded({ extended: true, limit: config.storage.maxFileSize })
);

// Request logging
app.use((req, res, next) => {
  logger.info(`${req.method} ${req.path}`, {
    ip: req.ip,
    userAgent: req.get("User-Agent"),
    requestId: req.id,
  });
  next();
});

// Health check endpoint
```

```
app.get(config.monitoring.healthCheckPath, (req, res) => {
  const health = {
    status: "healthy",
    timestamp: new Date().toISOString(),
    environment: config.app.environment,
    version: config.app.version,
    uptime: process.uptime(),
    memory: process.memoryUsage(),
    config: {
      database: config.database.url ? "connected" : "not configured",
      redis: config.redis.url ? "connected" : "not configured",
      email: config.email.host ? "configured" : "not configured",
    },
  };

  res.json(health);
});

// Configuration endpoint (development only)
if (config.app.environment === "development") {
  app.get("/api/config", (req, res) => {
    const { manager } = require("./config/config");
    res.json({
      success: true,
      config: manager.getSanitized(),
    });
  });
}

// API routes
app.use("/api/users", require("./routes/users"));
app.use("/api/auth", require("./routes/auth"));

// Error handling
app.use((err, req, res, next) => {
  logger.error("Unhandled error:", err);

  res.status(err.status || 500).json({
    success: false,
    error:
      config.app.environment === "production"
        ? "Internal server error"
        : err.message,
  });
});

// 404 handler
app.use("*", (req, res) => {
  res.status(404).json({
    success: false,
    error: "Endpoint not found",
  });
});
```

```
// Graceful shutdown
process.on("SIGTERM", () => {
  logger.info("SIGTERM received, shutting down gracefully");
  process.exit(0);
});

process.on("SIGINT", () => {
  logger.info("SIGINT received, shutting down gracefully");
  process.exit(0);
});

// Start server
const server = app.listen(config.server.port, config.server.host, () => {
  logger.info(`🚀 ${config.app.name} v${config.app.version} started`, {
    port: config.server.port,
    host: config.server.host,
    environment: config.app.environment,
    nodeVersion: process.version,
  });
});

// Set server timeouts
server.timeout = config.server.timeout;
server.keepAliveTimeout = config.server.keepAliveTimeout;

module.exports = app;
```

## Best Practices

### 1. Never Commit .env Files

```
# .gitignore
.env
.env.local
.env.*.local
.env.development
.env.production
.env.test

# But commit example files
!.env.example
```

### 2. Provide .env.example

```
# .env.example
# Copy this file to .env and fill in your values

# Application
APP_NAME=My App
```

```
NODE_ENV=development
PORT=3000

# Database
DATABASE_URL=postgresql://username:password@localhost:5432/database

# Authentication
JWT_SECRET=your-secret-key-here

# Email
EMAIL_HOST=smtp.example.com
EMAIL_USERNAME=your-email@example.com
EMAIL_PASSWORD=your-password
```

### 3. Validate Environment Variables

```
const requiredEnvVars = {
  production: ["DATABASE_URL", "JWT_SECRET", "EMAIL_PASSWORD"],
  development: ["DATABASE_URL"],
  test: [],
};

const validateEnv = () => {
  const env = process.env.NODE_ENV || "development";
  const required = requiredEnvVars[env] || [];
  const missing = required.filter((varName) => !process.env[varName]);

  if (missing.length > 0) {
    throw new Error(
      `Missing required environment variables: ${missing.join(", ")}`
    );
  }
};
```

### 4. Use Type Conversion

```
const config = {
  port: parseInt(process.env.PORT) || 3000,
  debug: process.env.DEBUG === "true",
  maxConnections: parseInt(process.env.MAX_CONNECTIONS) || 10,
  allowedOrigins: (process.env.ALLOWED_ORIGINS || "")
    .split(",")
    .filter(Boolean),
};
```

### 5. Environment-Specific Loading

```
const loadEnvFiles = () => {
  const env = process.env.NODE_ENV || "development";
  const files = [`.env.${env}.local`, `.env.local`, `.env.${env}`, ".env"];

  files.forEach((file) => {
    require("dotenv").config({ path: file, override: false });
  });
};
```

## Summary

Environment variables with dotenv provide:

### Configuration Management:

- Separate config from code
- Environment-specific settings
- Secure handling of sensitive data
- Easy deployment configuration

### Best Practices:

- Use `.env.example` for documentation
- Validate required variables
- Convert types appropriately
- Never commit actual `.env` files
- Use environment-specific files

### Advanced Features:

- Configuration classes with validation
- Type conversion and defaults
- Sanitized config for logging
- Environment-specific overrides
- Graceful error handling

### Security Considerations:

- Keep secrets out of source code
- Use different secrets per environment
- Implement proper access controls
- Log configuration without exposing secrets

Mastering environment variables is crucial for building maintainable, secure, and deployable applications. Next, we'll explore Express Router and modular route management for organizing larger applications.

# Express Router and Modular Route Management

---

## Overview

As Express.js applications grow in complexity, managing all routes in a single file becomes unwieldy and difficult to maintain. Express Router provides a powerful solution for organizing routes into modular, reusable components. It allows you to create mini-applications with their own middleware, routes, and error handling, making your codebase more organized, scalable, and maintainable.

## Key Concepts

### Express Router

Express Router is:

- A mini Express application without views or settings
- A complete middleware and routing system
- Mountable at different paths
- Capable of performing middleware and routing functions
- Stackable and composable

### Benefits of Modular Routing

1. **Organization:** Separate concerns into logical modules
2. **Maintainability:** Easier to find and modify specific functionality
3. **Reusability:** Routes can be reused across different applications
4. **Team Development:** Different developers can work on different modules
5. **Testing:** Easier to test individual route modules
6. **Scalability:** Better structure for large applications

### Router Patterns

- **Resource-based routing:** Routes organized by data entities
- **Feature-based routing:** Routes organized by application features
- **Version-based routing:** API versioning through separate routers
- **Middleware stacking:** Applying middleware at router level
- **Nested routing:** Routers within routers for complex hierarchies

## Example Code

### Basic Router Setup

```
// routes/users.js
const express = require("express");
const router = express.Router();

// Sample user data
let users = [
  { id: 1, name: "John Doe", email: "john@example.com", role: "user" },
  { id: 2, name: "Jane Smith", email: "jane@example.com", role: "admin" },
  { id: 3, name: "Bob Johnson", email: "bob@example.com", role: "user" },
];
let nextId = 4;
```

```
// Middleware specific to this router
router.use((req, res, next) => {
  console.log(`Users route accessed: ${req.method} ${req.originalUrl}`);
  next();
});

// GET /users - Get all users
router.get("/", (req, res) => {
  const { role, search, page = 1, limit = 10 } = req.query;

  let filteredUsers = [...users];

  // Filter by role
  if (role) {
    filteredUsers = filteredUsers.filter((user) => user.role === role);
  }

  // Search by name or email
  if (search) {
    const searchLower = search.toLowerCase();
    filteredUsers = filteredUsers.filter(
      (user) =>
        user.name.toLowerCase().includes(searchLower) ||
        user.email.toLowerCase().includes(searchLower)
    );
  }

  // Pagination
  const startIndex = (page - 1) * limit;
  const endIndex = startIndex + parseInt(limit);
  const paginatedUsers = filteredUsers.slice(startIndex, endIndex);

  res.json({
    success: true,
    data: paginatedUsers,
    pagination: {
      page: parseInt(page),
      limit: parseInt(limit),
      total: filteredUsers.length,
      totalPages: Math.ceil(filteredUsers.length / limit),
    },
  });
});

// GET /users/:id - Get user by ID
router.get("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const user = users.find((u) => u.id === id);

  if (!user) {
    return res.status(404).json({
      success: false,
      error: "User not found",
    });
  }
});
```



```
}

res.json({
  success: true,
  data: user,
});
});

// POST /users - Create new user
router.post("/", (req, res) => {
  const { name, email, role = "user" } = req.body;

  // Validation
  if (!name || !email) {
    return res.status(400).json({
      success: false,
      error: "Name and email are required",
    });
  }

  // Check if email already exists
  if (users.find((u) => u.email === email)) {
    return res.status(409).json({
      success: false,
      error: "Email already exists",
    });
  }

  const newUser = {
    id: nextId++,
    name,
    email,
    role,
  };

  users.push(newUser);

  res.status(201).json({
    success: true,
    data: newUser,
    message: "User created successfully",
  });
});

// PUT /users/:id - Update user
router.put("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const userIndex = users.findIndex((u) => u.id === id);

  if (userIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "User not found",
    });
  }
});
```

```
}

const { name, email, role } = req.body;

// Check if email is being changed and already exists
if (email && email !== users[userIndex].email) {
  if (users.find((u) => u.email === email)) {
    return res.status(409).json({
      success: false,
      error: "Email already exists",
    });
  }
}

// Update user
users[userIndex] = {
  ...users[userIndex],
  ...(name && { name }),
  ...(email && { email }),
  ...(role && { role }),
};

res.json({
  success: true,
  data: users[userIndex],
  message: "User updated successfully",
});

// DELETE /users/:id - Delete user
router.delete("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const userIndex = users.findIndex((u) => u.id === id);

  if (userIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "User not found",
    });
  }

  const deletedUser = users.splice(userIndex, 1)[0];

  res.json({
    success: true,
    data: deletedUser,
    message: "User deleted successfully",
  });
});

module.exports = router;
```

## Advanced Router with Middleware

```
// routes/posts.js
const express = require("express");
const router = express.Router();

// Sample posts data
let posts = [
  {
    id: 1,
    title: "First Post",
    content: "This is the first post",
    authorId: 1,
    published: true,
    createdAt: new Date(),
  },
  {
    id: 2,
    title: "Second Post",
    content: "This is the second post",
    authorId: 2,
    published: false,
    createdAt: new Date(),
  },
  {
    id: 3,
    title: "Third Post",
    content: "This is the third post",
    authorId: 1,
    published: true,
    createdAt: new Date(),
  },
];
let nextId = 4;

// Validation middleware
const validatePost = (req, res, next) => {
  const { title, content, authorId } = req.body;
  const errors = [];

  if (!title || title.trim().length === 0) {
    errors.push("Title is required");
  }

  if (!content || content.trim().length === 0) {
    errors.push("Content is required");
  }

  if (!authorId || isNaN(parseInt(authorId))) {
    errors.push("Valid author ID is required");
  }
}
```

```
    if (errors.length > 0) {
      return res.status(400).json({
        success: false,
        errors,
      });
    }

    next();
  };

  // Authentication middleware (simplified)
  const authenticate = (req, res, next) => {
    const token = req.get("Authorization");

    if (!token) {
      return res.status(401).json({
        success: false,
        error: "Authentication required",
      });
    }

    // Simplified token validation
    if (token !== "Bearer valid-token") {
      return res.status(401).json({
        success: false,
        error: "Invalid token",
      });
    }

    // Add user info to request
    req.user = { id: 1, role: "admin" };
    next();
  };

  // Authorization middleware
  const authorize = (roles = []) => {
    return (req, res, next) => {
      if (!req.user) {
        return res.status(401).json({
          success: false,
          error: "Authentication required",
        });
      }

      if (roles.length > 0 && !roles.includes(req.user.role)) {
        return res.status(403).json({
          success: false,
          error: "Insufficient permissions",
        });
      }

      next();
    };
  };
};
```

```
// Logging middleware for this router
router.use((req, res, next) => {
  console.log(
    `Posts route: ${req.method} ${
      req.originalUrl
    } at ${new Date().toISOString()}`
  );
  next();
});

// GET /posts - Get all posts (public)
router.get("/", (req, res) => {
  const {
    published,
    authorId,
    search,
    sortBy = "createdAt",
    order = "desc",
  } = req.query;

  let filteredPosts = [...posts];

  // Filter by published status
  if (published !== undefined) {
    filteredPosts = filteredPosts.filter(
      (post) => post.published === (published === "true")
    );
  }

  // Filter by author
  if (authorId) {
    filteredPosts = filteredPosts.filter(
      (post) => post.authorId === parseInt(authorId)
    );
  }

  // Search in title and content
  if (search) {
    const searchLower = search.toLowerCase();
    filteredPosts = filteredPosts.filter(
      (post) =>
        post.title.toLowerCase().includes(searchLower) ||
        post.content.toLowerCase().includes(searchLower)
    );
  }

  // Sorting
  filteredPosts.sort((a, b) => {
    let aVal = a[sortBy];
    let bVal = b[sortBy];

    if (sortBy === "createdAt") {
      aVal = new Date(aVal);
    }
  });
});
```

```
        bVal = new Date(bVal);
    }

    if (order === "desc") {
        return bVal > aVal ? 1 : -1;
    } else {
        return aVal > bVal ? 1 : -1;
    }
});

res.json({
    success: true,
    data: filteredPosts,
    count: filteredPosts.length,
});
});

// GET /posts/:id - Get single post (public)
router.get("/:id", (req, res) => {
    const id = parseInt(req.params.id);
    const post = posts.find((p) => p.id === id);

    if (!post) {
        return res.status(404).json({
            success: false,
            error: "Post not found",
        });
    }

    res.json({
        success: true,
        data: post,
    });
});

// POST /posts - Create new post (requires authentication)
router.post("/", authenticate, validatePost, (req, res) => {
    const { title, content, authorId, published = false } = req.body;

    const newPost = {
        id: nextId++,
        title: title.trim(),
        content: content.trim(),
        authorId: parseInt(authorId),
        published,
        createdAt: new Date(),
        updatedAt: new Date(),
    };

    posts.push(newPost);

    res.status(201).json({
        success: true,
        data: newPost,
    });
});
```

```
    message: "Post created successfully",
  });
});

// PUT /posts/:id - Update post (requires authentication)
router.put("/:id", authenticate, validatePost, (req, res) => {
  const id = parseInt(req.params.id);
  const postIndex = posts.findIndex((p) => p.id === id);

  if (postIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "Post not found",
    });
  }

  const { title, content, published } = req.body;

  // Update post
  posts[postIndex] = {
    ...posts[postIndex],
    title: title.trim(),
    content: content.trim(),
    published,
    updatedAt: new Date(),
  };

  res.json({
    success: true,
    data: posts[postIndex],
    message: "Post updated successfully",
  });
});

// DELETE /posts/:id - Delete post (requires admin role)
router.delete("/:id", authenticate, authorize(["admin"]), (req, res) => {
  const id = parseInt(req.params.id);
  const postIndex = posts.findIndex((p) => p.id === id);

  if (postIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "Post not found",
    });
  }

  const deletedPost = posts.splice(postIndex, 1)[0];

  res.json({
    success: true,
    data: deletedPost,
    message: "Post deleted successfully",
  });
});
```

```
// PATCH /posts/:id/publish - Toggle publish status (requires authentication)
router.patch("/:id/publish", authenticate, (req, res) => {
  const id = parseInt(req.params.id);
  const postIndex = posts.findIndex((p) => p.id === id);

  if (postIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "Post not found",
    });
  }

  posts[postIndex].published = !posts[postIndex].published;
  posts[postIndex].updatedAt = new Date();

  res.json({
    success: true,
    data: posts[postIndex],
    message: `Post ${
      posts[postIndex].published ? "published" : "unpublished"
    } successfully`,
  });
});

module.exports = router;
```

## Nested Routers and Route Parameters

```
// routes/api.js - Main API router
const express = require("express");
const router = express.Router();

// Import sub-routers
const usersRouter = require("./users");
const postsRouter = require("./posts");
const commentsRouter = require("./comments");
const authRouter = require("./auth");

// API-wide middleware
router.use((req, res, next) => {
  res.set({
    "X-API-Version": "1.0.0",
    "X-Powered-By": "Express.js",
  });
  next();
});

// Request logging
router.use((req, res, next) => {
  console.log(`API Request: ${req.method} ${req.originalUrl}`);
});
```



```
    console.log("Headers:", req.headers);
    console.log("Body:", req.body);
    next();
  });

  // Mount sub-routers
  router.use("/auth", authRouter);
  router.use("/users", usersRouter);
  router.use("/posts", postsRouter);
  router.use("/posts/:postId/comments", commentsRouter); // Nested route

  // API info endpoint
  router.get("/", (req, res) => {
    res.json({
      success: true,
      message: "Welcome to the API",
      version: "1.0.0",
      endpoints: {
        auth: "/api/auth",
        users: "/api/users",
        posts: "/api/posts",
        comments: "/api/posts/:postId/comments",
      },
      documentation: "/api/docs",
    });
  });

  // API documentation endpoint
  router.get("/docs", (req, res) => {
    const documentation = {
      version: "1.0.0",
      baseUrl: "/api",
      endpoints: {
        auth: {
          "POST /auth/login": "Authenticate user",
          "POST /auth/register": "Register new user",
          "POST /auth/logout": "Logout user",
        },
        users: {
          "GET /users": "Get all users",
          "GET /users/:id": "Get user by ID",
          "POST /users": "Create new user",
          "PUT /users/:id": "Update user",
          "DELETE /users/:id": "Delete user",
        },
        posts: {
          "GET /posts": "Get all posts",
          "GET /posts/:id": "Get post by ID",
          "POST /posts": "Create new post",
          "PUT /posts/:id": "Update post",
          "DELETE /posts/:id": "Delete post",
          "PATCH /posts/:id/publish": "Toggle publish status",
        },
        comments: {
```

```

        "GET /posts/:postId/comments": "Get comments for post",
        "POST /posts/:postId/comments": "Add comment to post",
        "PUT /posts/:postId/comments/:id": "Update comment",
        "DELETE /posts/:postId/comments/:id": "Delete comment",
    },
  },
};

res.json({
  success: true,
  documentation,
});
});

module.exports = router;

```

```

// routes/comments.js - Nested router for comments
const express = require("express");
const router = express.Router({ mergeParams: true }); // Important for accessing
parent params

// Sample comments data
let comments = [
  {
    id: 1,
    postId: 1,
    content: "Great post!",
    authorId: 2,
    createdAt: new Date(),
  },
  {
    id: 2,
    postId: 1,
    content: "Thanks for sharing",
    authorId: 3,
    createdAt: new Date(),
  },
  {
    id: 3,
    postId: 2,
    content: "Interesting perspective",
    authorId: 1,
    createdAt: new Date(),
  },
];
let nextId = 4;

// Middleware to validate post exists
const validatePost = (req, res, next) => {
  const postId = parseInt(req.params.postId);

```

```
// In a real app, you'd check the database
// For demo, we'll assume posts with IDs 1-3 exist
if (isNaN(postId) || postId < 1 || postId > 3) {
  return res.status(404).json({
    success: false,
    error: "Post not found",
  });
}

req.postId = postId;
next();
};

// Apply post validation to all routes
router.use(validatePost);

// GET /posts/:postId/comments - Get comments for a post
router.get("/", (req, res) => {
  const postComments = comments.filter(
    (comment) => comment.postId === req.postId
  );

  res.json({
    success: true,
    data: postComments,
    postId: req.postId,
    count: postComments.length,
  });
});

// GET /posts/:postId/comments/:id - Get specific comment
router.get("/:id", (req, res) => {
  const commentId = parseInt(req.params.id);
  const comment = comments.find(
    (c) => c.id === commentId && c.postId === req.postId
  );

  if (!comment) {
    return res.status(404).json({
      success: false,
      error: "Comment not found",
    });
  }

  res.json({
    success: true,
    data: comment,
  });
});

// POST /posts/:postId/comments - Add comment to post
router.post("/", (req, res) => {
  const { content, authorId } = req.body;
```

```
if (!content || !authorId) {
  return res.status(400).json({
    success: false,
    error: "Content and author ID are required",
  });
}

const newComment = {
  id: nextId++,
  postId: req.postId,
  content: content.trim(),
  authorId: parseInt(authorId),
  createdAt: new Date(),
};

comments.push(newComment);

res.status(201).json({
  success: true,
  data: newComment,
  message: "Comment added successfully",
});
});

// PUT /posts/:postId/comments/:id - Update comment
router.put("/:id", (req, res) => {
  const commentId = parseInt(req.params.id);
  const commentIndex = comments.findIndex(
    (c) => c.id === commentId && c.postId === req.postId
  );

  if (commentIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "Comment not found",
    });
  }

  const { content } = req.body;

  if (!content) {
    return res.status(400).json({
      success: false,
      error: "Content is required",
    });
  }

  comments[commentIndex] = {
    ...comments[commentIndex],
    content: content.trim(),
    updatedAt: new Date(),
  };

  res.json({
```

```
        success: true,
        data: comments[commentIndex],
        message: "Comment updated successfully",
    });
});

// DELETE /posts/:postId/comments/:id - Delete comment
router.delete("/:id", (req, res) => {
    const commentId = parseInt(req.params.id);
    const commentIndex = comments.findIndex(
        (c) => c.id === commentId && c.postId === req.postId
    );

    if (commentIndex === -1) {
        return res.status(404).json({
            success: false,
            error: "Comment not found",
        });
    }

    const deletedComment = comments.splice(commentIndex, 1)[0];

    res.json({
        success: true,
        data: deletedComment,
        message: "Comment deleted successfully",
    });
});

module.exports = router;
```

## Main Application with Router Integration

```
// app.js - Main application file
const express = require("express");
const cors = require("cors");
const helmet = require("helmet");
const morgan = require("morgan");

const app = express();

// Global middleware
app.use(helmet());
app.use(cors());
app.use(morgan("combined"));
app.use(express.json({ limit: "10mb" }));
app.use(express.urlencoded({ extended: true }));

// Import routers
const apiRouter = require("./routes/api");
const adminRouter = require("./routes/admin");
```

```
const webhooksRouter = require("./routes/webhooks");

// Mount routers
app.use("/api", apiRouter);
app.use("/admin", adminRouter);
app.use("/webhooks", webhooksRouter);

// Root endpoint
app.get("/", (req, res) => {
  res.json({
    success: true,
    message: "Welcome to the Express Router Demo",
    version: "1.0.0",
    routes: {
      api: "/api",
      admin: "/admin",
      webhooks: "/webhooks",
    },
    timestamp: new Date().toISOString(),
  });
});

// Health check
app.get("/health", (req, res) => {
  res.json({
    status: "healthy",
    timestamp: new Date().toISOString(),
    uptime: process.uptime(),
    memory: process.memoryUsage(),
  });
});

// 404 handler
app.use("*", (req, res) => {
  res.status(404).json({
    success: false,
    error: "Route not found",
    path: req.originalUrl,
    method: req.method,
  });
});

// Global error handler
app.use((err, req, res, next) => {
  console.error("Global error handler:", err);

  res.status(err.status || 500).json({
    success: false,
    error:
      process.env.NODE_ENV === "production"
        ? "Internal server error"
        : err.message,
    ...(process.env.NODE_ENV !== "production" && { stack: err.stack }),
  });
});
```

```
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`🚀 Server running on port ${PORT}`);
  console.log(
    `📖 API documentation available at http://localhost:${PORT}/api/docs`
  );
});

module.exports = app;
```

## Real-World Use Case

### E-commerce API with Modular Routing

```
// routes/products.js
const express = require("express");
const router = express.Router();

// Product data (in real app, this would be a database)
let products = [
  { id: 1, name: "Laptop", price: 999.99, category: "Electronics", stock: 10 },
  { id: 2, name: "Book", price: 19.99, category: "Education", stock: 50 },
  { id: 3, name: "Coffee Mug", price: 9.99, category: "Kitchen", stock: 25 },
];

// Middleware for product validation
const validateProduct = (req, res, next) => {
  const { name, price, category } = req.body;
  const errors = [];

  if (!name || name.trim().length === 0) {
    errors.push("Product name is required");
  }

  if (!price || isNaN(parseFloat(price)) || parseFloat(price) <= 0) {
    errors.push("Valid price is required");
  }

  if (!category || category.trim().length === 0) {
    errors.push("Category is required");
  }

  if (errors.length > 0) {
    return res.status(400).json({ success: false, errors });
  }

  next();
};
```

```
// GET /products - Get all products with filtering and pagination
router.get("/", (req, res) => {
  const {
    category,
    minPrice,
    maxPrice,
    search,
    sortBy = "name",
    order = "asc",
    page = 1,
    limit = 10,
  } = req.query;

  let filteredProducts = [...products];

  // Apply filters
  if (category) {
    filteredProducts = filteredProducts.filter(
      (p) => p.category.toLowerCase() === category.toLowerCase()
    );
  }

  if (minPrice) {
    filteredProducts = filteredProducts.filter(
      (p) => p.price >= parseFloat(minPrice)
    );
  }

  if (maxPrice) {
    filteredProducts = filteredProducts.filter(
      (p) => p.price <= parseFloat(maxPrice)
    );
  }

  if (search) {
    const searchLower = search.toLowerCase();
    filteredProducts = filteredProducts.filter(
      (p) =>
        p.name.toLowerCase().includes(searchLower) ||
        p.category.toLowerCase().includes(searchLower)
    );
  }

  // Apply sorting
  filteredProducts.sort((a, b) => {
    let aVal = a[sortBy];
    let bVal = b[sortBy];

    if (typeof aVal === "string") {
      aVal = aVal.toLowerCase();
      bVal = bVal.toLowerCase();
    }

    if (order === "desc") {

```



```
        return bVal > aVal ? 1 : -1;
    } else {
        return aVal > bVal ? 1 : -1;
    }
});

// Apply pagination
const startIndex = (page - 1) * limit;
const endIndex = startIndex + parseInt(limit);
const paginatedProducts = filteredProducts.slice(startIndex, endIndex);

res.json({
  success: true,
  data: paginatedProducts,
  pagination: {
    page: parseInt(page),
    limit: parseInt(limit),
    total: filteredProducts.length,
    totalPages: Math.ceil(filteredProducts.length / limit),
  },
  filters: { category, minPrice, maxPrice, search },
  sorting: { sortBy, order },
});
});

// GET /products/categories - Get all categories
router.get("/categories", (req, res) => {
  const categories = [...new Set(products.map((p) => p.category))];

  res.json({
    success: true,
    data: categories,
    count: categories.length,
  });
});

// GET /products/:id - Get single product
router.get("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const product = products.find((p) => p.id === id);

  if (!product) {
    return res.status(404).json({
      success: false,
      error: "Product not found",
    });
  }

  res.json({
    success: true,
    data: product,
  });
});
```

```
// POST /products - Create new product (admin only)
router.post("/", validateProduct, (req, res) => {
  const { name, price, category, stock = 0 } = req.body;

  const newProduct = {
    id: Math.max(...products.map((p) => p.id)) + 1,
    name: name.trim(),
    price: parseFloat(price),
    category: category.trim(),
    stock: parseInt(stock) || 0,
    createdAt: new Date(),
    updatedAt: new Date(),
  };

  products.push(newProduct);

  res.status(201).json({
    success: true,
    data: newProduct,
    message: "Product created successfully",
  });
});

// PUT /products/:id - Update product
router.put("/:id", validateProduct, (req, res) => {
  const id = parseInt(req.params.id);
  const productIndex = products.findIndex((p) => p.id === id);

  if (productIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "Product not found",
    });
  }

  const { name, price, category, stock } = req.body;

  products[productIndex] = {
    ...products[productIndex],
    name: name.trim(),
    price: parseFloat(price),
    category: category.trim(),
    stock: parseInt(stock) || 0,
    updatedAt: new Date(),
  };

  res.json({
    success: true,
    data: products[productIndex],
    message: "Product updated successfully",
  });
});

// PATCH /products/:id/stock - Update stock only
```

```
router.patch("/:id/stock", (req, res) => {
  const id = parseInt(req.params.id);
  const productIndex = products.findIndex((p) => p.id === id);

  if (productIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "Product not found",
    });
  }

  const { stock } = req.body;

  if (stock === undefined || isNaN(parseInt(stock))) {
    return res.status(400).json({
      success: false,
      error: "Valid stock quantity is required",
    });
  }

  products[productIndex].stock = parseInt(stock);
  products[productIndex].updatedAt = new Date();

  res.json({
    success: true,
    data: products[productIndex],
    message: "Stock updated successfully",
  });
});

// DELETE /products/:id - Delete product
router.delete("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const productIndex = products.findIndex((p) => p.id === id);

  if (productIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "Product not found",
    });
  }

  const deletedProduct = products.splice(productIndex, 1)[0];

  res.json({
    success: true,
    data: deletedProduct,
    message: "Product deleted successfully",
  });
});

module.exports = router;
```

```
// routes/orders.js
const express = require("express");
const router = express.Router();

// Sample orders data
let orders = [
  {
    id: 1,
    userId: 1,
    items: [
      { productId: 1, quantity: 1, price: 999.99 },
      { productId: 3, quantity: 2, price: 9.99 },
    ],
    total: 1019.97,
    status: "pending",
    createdAt: new Date(),
  },
];
let nextId = 2;

// Order validation middleware
const validateOrder = (req, res, next) => {
  const { userId, items } = req.body;

  if (!userId || isNaN(parseInt(userId))) {
    return res.status(400).json({
      success: false,
      error: "Valid user ID is required",
    });
  }

  if (!items || !Array.isArray(items) || items.length === 0) {
    return res.status(400).json({
      success: false,
      error: "Order must contain at least one item",
    });
  }

  // Validate each item
  for (const item of items) {
    if (!item.productId || !item.quantity || !item.price) {
      return res.status(400).json({
        success: false,
        error: "Each item must have productId, quantity, and price",
      });
    }
  }

  next();
};

// GET /orders - Get all orders
router.get("/", (req, res) => {
```

```
const { userId, status, page = 1, limit = 10 } = req.query;

let filteredOrders = [...orders];

if (userId) {
  filteredOrders = filteredOrders.filter(
    (order) => order.userId === parseInt(userId)
  );
}

if (status) {
  filteredOrders = filteredOrders.filter((order) => order.status === status);
}

// Pagination
const startIndex = (page - 1) * limit;
const endIndex = startIndex + parseInt(limit);
const paginatedOrders = filteredOrders.slice(startIndex, endIndex);

res.json({
  success: true,
  data: paginatedOrders,
  pagination: {
    page: parseInt(page),
    limit: parseInt(limit),
    total: filteredOrders.length,
    totalPages: Math.ceil(filteredOrders.length / limit),
  },
});
});

// GET /orders/:id - Get single order
router.get("/:id", (req, res) => {
  const id = parseInt(req.params.id);
  const order = orders.find((o) => o.id === id);

  if (!order) {
    return res.status(404).json({
      success: false,
      error: "Order not found",
    });
  }

  res.json({
    success: true,
    data: order,
  });
});

// POST /orders - Create new order
router.post("/", validateOrder, (req, res) => {
  const { userId, items } = req.body;

  // Calculate total
```

```
const total = items.reduce((sum, item) => {
  return sum + item.price * item.quantity;
}, 0);

const newOrder = {
  id: nextId++,
  userId: parseInt(userId),
  items,
  total: parseFloat(total.toFixed(2)),
  status: "pending",
  createdAt: new Date(),
  updatedAt: new Date(),
};

orders.push(newOrder);

res.status(201).json({
  success: true,
  data: newOrder,
  message: "Order created successfully",
});
});

// PATCH /orders/:id/status - Update order status
router.patch("/:id/status", (req, res) => {
  const id = parseInt(req.params.id);
  const orderIndex = orders.findIndex((o) => o.id === id);

  if (orderIndex === -1) {
    return res.status(404).json({
      success: false,
      error: "Order not found",
    });
  }

  const { status } = req.body;
  const validStatuses = [
    "pending",
    "processing",
    "shipped",
    "delivered",
    "cancelled",
  ];

  if (!status || !validStatuses.includes(status)) {
    return res.status(400).json({
      success: false,
      error: `Status must be one of: ${validStatuses.join(", ")}`,
    });
  }

  orders[orderIndex].status = status;
  orders[orderIndex].updatedAt = new Date();
});
```

```
    res.json({
      success: true,
      data: orders[orderIndex],
      message: "Order status updated successfully",
    });
  });
});

module.exports = router;
```

## Best Practices

### 1. Organize Routes by Resource

```
routes/
├── api.js           # Main API router
├── auth.js          # Authentication routes
├── users.js          # User management
├── products.js       # Product management
├── orders.js         # Order management
├── admin.js          # Admin routes
└── webhooks.js       # Webhook endpoints
```

### 2. Use Middleware Effectively

```
// Apply middleware at router level
router.use(authenticate); // All routes require auth
router.use("/admin", authorize(["admin"])); // Admin routes only

// Route-specific middleware
router.post("/users", validateUser, createUser);
router.put(
  "/users/:id",
  validateUser,
  authorize(["admin", "user"]),
  updateUser
);
```

### 3. Handle Route Parameters

```
// Use router.param for parameter preprocessing
router.param("id", (req, res, next, id) => {
  const numericId = parseInt(id);
  if (isNaN(numericId)) {
    return res.status(400).json({ error: "Invalid ID format" });
  }
  req.params.id = numericId;
});
```

```
    next();  
  });
```

## 4. Implement Consistent Error Handling

```
// Async error wrapper  
const asyncHandler = (fn) => (req, res, next) => {  
  Promise.resolve(fn(req, res, next)).catch(next);  
};  
  
// Use in routes  
router.get(  
  "/users",  
  asyncHandler(async (req, res) => {  
    const users = await User.find();  
    res.json({ success: true, data: users });  
  })  
);
```

## 5. Version Your APIs

```
// routes/v1/index.js  
const router = express.Router();  
router.use("/users", require("./users"));  
router.use("/posts", require("./posts"));  
  
// routes/v2/index.js  
const router = express.Router();  
router.use("/users", require("./users"));  
router.use("/posts", require("./posts"));  
  
// app.js  
app.use("/api/v1", require("./routes/v1"));  
app.use("/api/v2", require("./routes/v2"));
```

## Summary

Express Router enables modular route management through:

### Core Features:

- Mini Express applications with routing and middleware
- Mountable at different paths
- Parameter handling with `mergeParams`
- Route-specific and router-level middleware

### Organization Benefits:



- Separation of concerns by resource or feature
- Easier maintenance and testing
- Better team collaboration
- Reusable route modules

#### Advanced Patterns:

- Nested routers for complex hierarchies
- Parameter preprocessing with `router.param()`
- Middleware stacking for authentication and authorization
- API versioning through separate routers

#### Best Practices:

- Organize routes by logical groupings
- Use consistent error handling
- Implement proper validation middleware
- Document API endpoints
- Version APIs for backward compatibility

Mastering Express Router is essential for building scalable, maintainable Express.js applications. Next, we'll explore template engines like EJS and Pug for server-side rendering.

## Template Engines: EJS and Pug for Server-Side Rendering

---

### Overview

Template engines allow you to generate dynamic HTML content on the server side by combining static templates with dynamic data. Express.js supports various template engines, with EJS (Embedded JavaScript) and Pug (formerly Jade) being among the most popular. Template engines enable you to create reusable layouts, inject data into HTML, implement conditional rendering, and build complete web applications with server-side rendering (SSR).

### Key Concepts

#### Template Engines Fundamentals

**Template Engine:** A tool that combines templates (static markup) with data to produce HTML output.

#### Benefits:

- **Server-Side Rendering (SSR):** Better SEO and initial page load performance
- **Dynamic Content:** Inject data from databases or APIs
- **Reusable Components:** Create layouts and partials
- **Logic in Templates:** Conditional rendering and loops
- **Security:** Automatic escaping prevents XSS attacks

#### EJS (Embedded JavaScript)

**Characteristics:**

- Uses plain JavaScript syntax
- Familiar HTML-like structure
- Easy to learn for JavaScript developers
- Supports includes and layouts
- Good performance

**Syntax:**

- `<% %>` - JavaScript code (no output)
- `<%= %>` - Output escaped content
- `<%- %>` - Output unescaped content
- `<%# %>` - Comments
- `<%- include('partial') %>` - Include partials

**Pug (formerly Jade)****Characteristics:**

- Indentation-based syntax (no closing tags)
- Concise and clean
- Built-in features like mixins and inheritance
- Powerful templating features
- Steeper learning curve

**Syntax:**

- Indentation defines nesting
- `#{} - Interpolation`
- `!{} - Unescaped interpolation`
- `// - Comments`
- `include` and `extends` for modularity

## Example Code

**Setting Up EJS**

```
// app.js - EJS Setup
const express = require("express");
const path = require("path");
const app = express();

// Set EJS as template engine
app.set("view engine", "ejs");
app.set("views", path.join(__dirname, "views"));

// Middleware
app.use(express.static("public"));
app.use(express.urlencoded({ extended: true }));
```

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

// Sample data
const users = [
  {
    id: 1,
    name: "John Doe",
    email: "john@example.com",
    role: "admin",
    active: true,
  },
  {
    id: 2,
    name: "Jane Smith",
    email: "jane@example.com",
    role: "user",
    active: true,
  },
  {
    id: 3,
    name: "Bob Johnson",
    email: "bob@example.com",
    role: "user",
    active: false,
  },
];

const posts = [
  {
    id: 1,
    title: "Getting Started with EJS",
    content: "EJS is a simple templating language...",
    authorId: 1,
    published: true,
    createdAt: new Date("2024-01-15"),
  },
  {
    id: 2,
    title: "Advanced EJS Techniques",
    content: "Learn advanced EJS features...",
    authorId: 2,
    published: true,
    createdAt: new Date("2024-01-20"),
  },
  {
    id: 3,
    title: "Draft Post",
    content: "This is a draft...",
    authorId: 1,
    published: false,
    createdAt: new Date("2024-01-25"),
  },
];
```

```
// Routes
app.get("/", (req, res) => {
  const publishedPosts = posts.filter((post) => post.published);
  res.render("index", {
    title: "Welcome to EJS Demo",
    posts: publishedPosts,
    users,
    currentUser: users[0], // Simulate logged-in user
  });
});

app.get("/users", (req, res) => {
  res.render("users", {
    title: "User Management",
    users,
    currentUser: users[0],
  });
});

app.get("/users/:id", (req, res) => {
  const userId = parseInt(req.params.id);
  const user = users.find((u) => u.id === userId);

  if (!user) {
    return res.status(404).render("error", {
      title: "User Not Found",
      error: "The requested user could not be found.",
      currentUser: users[0],
    });
  }

  const userPosts = posts.filter((post) => post.authorId === userId);

  res.render("user-detail", {
    title: `User: ${user.name}`,
    user,
    posts: userPosts,
    currentUser: users[0],
  });
});

app.get("/posts/new", (req, res) => {
  res.render("post-form", {
    title: "Create New Post",
    post: null,
    users,
    currentUser: users[0],
  });
});

app.post("/posts", (req, res) => {
  const { title, content, authorId, published } = req.body;

  const newPost = {
```

```

    id: Math.max(...posts.map((p) => p.id)) + 1,
    title,
    content,
    authorId: parseInt(authorId),
    published: published === "on",
    createdAt: new Date(),
  });

  posts.push(newPost);
  res.redirect("/");
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`🚀 EJS Server running on http://localhost:${PORT}`);
});

```

## EJS Templates

```

<!-- views/layout.ejs - Main Layout -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title><%= title %> | EJS Demo</title>
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
      rel="stylesheet"
    />
    <link
      href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0/css/all.min.css"
      rel="stylesheet"
    />
    <style>
      .navbar-brand {
        font-weight: bold;
      }
      .post-card {
        transition: transform 0.2s;
      }
      .post-card:hover {
        transform: translateY(-2px);
      }
      .user-avatar {
        width: 40px;
        height: 40px;
      }
      footer {

```

```

        margin-top: 50px;
    }
</style>
</head>
<body>
    <!-- Navigation -->
    <%- include('partials/navbar') %>

    <!-- Main Content -->
    <main class="container mt-4">
        <!-- Flash Messages -->
        <% if (typeof message !== 'undefined' && message) { %>
            <div class="alert alert-info alert-dismissible fade show" role="alert">
                <%= message %>
                <button
                    type="button"
                    class="btn-close"
                    data-bs-dismiss="alert"
                ></button>
            </div>
            <% } %>

        <!-- Page Content -->
        <%- body %>
    </main>

    <!-- Footer -->
    <%- include('partials/footer') %>

    <script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js"
></script>
    </body>
</html>

```

```

<!-- views/partials/navbar.ejs -->
<nav class="navbar navbar-expand-lg navbar-dark bg-primary">
    <div class="container">
        <a class="navbar-brand" href="/"> <i class="fas fa-code"></i> EJS Demo </a>

        <button
            class="navbar-toggler"
            type="button"
            data-bs-toggle="collapse"
            data-bs-target="#navbarNav"
        >
            <span class="navbar-toggler-icon"></span>
        </button>

        <div class="collapse navbar-collapse" id="navbarNav">
            <ul class="navbar-nav me-auto">

```

```

    <li class="nav-item">
      <a class="nav-link" href="/"> <i class="fas fa-home"></i> Home </a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="/users">
        <i class="fas fa-users"></i> Users
      </a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="/posts/new">
        <i class="fas fa-plus"></i> New Post
      </a>
    </li>
  </ul>

  <!-- User Info -->
  <% if (typeof currentUser !== 'undefined' && currentUser) { %>
    <div class="navbar-nav">
      <div class="nav-item dropdown">
        <a
          class="nav-link dropdown-toggle"
          href="#"
          role="button"
          data-bs-toggle="dropdown"
        >
          "
            class="rounded-circle user-avatar me-2"
          />
          <%= currentUser.name %>
        </a>
        <ul class="dropdown-menu">
          <li>
            <a class="dropdown-item" href="/users/<%= currentUser.id %>"
              >Profile</a>
          </li>
          <li><a class="dropdown-item" href="/settings">Settings</a></li>
          <li><hr class="dropdown-divider" /></li>
          <li><a class="dropdown-item" href="/logout">Logout</a></li>
        </ul>
      </div>
    </div>
  <% } %>
</div>
</div>
</nav>

```

```

<!-- views/partials/footer.ejs -->
<footer class="bg-light py-4 mt-5">
  <div class="container">
    <div class="row">
      <div class="col-md-6">
        <h5>EJS Demo Application</h5>
        <p class="text-muted">Built with Express.js and EJS template engine.</p>
      </div>
      <div class="col-md-6 text-md-end">
        <p class="text-muted">
          &copy; <%= new Date().getFullYear() %> EJS Demo. Generated at <%= new
            Date().toLocaleString() %>
        </p>
      </div>
    </div>
  </div>
</footer>

```

```

<!-- views/index.ejs -->
<% // Define the body content const bodyContent = `
<div class="row">
  <div class="col-lg-8">
    <h1 class="mb-4"><i class="fas fa-newspaper"></i> Latest Posts</h1>

    <% if (posts && posts.length > 0) { %> <% posts.forEach(post => { %> <%
      const author = users.find(u => u.id === post.authorId) %>
      <div class="card post-card mb-4 shadow-sm">
        <div class="card-body">
          <div class="d-flex justify-content-between align-items-start mb-3">
            <h5 class="card-title mb-0">
              <a href="/posts/<%= post.id %>" class="text-decoration-none">
                <%= post.title %>
              </a>
            </h5>
            <% if (post.published) { %>
              <span class="badge bg-success">Published</span>
            <% } else { %>
              <span class="badge bg-warning">Draft</span>
            <% } %>
          </div>

          <p class="card-text text-muted">
            <%= post.content.substring(0, 150) %> <% if (post.content.length >
              150) { %>...<% } %>
          </p>

          <div class="d-flex justify-content-between align-items-center">
            <div class="d-flex align-items-center">
              <img
                src="https://ui-avatars.com/api/?name=<%=
encodeURIComponent(author.name) %>&background=random"

```



```

        alt="<%= author.name %>"
        class="rounded-circle me-2"
        width="32"
        height="32"
    />
    <small class="text-muted">
        By
        <a href="/users/<%= author.id %>" class="text-decoration-none"
            ><%= author.name %></a>
        >
        on <%= post.createdAt.toLocaleDateString() %>
    </small>
</div>
<a
    href="/posts/<%= post.id %>"
    class="btn btn-outline-primary btn-sm"
    >
    Read More <i class="fas fa-arrow-right"></i>
</a>
</div>
</div>
</div>
<% }) %> <% } else { %>
<div class="text-center py-5">
    <i class="fas fa-newspaper fa-3x text-muted mb-3"></i>
    <h3 class="text-muted">No posts available</h3>
    <p class="text-muted">Be the first to create a post!</p>
    <a href="/posts/new" class="btn btn-primary">
        <i class="fas fa-plus"></i> Create Post
    </a>
</div>
<% } %>
</div>

<div class="col-lg-4">
    <div class="card">
        <div class="card-header">
            <h5 class="mb-0"><i class="fas fa-users"></i> Active Users</h5>
        </div>
        <div class="card-body">
            <% const activeUsers = users.filter(u => u.active) %> <%
            activeUsers.forEach(user => { %>
            <div class="d-flex align-items-center mb-3">
                "
                    class="rounded-circle me-3"
                    width="40"
                    height="40"
                />
            </div>
            <h6 class="mb-0">
                <a href="/users/<%= user.id %>" class="text-decoration-none"

```

```

        ><%= user.name %></a
      >
    </h6>
    <small class="text-muted">
      <i class="fas fa-crown text-warning"></i> <%= user.role %>
    </small>
  </div>
</div>
<% }) %>
</div>
</div>

<div class="card mt-4">
  <div class="card-header">
    <h5 class="mb-0"><i class="fas fa-chart-bar"></i> Statistics</h5>
  </div>
  <div class="card-body">
    <div class="row text-center">
      <div class="col-6">
        <h3 class="text-primary">
          <%= posts.filter(p => p.published).length %>
        </h3>
        <small class="text-muted">Published Posts</small>
      </div>
      <div class="col-6">
        <h3 class="text-success">
          <%= users.filter(u => u.active).length %>
        </h3>
        <small class="text-muted">Active Users</small>
      </div>
    </div>
  </div>
</div>
</div>
</div>
</div>
<%> <%- include('layout', { body: bodyContent }) %>

```

```

<!-- views/users.ejs -->
<% const bodyContent = `
<div class="d-flex justify-content-between align-items-center mb-4">
  <h1><i class="fas fa-users"></i> User Management</h1>
  <a href="/users/new" class="btn btn-primary">
    <i class="fas fa-plus"></i> Add User
  </a>
</div>

<div class="card">
  <div class="card-body">
    <div class="table-responsive">
      <table class="table table-hover">
        <thead class="table-dark">

```

```

        <tr>
          <th>Avatar</th>
          <th>Name</th>
          <th>Email</th>
          <th>Role</th>
          <th>Status</th>
          <th>Actions</th>
        </tr>
      </thead>
      <tbody>
        <% users.forEach(user => { %>
          <tr>
            <td>
              "
                class="rounded-circle"
                width="40"
                height="40"
              />
            </td>
            <td>
              <strong><%= user.name %></strong>
            </td>
            <td>
              <a href="mailto:<%= user.email %>" class="text-decoration-none">
                <%= user.email %>
              </a>
            </td>
            <td>
              <% if (user.role === 'admin') { %>
                <span class="badge bg-danger">
                  <i class="fas fa-crown"></i> Admin
                </span>
              <% } else { %>
                <span class="badge bg-primary">
                  <i class="fas fa-user"></i> User
                </span>
              <% } %>
            </td>
            <td>
              <% if (user.active) { %>
                <span class="badge bg-success">
                  <i class="fas fa-check"></i> Active
                </span>
              <% } else { %>
                <span class="badge bg-secondary">
                  <i class="fas fa-times"></i> Inactive
                </span>
              <% } %>
            </td>
            <td>
              <div class="btn-group" role="group">

```

```

        <a
          href="/users/<%= user.id %>"
          class="btn btn-sm btn-outline-primary"
        >
          <i class="fas fa-eye"></i>
        </a>
        <a
          href="/users/<%= user.id %>/edit"
          class="btn btn-sm btn-outline-warning"
        >
          <i class="fas fa-edit"></i>
        </a>
        <% if (user.id !== currentUser.id) { %>
        <button
          class="btn btn-sm btn-outline-danger"
          onclick="deleteUser(<%= user.id %>)"
        >
          <i class="fas fa-trash"></i>
        </button>
        <% } %>
      </div>
    </td>
  </tr>
  <% }) %>
</tbody>
</table>
</div>
</div>
</div>
</div>

<script>
function deleteUser(userId) {
  if (confirm('Are you sure you want to delete this user?')) {
    fetch(`/users/${userId}`, {
      method: 'DELETE',
      headers: {
        'Content-Type': 'application/json'
      }
    })
    .then(response => {
      if (response.ok) {
        location.reload();
      } else {
        alert('Error deleting user');
      }
    })
    .catch(error => {
      console.error('Error:', error);
      alert('Error deleting user');
    });
  }
}
</script>
<% <- include('layout', { body: bodyContent }) %>

```

## Setting Up Pug

```
// app-pug.js - Pug Setup
const express = require("express");
const path = require("path");
const app = express();

// Set Pug as template engine
app.set("view engine", "pug");
app.set("views", path.join(__dirname, "views-pug"));

// Middleware
app.use(express.static("public"));
app.use(express.urlencoded({ extended: true }));
app.use(express.json());

// Helper functions for templates
app.locals.formatDate = (date) => {
  return new Date(date).toLocaleDateString("en-US", {
    year: "numeric",
    month: "long",
    day: "numeric",
  });
};

app.locals.truncate = (text, length = 100) => {
  return text.length > length ? text.substring(0, length) + "..." : text;
};

// Sample data (same as EJS example)
const users = [
  {
    id: 1,
    name: "John Doe",
    email: "john@example.com",
    role: "admin",
    active: true,
  },
  {
    id: 2,
    name: "Jane Smith",
    email: "jane@example.com",
    role: "user",
    active: true,
  },
  {
    id: 3,
    name: "Bob Johnson",
    email: "bob@example.com",
    role: "user",
  }
];
```

```
    active: false,
  },
];

const posts = [
  {
    id: 1,
    title: "Getting Started with Pug",
    content: "Pug is a clean, whitespace-sensitive syntax...",
    authorId: 1,
    published: true,
    createdAt: new Date("2024-01-15"),
  },
  {
    id: 2,
    title: "Advanced Pug Features",
    content: "Learn about mixins, inheritance, and more...",
    authorId: 2,
    published: true,
    createdAt: new Date("2024-01-20"),
  },
  {
    id: 3,
    title: "Draft Post",
    content: "This is a draft...",
    authorId: 1,
    published: false,
    createdAt: new Date("2024-01-25"),
  },
];

// Routes
app.get("/", (req, res) => {
  const publishedPosts = posts.filter((post) => post.published);
  res.render("index", {
    title: "Welcome to Pug Demo",
    posts: publishedPosts,
    users,
    currentUser: users[0],
  });
});

app.get("/users", (req, res) => {
  res.render("users", {
    title: "User Management",
    users,
    currentUser: users[0],
  });
});

app.get("/about", (req, res) => {
  res.render("about", {
    title: "About Pug",
    features: [
```

```

    {
      name: "Clean Syntax",
      description: "No closing tags, indentation-based",
    },
    {
      name: "Powerful Features",
      description: "Mixins, inheritance, includes",
    },
    {
      name: "JavaScript Integration",
      description: "Full JavaScript support",
    },
    { name: "Performance", description: "Compiled templates for speed" },
  ],
  currentUser: users[0],
});
});

const PORT = process.env.PORT || 3001;
app.listen(PORT, () => {
  console.log(`🚀 Pug Server running on http://localhost:${PORT}`);
});

```

## Pug Templates

```

//- views-pug/layout.pug - Main Layout
doctype html
html(lang='en')
  head
    meta(charset='UTF-8')
    meta(name='viewport', content='width=device-width, initial-scale=1.0')
    title #{title} | Pug Demo

  link(href='https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css', rel='stylesheet')
  link(href='https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0/css/all.min.css', rel='stylesheet')
  style.
    .navbar-brand { font-weight: bold; }
    .post-card { transition: transform 0.2s; }
    .post-card:hover { transform: translateY(-2px); }
    .user-avatar { width: 40px; height: 40px; }
    footer { margin-top: 50px; }

  body
    //- Navigation
    include partials/navbar

    //- Main Content
    main.container.mt-4
      //- Flash Messages

```

```

    if message
      .alert.alert-info.alert-dismissible.fade.show(role='alert')
        = message
      button.btn-close(type='button', data-bs-dismiss='alert')

```

```

//- Page Content
block content

```

```

//- Footer
include partials/footer

```

```

script(src='https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.js')

```

```

//- views-pug/partials/navbar.pug
nav.navbar.navbar-expand-lg.navbar-dark.bg-primary
  .container
    a.navbar-brand(href='/')
      i.fas.fa-code
      | Pug Demo

    button.navbar-toggler(type='button', data-bs-toggle='collapse', data-bs-target='#navbarNav')
      span.navbar-toggler-icon

    .collapse.navbar-collapse#navbarNav
      ul.navbar-nav.me-auto
        li.nav-item
          a.nav-link(href='/')
            i.fas.fa-home
            | Home
        li.nav-item
          a.nav-link(href='/users')
            i.fas.fa-users
            | Users
        li.nav-item
          a.nav-link(href='/about')
            i.fas.fa-info-circle
            | About

      //- User Info
      if currentUser
        .navbar-nav
          .nav-item.dropdown
            a.nav-link.dropdown-toggle(href='#', role='button', data-bs-toggle='dropdown')
              img.rounded-circle.user-avatar.me-2(src=`https://ui-avatars.com/api/?name=${encodeURIComponent(currentUser.name)}&background=random`, alt=currentUser.name)
              = currentUser.name

```



```

    ul.dropdown-menu
      li: a.dropdown-item(href=`/users/${currentUser.id}`) Profile
      li: a.dropdown-item(href='/settings') Settings
      li: hr.dropdown-divider
      li: a.dropdown-item(href='/logout') Logout

```

```

//- views-pug/partials/footer.pug
footer.bg-light.py-4.mt-5
  .container
    .row
      .col-md-6
        h5 Pug Demo Application
        p.text-muted Built with Express.js and Pug template engine.
      .col-md-6.text-md-end
        p.text-muted
          | &copy; #{new Date().getFullYear()} Pug Demo.
          | Generated at #{new Date().toLocaleString()}

```

```

//- views-pug/mixins/post-card.pug
mixin postCard(post, author)
  .card.post-card.mb-4.shadow-sm
    .card-body
      .d-flex.justify-content-between.align-items-start.mb-3
        h5.card-title.mb-0
          a.text-decoration-none(href=`/posts/${post.id}`)= post.title
        if post.published
          span.badge.bg-success Published
        else
          span.badge.bg-warning Draft

      p.card-text.text-muted= truncate(post.content, 150)

      .d-flex.justify-content-between.align-items-center
        .d-flex.align-items-center
          img.rounded-circle.me-2(src=`https://ui-avatars.com/api/?
name=${encodeURIComponent(author.name)}&background=random`, alt=author.name,
width='32', height='32')
          small.text-muted
            | By
            a.text-decoration-none(href=`/users/${author.id}`)= author.name
            | on #{formatDate(post.createdAt)}
          a.btn.btn-outline-primary.btn-sm(href=`/posts/${post.id}`)
            | Read More
            i.fas.fa-arrow-right

mixin userBadge(user)
  .d-flex.align-items-center.mb-3
    img.rounded-circle.me-3(src=`https://ui-avatars.com/api/?
name=${encodeURIComponent(user.name)}&background=random`, alt=user.name,

```

```
width='40', height='40')
  div
    h6.mb-0
      a.text-decoration-none(href=`/users/${user.id}`)= user.name
    small.text-muted
      i.fas.fa-crown.text-warning
      | #{user.role}
```

```
//- views-pug/index.pug
extends layout
include mixins/post-card

block content
  .row
    .col-lg-8
      h1.mb-4
        i.fas.fa-newspaper
        | Latest Posts

      if posts && posts.length > 0
        each post in posts
          - const author = users.find(u => u.id === post.authorId)
          +postCard(post, author)
      else
        .text-center.py-5
          i.fas.fa-newspaper.fa-3x.text-muted.mb-3
          h3.text-muted No posts available
          p.text-muted Be the first to create a post!
          a.btn.btn-primary(href='/posts/new')
            i.fas.fa-plus
            | Create Post

    .col-lg-4
      .card
        .card-header
          h5.mb-0
            i.fas.fa-users
            | Active Users
        .card-body
          - const activeUsers = users.filter(u => u.active)
          each user in activeUsers
            +userBadge(user)

      .card.mt-4
        .card-header
          h5.mb-0
            i.fas.fa-chart-bar
            | Statistics
        .card-body
          .row.text-center
            .col-6
```

```

      h3.text-primary= posts.filter(p => p.published).length
      small.text-muted Published Posts
    .col-6
      h3.text-success= users.filter(u => u.active).length
      small.text-muted Active Users

```

```

//- views-pug/about.pug
extends layout

block content
  .row.justify-content-center
    .col-lg-8
      h1.text-center.mb-5
        i.fas.fa-info-circle.text-primary
        | About Pug Template Engine

      .card.shadow
        .card-body.p-5
          p.lead.text-center.mb-4
            | Pug is a clean, whitespace-sensitive syntax for writing HTML.

          h3.mb-4 Key Features

          .row
            each feature in features
              .col-md-6.mb-4
                .card.h-100.border-0.bg-light
                  .card-body.text-center
                    h5.card-title.text-primary= feature.name
                    p.card-text= feature.description

          h3.mb-3 Syntax Comparison

          .row
            .col-md-6
              h5 HTML
              pre.bg-light.p-3
                code.
                  <div class="container">
                    <h1>Hello World</h1>
                    <p>Welcome to our site</p>
                  </div>

            .col-md-6
              h5 Pug
              pre.bg-light.p-3
                code.
                  .container
                    h1 Hello World
                    p Welcome to our site

```

```
.text-center.mt-4
a.btn.btn-primary.btn-lg(href='https://pugjs.org', target='_blank')
  | Learn More About Pug
i.fas.fa-external-link-alt.ms-2
```

## Real-World Use Case

### Blog Application with Both Template Engines

```
// blog-app.js - Complete Blog Application
const express = require('express');
const path = require('path');
const fs = require('fs').promises;
const app = express();

// Configuration
const config = {
  templateEngine: process.env.TEMPLATE_ENGINE || 'ejs', // 'ejs' or 'pug'
  port: process.env.PORT || 3000,
  postsPerPage: 5
};

// Set template engine based on config
if (config.templateEngine === 'pug') {
  app.set('view engine', 'pug');
  app.set('views', path.join(__dirname, 'views-pug'));
} else {
  app.set('view engine', 'ejs');
  app.set('views', path.join(__dirname, 'views'));
}

// Middleware
app.use(express.static('public'));
app.use(express.urlencoded({ extended: true }));
app.use(express.json());

// Template helpers
app.locals.formatDate = (date) => {
  return new Date(date).toLocaleDateString('en-US', {
    year: 'numeric',
    month: 'long',
    day: 'numeric',
    hour: '2-digit',
    minute: '2-digit'
  });
};

app.locals.truncate = (text, length = 100) => {
  return text.length > length ? text.substring(0, length) + '...' : text;
};
```

```
app.locals.slugify = (text) => {
  return text.toLowerCase()
    .replace(/[^\w\s-]/g, '')
    .replace(/[\s_-]+/g, '-')
    .replace(/^-+|-+$/g, '');
};

// Data storage (in production, use a database)
let blogData = {
  posts: [
    {
      id: 1,
      title: 'Getting Started with Template Engines',
      slug: 'getting-started-template-engines',
      content: `Template engines are powerful tools that allow developers to
generate dynamic HTML content by combining static templates with dynamic data. In
this comprehensive guide, we'll explore the fundamentals of template engines and
how they can revolutionize your web development workflow.

## What are Template Engines?

Template engines separate the presentation layer from the business logic, making
your code more maintainable and organized. They allow you to:

- Create reusable layouts and components
- Inject dynamic data into static templates
- Implement conditional rendering and loops
- Maintain clean separation of concerns

## Popular Template Engines

### EJS (Embedded JavaScript)
EJS uses familiar JavaScript syntax and HTML-like structure, making it easy for
developers to adopt.

### Pug (formerly Jade)
Pug offers a clean, indentation-based syntax that eliminates the need for closing
tags.

### Handlebars
Handlebars provides a logic-less templating approach with powerful helpers.

## Best Practices

1. **Keep logic minimal** - Templates should focus on presentation
2. **Use partials** - Break down templates into reusable components
3. **Escape user input** - Prevent XSS attacks with proper escaping
4. **Cache templates** - Improve performance in production

Template engines are essential tools for modern web development, enabling
developers to create dynamic, maintainable, and secure web applications.`
      excerpt: 'Learn the fundamentals of template engines and how they can
improve your web development workflow.',
      authorId: 1,
    }
  ]
}
```

```

        categoryId: 1,
        published: true,
        featured: true,
        tags: ['web-development', 'templates', 'ejs', 'pug'],
        createdAt: new Date('2024-01-15T10:00:00Z'),
        updatedAt: new Date('2024-01-15T10:00:00Z'),
        views: 1250,
        likes: 89
      },
      {
        id: 2,
        title: 'Advanced EJS Techniques and Best Practices',
        slug: 'advanced-ejs-techniques',
        content: `EJS (Embedded JavaScript) is one of the most popular
template engines for Node.js applications. While it's easy to get started with
EJS, mastering its advanced features can significantly improve your development
productivity and code quality.

## Advanced EJS Features

### Custom Delimiters
You can customize EJS delimiters to avoid conflicts with other templating systems:

\\\`javascript
app.set('view options', {
  delimiter: '?'
});
\\\`

### Includes with Data
Pass data to included templates:

\\\`ejs
<%- include('partials/header', { pageTitle: 'Custom Title' }) %>
\\\`

### Caching for Performance
Enable template caching in production:

\\\`javascript
if (process.env.NODE_ENV === 'production') {
  app.set('view cache', true);
}
\\\`

### Error Handling
Implement proper error handling in templates:

\\\`ejs
<% try { %>
  <%= user.name %>
<% } catch (error) { %>
  <span class="text-muted">Name not available</span>
<% } %>

```

```
\\`\\`
```

## ## Performance Optimization

1. **\*\*Enable caching\*\*** in production environments
2. **\*\*Minimize logic\*\*** in templates
3. **\*\*Use partials\*\*** for reusable components
4. **\*\*Precompile templates\*\*** for better performance

## ## Security Considerations

- Always use `<<%= %>` for user input to prevent XSS
- Validate data before passing to templates
- Use Content Security Policy (CSP) headers
- Sanitize HTML content when necessary

By following these advanced techniques and best practices, you can build robust, secure, and performant web applications with EJS.`,

excerpt: 'Discover advanced EJS techniques, performance optimization tips, and security best practices.',

```
authorId: 2,
categoryId: 1,
published: true,
featured: false,
tags: ['ejs', 'performance', 'security', 'best-practices'],
createdAt: new Date('2024-01-20T14:30:00Z'),
updatedAt: new Date('2024-01-20T14:30:00Z'),
views: 892,
likes: 67
```

```
},
{
```

```
id: 3,
title: 'Mastering Pug: From Basics to Advanced Features',
slug: 'mastering-pug-template-engine',
content: `Pug (formerly known as Jade) is a powerful, clean, and
feature-rich template engine for Node.js. Its indentation-based syntax and
powerful features make it a favorite among developers who value clean,
maintainable code.
```

## ## Why Choose Pug?

### ### Clean Syntax

Pug's indentation-based syntax eliminates the need for closing tags:

```
\\`\\`\\`pug
div.container
  h1.title Welcome to Pug
  p.description This is much cleaner than HTML
\\`\\`\\`
```

### ### Powerful Features

#### #### Mixins

Create reusable template functions:

```

\`\`\`pug
mixin button(text, type='button')
  button(class=\`btn btn-\${type}\`)= text

```

```

+button('Click me', 'primary')
+button('Cancel', 'secondary')
\`\`\`

```

#### #### Template Inheritance

Extend layouts for consistent structure:

```

\`\`\`pug
//- layout.pug
doctype html
html
  head
    title= title
  body
    block content

```

```

//- page.pug
extends layout
block content
  h1 Page Content
\`\`\`

```

#### #### Filters

Process content with built-in filters:

```

\`\`\`pug
script.
  const message = 'Hello from Pug!';
  console.log(message);

```

```

style.
  .highlight {
    background-color: yellow;
  }
\`\`\`

```

## ## Advanced Techniques

### ### Conditional Classes

```

\`\`\`pug
div(class={ active: isActive, disabled: !isEnabled })
\`\`\`

```

### ### Iteration with Index

```

\`\`\`pug
ul
  each item, index in items
    li(class=index % 2 ? 'odd' : 'even')= item
\`\`\`

```



### Case Statements

```

\`\`\`pug
case user.role
  when 'admin'
    p You are an administrator
  when 'user'
    p You are a regular user
  default
    p Unknown role
\`\`\`

```

### Best Practices

1. **Use mixins** for reusable components
2. **Leverage inheritance** for consistent layouts
3. **Keep indentation consistent** (2 or 4 spaces)
4. **Use meaningful variable names**
5. **Comment complex logic**

Pug's powerful features and clean syntax make it an excellent choice for developers who want to write maintainable, readable templates while leveraging advanced templating capabilities.`,

```

    excerpt: 'Explore Pug\'s powerful features including mixins,
inheritance, and advanced templating techniques.',
    authorId: 1,
    categoryId: 1,
    published: true,
    featured: true,
    tags: ['pug', 'jade', 'templates', 'mixins', 'inheritance'],
    createdAt: new Date('2024-01-25T09:15:00Z'),
    updatedAt: new Date('2024-01-25T09:15:00Z'),
    views: 1456,
    likes: 123
  }
],
authors: [
  {
    id: 1,
    name: 'Alex Johnson',
    email: 'alex@example.com',
    bio: 'Full-stack developer with 8+ years of experience in Node.js and
modern web technologies.',
    avatar: 'https://ui-avatars.com/api/?
name=Alex+Johnson&background=random',
    social: {
      twitter: '@alexjohnson',
      github: 'alexjohnson',
      linkedin: 'alex-johnson'
    },
    active: true
  },
  {
    id: 2,

```

```
      name: 'Sarah Chen',
      email: 'sarah@example.com',
      bio: 'Frontend specialist and UI/UX enthusiast. Passionate about
creating beautiful, accessible web experiences.',
      avatar: 'https://ui-avatars.com/api/?
name=Sarah+Chen&background=random',
      social: {
        twitter: '@sarahchen',
        github: 'sarahchen',
        dribbble: 'sarahchen'
      },
      active: true
    }
  ],
  categories: [
    { id: 1, name: 'Web Development', slug: 'web-development', description:
'Articles about web development technologies and best practices' },
    { id: 2, name: 'JavaScript', slug: 'javascript', description: 'JavaScript
tutorials, tips, and advanced techniques' },
    { id: 3, name: 'Node.js', slug: 'nodejs', description: 'Server-side
JavaScript with Node.js' }
  ]
};

// Helper functions
const getPostsByCategory = (categoryId) => {
  return blogData.posts.filter(post => post.categoryId === categoryId &&
post.published);
};

const getPostsByTag = (tag) => {
  return blogData.posts.filter(post => post.tags.includes(tag) &&
post.published);
};

const getFeaturedPosts = () => {
  return blogData.posts.filter(post => post.featured && post.published);
};

const getPopularPosts = (limit = 5) => {
  return blogData.posts
    .filter(post => post.published)
    .sort((a, b) => b.views - a.views)
    .slice(0, limit);
};

// Routes
app.get('/', (req, res) => {
  const page = parseInt(req.query.page) || 1;
  const limit = config.postsPerPage;
  const offset = (page - 1) * limit;

  const publishedPosts = blogData.posts.filter(post => post.published);
  const totalPosts = publishedPosts.length;
```

```
const totalPages = Math.ceil(totalPosts / limit);

const posts = publishedPosts
  .sort((a, b) => new Date(b.createdAt) - new Date(a.createdAt))
  .slice(offset, offset + limit);

const featuredPosts = getFeaturedPosts();
const popularPosts = getPopularPosts(3);

res.render('blog/index', {
  title: 'Template Engine Blog',
  posts,
  featuredPosts,
  popularPosts,
  authors: blogData.authors,
  categories: blogData.categories,
  pagination: {
    page,
    totalPages,
    hasNext: page < totalPages,
    hasPrev: page > 1,
    nextPage: page + 1,
    prevPage: page - 1
  },
  templateEngine: config.templateEngine
});
});

app.get('/post/:slug', (req, res) => {
  const post = blogData.posts.find(p => p.slug === req.params.slug &&
    p.published);

  if (!post) {
    return res.status(404).render('error', {
      title: 'Post Not Found',
      error: 'The requested blog post could not be found.',
      statusCode: 404
    });
  }

  // Increment views
  post.views++;

  const author = blogData.authors.find(a => a.id === post.authorId);
  const category = blogData.categories.find(c => c.id === post.categoryId);
  const relatedPosts = getPostsByCategory(post.categoryId)
    .filter(p => p.id !== post.id)
    .slice(0, 3);

  res.render('blog/post', {
    title: post.title,
    post,
    author,
    category,
  });
});
```

```
        relatedPosts,  
        authors: blogData.authors  
    });  
});  
  
app.get('/category/:slug', (req, res) => {  
    const category = blogData.categories.find(c => c.slug === req.params.slug);  
  
    if (!category) {  
        return res.status(404).render('error', {  
            title: 'Category Not Found',  
            error: 'The requested category could not be found.',  
            statusCode: 404  
        });  
    }  
  
    const posts = getPostsByCategory(category.id);  
  
    res.render('blog/category', {  
        title: `Category: ${category.name}`,  
        category,  
        posts,  
        authors: blogData.authors  
    });  
});  
  
app.get('/tag/:tag', (req, res) => {  
    const tag = req.params.tag;  
    const posts = getPostsByTag(tag);  
  
    res.render('blog/tag', {  
        title: `Tag: ${tag}`,  
        tag,  
        posts,  
        authors: blogData.authors  
    });  
});  
  
app.get('/author/:id', (req, res) => {  
    const authorId = parseInt(req.params.id);  
    const author = blogData.authors.find(a => a.id === authorId);  
  
    if (!author) {  
        return res.status(404).render('error', {  
            title: 'Author Not Found',  
            error: 'The requested author could not be found.',  
            statusCode: 404  
        });  
    }  
  
    const posts = blogData.posts.filter(p => p.authorId === authorId &&  
p.published);  
  
    res.render('blog/author', {
```

```
        title: `Author: ${author.name}`,
        author,
        posts
    });
});

// API endpoints for AJAX requests
app.get('/api/posts/search', (req, res) => {
    const { q, category, tag } = req.query;
    let posts = blogData.posts.filter(post => post.published);

    if (q) {
        const query = q.toLowerCase();
        posts = posts.filter(post =>
            post.title.toLowerCase().includes(query) ||
            post.content.toLowerCase().includes(query) ||
            post.excerpt.toLowerCase().includes(query)
        );
    }

    if (category) {
        const categoryId = parseInt(category);
        posts = posts.filter(post => post.categoryId === categoryId);
    }

    if (tag) {
        posts = posts.filter(post => post.tags.includes(tag));
    }

    res.json({
        success: true,
        data: posts.map(post => ({
            id: post.id,
            title: post.title,
            slug: post.slug,
            excerpt: post.excerpt,
            author: blogData.authors.find(a => a.id === post.authorId)?.name,
            category: blogData.categories.find(c => c.id ===
post.categoryId)?.name,
            createdAt: post.createdAt,
            views: post.views,
            likes: post.likes
        })),
        count: posts.length
    });
});

// Error handling
app.use((req, res) => {
    res.status(404).render('error', {
        title: 'Page Not Found',
        error: 'The requested page could not be found.',
        statusCode: 404
    });
});
```

```
});

app.use((err, req, res, next) => {
  console.error('Error:', err);
  res.status(500).render('error', {
    title: 'Server Error',
    error: process.env.NODE_ENV === 'production'
      ? 'An internal server error occurred.'
      : err.message,
    statusCode: 500
  });
});

app.listen(config.port, () => {
  console.log(`🚀 Blog Server running on http://localhost:${config.port}`);
  console.log(`📄 Template Engine: ${config.templateEngine.toUpperCase()}`);
  console.log(`📖 Total Posts: ${blogData.posts.length}`);
});

module.exports = app;
```

## Best Practices

### 1. Choose the Right Template Engine

#### Use EJS when:

- Team is familiar with HTML and JavaScript
- Need quick setup and minimal learning curve
- Working with existing HTML templates
- Prefer explicit syntax

#### Use Pug when:

- Want clean, concise syntax
- Need powerful features like mixins and inheritance
- Building new projects from scratch
- Team values code brevity

### 2. Organize Templates Effectively

```
views/
├── layouts/
│   ├── main.ejs
│   └── admin.ejs
├── partials/
│   ├── header.ejs
│   ├── footer.ejs
│   └── navigation.ejs
└── pages/
```

```
|   |   | home.ejs  
|   |   | about.ejs  
|   |   | contact.ejs  
|   | components/  
|   |   | post-card.ejs  
|   |   | user-profile.ejs
```

### 3. Security Considerations

```
// Always escape user input  
<%= userInput %> // Safe - automatically escaped  
<%- userInput %> // Dangerous - unescaped  
  
// Validate data before rendering  
const sanitizeHtml = require('sanitize-html');  
app.locals.sanitize = (html) => sanitizeHtml(html);
```

### 4. Performance Optimization

```
// Enable template caching in production  
if (process.env.NODE_ENV === "production") {  
  app.set("view cache", true);  
}  
  
// Precompile templates  
const ejs = require("ejs");  
const template = ejs.compile(templateString, { cache: true });
```

### 5. Error Handling

```
// Graceful error handling in templates  
<% try { %>  
  <%= user.profile.name %>  
<% } catch (error) { %>  
  <span class="text-muted">Name not available</span>  
<% } %>
```

## Summary

Template engines are essential tools for server-side rendering:

#### Key Benefits:

- **Dynamic Content:** Inject data into static templates
- **Code Reusability:** Create layouts, partials, and components

- **Separation of Concerns:** Keep presentation separate from logic
- **Security:** Automatic escaping prevents XSS attacks

#### **EJS Advantages:**

- Familiar JavaScript syntax
- Easy learning curve
- Good performance
- Extensive ecosystem

#### **Pug Advantages:**

- Clean, concise syntax
- Powerful features (mixins, inheritance)
- Built-in filters and helpers
- Excellent for new projects

#### **Best Practices:**

- Choose the right engine for your team and project
- Organize templates logically
- Always escape user input
- Enable caching in production
- Handle errors gracefully

Template engines enable you to build dynamic, maintainable web applications with server-side rendering capabilities. Next, we'll explore RESTful API design principles for building robust web services.

## RESTful API Design Principles

---

### Overview

REST (Representational State Transfer) is an architectural style for designing networked applications, particularly web services. RESTful APIs have become the standard for building web services due to their simplicity, scalability, and stateless nature. Understanding REST principles is crucial for creating APIs that are intuitive, maintainable, and follow industry best practices.

### Key Concepts

#### REST Fundamentals

**REST (Representational State Transfer):** An architectural style that defines a set of constraints for creating web services.

#### **Key Principles:**

1. **Stateless:** Each request contains all information needed to process it
2. **Client-Server:** Separation of concerns between client and server
3. **Cacheable:** Responses should be cacheable when appropriate
4. **Uniform Interface:** Consistent interface for all resources



5. **Layered System**: Architecture can be composed of hierarchical layers
6. **Code on Demand** (optional): Server can send executable code to client

## HTTP Methods and Their Usage

- **GET**: Retrieve data (safe and idempotent)
- **POST**: Create new resources
- **PUT**: Update/replace entire resource (idempotent)
- **PATCH**: Partial update of resource
- **DELETE**: Remove resource (idempotent)
- **HEAD**: Get headers only (like GET but no body)
- **OPTIONS**: Get allowed methods for resource

## HTTP Status Codes

### Success (2xx):

- **200 OK**: Request successful
- **201 Created**: Resource created successfully
- **204 No Content**: Successful request with no response body

### Client Error (4xx):

- **400 Bad Request**: Invalid request syntax
- **401 Unauthorized**: Authentication required
- **403 Forbidden**: Access denied
- **404 Not Found**: Resource not found
- **409 Conflict**: Request conflicts with current state
- **422 Unprocessable Entity**: Validation errors

### Server Error (5xx):

- **500 Internal Server Error**: Generic server error
- **502 Bad Gateway**: Invalid response from upstream server
- **503 Service Unavailable**: Server temporarily unavailable

## Resource Naming Conventions

- Use **nouns**, not verbs: `/users` not `/getUsers`
- Use **plural nouns**: `/users` not `/user`
- Use **hierarchical structure**: `/users/123/posts`
- Use **lowercase**: `/users` not `/Users`
- Use **hyphens** for multi-word resources: `/user-profiles`

## Example Code

### Basic RESTful API Structure

```
// models/User.js - User Model
class User {
  constructor(data) {
    this.id = data.id;
    this.username = data.username;
    this.email = data.email;
    this.firstName = data.firstName;
    this.lastName = data.lastName;
    this.role = data.role || "user";
    this.active = data.active !== undefined ? data.active : true;
    this.createdAt = data.createdAt || new Date();
    this.updatedAt = data.updatedAt || new Date();
  }

  // Validation method
  validate() {
    const errors = [];

    if (!this.username || this.username.length < 3) {
      errors.push("Username must be at least 3 characters long");
    }

    if (!this.email || !this.isValidEmail(this.email)) {
      errors.push("Valid email is required");
    }

    if (!this.firstName || this.firstName.length < 1) {
      errors.push("First name is required");
    }

    if (!this.lastName || this.lastName.length < 1) {
      errors.push("Last name is required");
    }

    return errors;
  }

  isValidEmail(email) {
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return emailRegex.test(email);
  }

  // Get public representation (exclude sensitive data)
  toPublic() {
    return {
      id: this.id,
      username: this.username,
      email: this.email,
      firstName: this.firstName,
      lastName: this.lastName,
      fullName: `${this.firstName} ${this.lastName}`,
      role: this.role,
      active: this.active,
    };
  }
}
```

```

        createdAt: this.createdAt,
        updatedAt: this.updatedAt,
    };
}
}

module.exports = User;

```

```

// services/UserService.js - Business Logic Layer
const User = require("../models/User");

class UserService {
  constructor() {
    // In-memory storage (use database in production)
    this.users = [
      new User({
        id: 1,
        username: "johndoe",
        email: "john@example.com",
        firstName: "John",
        lastName: "Doe",
        role: "admin",
      }),
      new User({
        id: 2,
        username: "janedoe",
        email: "jane@example.com",
        firstName: "Jane",
        lastName: "Doe",
        role: "user",
      }),
    ];
    this.nextId = 3;
  }

  // Get all users with filtering and pagination
  async getAllUsers(options = {}) {
    const {
      page = 1,
      limit = 10,
      sortBy = "createdAt",
      sortOrder = "desc",
      role,
      active,
      search,
    } = options;

    let filteredUsers = [...this.users];

    // Apply filters
    if (role) {

```

```
    filteredUsers = filteredUsers.filter((user) => user.role === role);
  }

  if (active !== undefined) {
    filteredUsers = filteredUsers.filter((user) => user.active === active);
  }

  if (search) {
    const searchLower = search.toLowerCase();
    filteredUsers = filteredUsers.filter(
      (user) =>
        user.username.toLowerCase().includes(searchLower) ||
        user.email.toLowerCase().includes(searchLower) ||
        user.firstName.toLowerCase().includes(searchLower) ||
        user.lastName.toLowerCase().includes(searchLower)
    );
  }

  // Apply sorting
  filteredUsers.sort((a, b) => {
    let aVal = a[sortBy];
    let bVal = b[sortBy];

    if (sortBy === "createdAt" || sortBy === "updatedAt") {
      aVal = new Date(aVal);
      bVal = new Date(bVal);
    }

    if (sortOrder === "desc") {
      return bVal > aVal ? 1 : -1;
    } else {
      return aVal > bVal ? 1 : -1;
    }
  });

  // Apply pagination
  const total = filteredUsers.length;
  const totalPages = Math.ceil(total / limit);
  const offset = (page - 1) * limit;
  const paginatedUsers = filteredUsers.slice(offset, offset + limit);

  return {
    users: paginatedUsers.map((user) => user.toPublic()),
    pagination: {
      page: parseInt(page),
      limit: parseInt(limit),
      total,
      totalPages,
      hasNext: page < totalPages,
      hasPrev: page > 1,
    },
  };
}
```

```
// Get user by ID
async getUserId(id) {
  const user = this.users.find((u) => u.id === parseInt(id));
  return user ? user.toPublic() : null;
}

// Create new user
async createUser(userData) {
  // Check if username or email already exists
  const existingUser = this.users.find(
    (u) => u.username === userData.username || u.email === userData.email
  );

  if (existingUser) {
    throw new Error("Username or email already exists");
  }

  const user = new User({
    ...userData,
    id: this.nextId++,
  });

  const validationErrors = user.validate();
  if (validationErrors.length > 0) {
    throw new Error(`Validation failed: ${validationErrors.join(", ")}`);
  }

  this.users.push(user);
  return user.toPublic();
}

// Update user
async updateUser(id, updateData) {
  const userIndex = this.users.findIndex((u) => u.id === parseInt(id));

  if (userIndex === -1) {
    return null;
  }

  // Check for conflicts with other users
  if (updateData.username || updateData.email) {
    const conflictUser = this.users.find(
      (u) =>
        u.id !== parseInt(id) &&
        (u.username === updateData.username || u.email === updateData.email)
    );

    if (conflictUser) {
      throw new Error("Username or email already exists");
    }
  }

  // Update user data
  const updatedUser = new User({
```

```
        ...this.users[userIndex],
        ...updateData,
        id: parseInt(id),
        updatedAt: new Date(),
    });

    const validationErrors = updatedUser.validate();
    if (validationErrors.length > 0) {
        throw new Error(`Validation failed: ${validationErrors.join(", ")}`);
    }

    this.users[userIndex] = updatedUser;
    return updatedUser.toPublic();
}

// Delete user
async deleteUser(id) {
    const userIndex = this.users.findIndex((u) => u.id === parseInt(id));

    if (userIndex === -1) {
        return null;
    }

    const deletedUser = this.users.splice(userIndex, 1)[0];
    return deletedUser.toPublic();
}

// Soft delete (deactivate) user
async deactivateUser(id) {
    const userIndex = this.users.findIndex((u) => u.id === parseInt(id));

    if (userIndex === -1) {
        return null;
    }

    this.users[userIndex].active = false;
    this.users[userIndex].updatedAt = new Date();

    return this.users[userIndex].toPublic();
}
}

module.exports = UserService;
```

```
// controllers/UserController.js - Request/Response Handling
const UserService = require("../services/UserService");

class UserController {
    constructor() {
        this.userService = new UserService();
    }
}
```

```
// GET /api/users
async getUsers(req, res) {
  try {
    const options = {
      page: req.query.page,
      limit: req.query.limit,
      sortBy: req.query.sortBy,
      sortOrder: req.query.sortOrder,
      role: req.query.role,
      active:
        req.query.active === "true"
          ? true
          : req.query.active === "false"
          ? false
          : undefined,
      search: req.query.search,
    };

    const result = await this.userService.getAllUsers(options);

    res.json({
      success: true,
      data: result.users,
      pagination: result.pagination,
      message: "Users retrieved successfully",
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      error: "Internal server error",
      message: error.message,
    });
  }
}

// GET /api/users/:id
async getUserById(req, res) {
  try {
    const user = await this.userService.getUserById(req.params.id);

    if (!user) {
      return res.status(404).json({
        success: false,
        error: "User not found",
        message: `User with ID ${req.params.id} does not exist`,
      });
    }

    res.json({
      success: true,
      data: user,
      message: "User retrieved successfully",
    });
  }
}
```

```
    } catch (error) {
      res.status(500).json({
        success: false,
        error: "Internal server error",
        message: error.message,
      });
    }
  }
}

// POST /api/users
async createUser(req, res) {
  try {
    const user = await this.userService.createUser(req.body);

    res.status(201).json({
      success: true,
      data: user,
      message: "User created successfully",
    });
  } catch (error) {
    if (error.message.includes("already exists")) {
      return res.status(409).json({
        success: false,
        error: "Conflict",
        message: error.message,
      });
    }

    if (error.message.includes("Validation failed")) {
      return res.status(422).json({
        success: false,
        error: "Validation error",
        message: error.message,
      });
    }

    res.status(500).json({
      success: false,
      error: "Internal server error",
      message: error.message,
    });
  }
}

// PUT /api/users/:id
async updateUser(req, res) {
  try {
    const user = await this.userService.updateUser(req.params.id, req.body);

    if (!user) {
      return res.status(404).json({
        success: false,
        error: "User not found",
        message: `User with ID ${req.params.id} does not exist`,
      });
    }
  }
}
```



```
    });
  }

  res.json({
    success: true,
    data: user,
    message: "User updated successfully",
  });
} catch (error) {
  if (error.message.includes("already exists")) {
    return res.status(409).json({
      success: false,
      error: "Conflict",
      message: error.message,
    });
  }

  if (error.message.includes("Validation failed")) {
    return res.status(422).json({
      success: false,
      error: "Validation error",
      message: error.message,
    });
  }

  res.status(500).json({
    success: false,
    error: "Internal server error",
    message: error.message,
  });
}
}

// PATCH /api/users/:id
async patchUser(req, res) {
  try {
    const user = await this.userService.updateUser(req.params.id, req.body);

    if (!user) {
      return res.status(404).json({
        success: false,
        error: "User not found",
        message: `User with ID ${req.params.id} does not exist`,
      });
    }

    res.json({
      success: true,
      data: user,
      message: "User updated successfully",
    });
  } catch (error) {
    if (error.message.includes("already exists")) {
      return res.status(409).json({
```

```
        success: false,
        error: "Conflict",
        message: error.message,
    });
}

if (error.message.includes("Validation failed")) {
    return res.status(422).json({
        success: false,
        error: "Validation error",
        message: error.message,
    });
}

res.status(500).json({
    success: false,
    error: "Internal server error",
    message: error.message,
});
}
}

// DELETE /api/users/:id
async deleteUser(req, res) {
    try {
        const user = await this.userService.deleteUser(req.params.id);

        if (!user) {
            return res.status(404).json({
                success: false,
                error: "User not found",
                message: `User with ID ${req.params.id} does not exist`,
            });
        }

        res.json({
            success: true,
            data: user,
            message: "User deleted successfully",
        });
    } catch (error) {
        res.status(500).json({
            success: false,
            error: "Internal server error",
            message: error.message,
        });
    }
}

// PATCH /api/users/:id/deactivate
async deactivateUser(req, res) {
    try {
        const user = await this.userService.deactivateUser(req.params.id);
```

```

    if (!user) {
      return res.status(404).json({
        success: false,
        error: "User not found",
        message: `User with ID ${req.params.id} does not exist`,
      });
    }

    res.json({
      success: true,
      data: user,
      message: "User deactivated successfully",
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      error: "Internal server error",
      message: error.message,
    });
  }
}
}

module.exports = UserController;

```

## RESTful Routes Implementation

```

// routes/users.js - User Routes
const express = require("express");
const UserController = require("../controllers/UserController");
const router = express.Router();

const userController = new UserController();

// Middleware for request logging
router.use((req, res, next) => {
  console.log(`${new Date().toISOString()} - ${req.method} ${req.originalUrl}`);
  next();
});

// Validation middleware
const validateUserInput = (req, res, next) => {
  const { method } = req;
  const { body } = req;

  if (method === "POST") {
    const required = ["username", "email", "firstName", "lastName"];
    const missing = required.filter((field) => !body[field]);

    if (missing.length > 0) {
      return res.status(400).json({

```

```
        success: false,
        error: "Bad Request",
        message: `Missing required fields: ${missing.join(", ")}` ,
    });
}
}

next();
};

// Routes following REST conventions

// GET /api/users - Get all users
router.get("/", userController.getUsers.bind(userController));

// GET /api/users/:id - Get specific user
router.get("/:id", userController.getUserById.bind(userController));

// POST /api/users - Create new user
router.post(
    "/",
    validateUserInput,
    userController.createUser.bind(userController)
);

// PUT /api/users/:id - Update entire user
router.put(
   ("/:id",
    validateUserInput,
    userController.updateUser.bind(userController)
);

// PATCH /api/users/:id - Partial update
router.patch("/:id", userController.patchUser.bind(userController));

// DELETE /api/users/:id - Delete user
router.delete("/:id", userController.deleteUser.bind(userController));

// PATCH /api/users/:id/deactivate - Deactivate user (custom action)
router.patch(
   ("/:id/deactivate",
    userController.deactivateUser.bind(userController)
);

// OPTIONS /api/users - Get allowed methods
router.options("/", (req, res) => {
    res.set({
        Allow: "GET, POST, OPTIONS",
        "Access-Control-Allow-Methods": "GET, POST, OPTIONS",
    });
    res.status(200).end();
});

router.options("/:id", (req, res) => {
```

```
res.set({
  Allow: "GET, PUT, PATCH, DELETE, OPTIONS",
  "Access-Control-Allow-Methods": "GET, PUT, PATCH, DELETE, OPTIONS",
});
res.status(200).end();
});

module.exports = router;
```

## Complete RESTful API Application

```
// app.js - Main Application
const express = require("express");
const cors = require("cors");
const helmet = require("helmet");
const morgan = require("morgan");
const rateLimit = require("express-rate-limit");

const app = express();

// Security middleware
app.use(helmet());
app.use(
  cors({
    origin: process.env.ALLOWED_ORIGINS?.split(",") || [
      "http://localhost:3000",
    ],
    credentials: true,
  })
);

// Rate limiting
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests per windowMs
  message: {
    success: false,
    error: "Too many requests",
    message: "Rate limit exceeded. Please try again later.",
  },
});
app.use("/api/", limiter);

// Logging
app.use(morgan("combined"));

// Body parsing
app.use(express.json({ limit: "10mb" }));
app.use(express.urlencoded({ extended: true, limit: "10mb" }));

// API versioning
```

```
const v1Router = express.Router();

// Import route modules
const userRoutes = require("./routes/users");
const postRoutes = require("./routes/posts");
const categoryRoutes = require("./routes/categories");

// Mount routes
v1Router.use("/users", userRoutes);
v1Router.use("/posts", postRoutes);
v1Router.use("/categories", categoryRoutes);

// API info endpoint
v1Router.get("/", (req, res) => {
  res.json({
    success: true,
    message: "RESTful API v1",
    version: "1.0.0",
    endpoints: {
      users: "/api/v1/users",
      posts: "/api/v1/posts",
      categories: "/api/v1/categories",
    },
    documentation: "/api/v1/docs",
    timestamp: new Date().toISOString(),
  });
});

// API documentation endpoint
v1Router.get("/docs", (req, res) => {
  res.json({
    success: true,
    documentation: {
      version: "1.0.0",
      baseUrl: "/api/v1",
      authentication: "Bearer token required for protected endpoints",
      rateLimit: "100 requests per 15 minutes per IP",
      endpoints: {
        users: {
          "GET /users": {
            description: "Get all users",
            parameters: {
              page: "Page number (default: 1)",
              limit: "Items per page (default: 10)",
              sortBy: "Sort field (default: createdAt)",
              sortOrder: "Sort order: asc|desc (default: desc)",
              role: "Filter by role",
              active: "Filter by active status: true|false",
              search: "Search in username, email, firstName, lastName",
            },
            response: "200 OK with users array and pagination info",
          },
          "GET /users/:id": {
            description: "Get user by ID",
          },
        },
      },
    },
  });
});
```

```

        parameters: { id: "User ID" },
        response: "200 OK with user object or 404 Not Found",
    },
    "POST /users": {
        description: "Create new user",
        body: {
            username: "string (required)",
            email: "string (required)",
            firstName: "string (required)",
            lastName: "string (required)",
            role: "string (optional, default: user)",
        },
        response: "201 Created with user object",
    },
    "PUT /users/:id": {
        description: "Update entire user",
        parameters: { id: "User ID" },
        body: "Complete user object",
        response: "200 OK with updated user or 404 Not Found",
    },
    "PATCH /users/:id": {
        description: "Partial user update",
        parameters: { id: "User ID" },
        body: "Partial user object",
        response: "200 OK with updated user or 404 Not Found",
    },
    "DELETE /users/:id": {
        description: "Delete user",
        parameters: { id: "User ID" },
        response: "200 OK with deleted user or 404 Not Found",
    },
},
},
statusCodes: {
    200: "OK - Request successful",
    201: "Created - Resource created successfully",
    400: "Bad Request - Invalid request syntax",
    401: "Unauthorized - Authentication required",
    403: "Forbidden - Access denied",
    404: "Not Found - Resource not found",
    409: "Conflict - Request conflicts with current state",
    422: "Unprocessable Entity - Validation errors",
    429: "Too Many Requests - Rate limit exceeded",
    500: "Internal Server Error - Server error",
},
},
});
});

// Mount API version
app.use("/api/v1", v1Router);

// Root endpoint
app.get("/", (req, res) => {

```

```
res.json({
  success: true,
  message: "RESTful API Server",
  version: "1.0.0",
  api: {
    v1: "/api/v1",
    documentation: "/api/v1/docs",
  },
  status: "healthy",
  timestamp: new Date().toISOString(),
});
});

// Health check endpoint
app.get("/health", (req, res) => {
  res.json({
    status: "healthy",
    timestamp: new Date().toISOString(),
    uptime: process.uptime(),
    memory: process.memoryUsage(),
    version: "1.0.0",
  });
});

// 404 handler
app.use("*", (req, res) => {
  res.status(404).json({
    success: false,
    error: "Not Found",
    message: `Route ${req.method} ${req.originalUrl} not found`,
    availableEndpoints: {
      api: "/api/v1",
      documentation: "/api/v1/docs",
      health: "/health",
    },
  });
});

// Global error handler
app.use((err, req, res, next) => {
  console.error("Global error handler:", err);

  // Handle specific error types
  if (err.type === "entity.parse.failed") {
    return res.status(400).json({
      success: false,
      error: "Bad Request",
      message: "Invalid JSON in request body",
    });
  }

  if (err.type === "entity.too.large") {
    return res.status(413).json({
      success: false,
```



```

        error: "Payload Too Large",
        message: "Request body exceeds size limit",
    });
}

res.status(err.status || 500).json({
    success: false,
    error: err.status === 500 ? "Internal Server Error" : err.name || "Error",
    message:
        process.env.NODE_ENV === "production"
            ? "An error occurred while processing your request"
            : err.message,
    ...(process.env.NODE_ENV !== "production" && { stack: err.stack }),
});
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`🚀 RESTful API Server running on port ${PORT}`);
    console.log(`📖 API Documentation: http://localhost:${PORT}/api/v1/docs`);
    console.log(`🏥 Health Check: http://localhost:${PORT}/health`);
});

module.exports = app;

```

## Real-World Use Case

### E-commerce Product API

```

// models/Product.js
class Product {
    constructor(data) {
        this.id = data.id;
        this.name = data.name;
        this.description = data.description;
        this.price = parseFloat(data.price);
        this.category = data.category;
        this.brand = data.brand;
        this.sku = data.sku;
        this.stock = parseInt(data.stock) || 0;
        this.images = data.images || [];
        this.specifications = data.specifications || {};
        this.tags = data.tags || [];
        this.active = data.active !== undefined ? data.active : true;
        this.featured = data.featured || false;
        this.rating = data.rating || 0;
        this.reviewCount = data.reviewCount || 0;
        this.createdAt = data.createdAt || new Date();
        this.updatedAt = data.updatedAt || new Date();
    }
}

```

```
validate() {
  const errors = [];

  if (!this.name || this.name.length < 3) {
    errors.push("Product name must be at least 3 characters");
  }

  if (!this.description || this.description.length < 10) {
    errors.push("Description must be at least 10 characters");
  }

  if (!this.price || this.price <= 0) {
    errors.push("Price must be greater than 0");
  }

  if (!this.category) {
    errors.push("Category is required");
  }

  if (!this.sku) {
    errors.push("SKU is required");
  }

  return errors;
}

toPublic() {
  return {
    id: this.id,
    name: this.name,
    description: this.description,
    price: this.price,
    category: this.category,
    brand: this.brand,
    sku: this.sku,
    stock: this.stock,
    images: this.images,
    specifications: this.specifications,
    tags: this.tags,
    active: this.active,
    featured: this.featured,
    rating: this.rating,
    reviewCount: this.reviewCount,
    availability: this.stock > 0 ? "in-stock" : "out-of-stock",
    createdAt: this.createdAt,
    updatedAt: this.updatedAt,
  };
}
}

module.exports = Product;
```

```
// routes/products.js - Product API Routes
const express = require("express");
const Product = require("../models/Product");
const router = express.Router();

// Sample product data
let products = [
  new Product({
    id: 1,
    name: 'MacBook Pro 16"',
    description: "Powerful laptop for professionals with M2 Pro chip",
    price: 2499.99,
    category: "Electronics",
    brand: "Apple",
    sku: "MBP16-M2PRO-512",
    stock: 15,
    images: ["macbook-1.jpg", "macbook-2.jpg"],
    specifications: {
      processor: "M2 Pro",
      memory: "16GB",
      storage: "512GB SSD",
      display: "16-inch Retina",
    },
    tags: ["laptop", "apple", "professional"],
    featured: true,
    rating: 4.8,
    reviewCount: 127,
  }),
  new Product({
    id: 2,
    name: "iPhone 15 Pro",
    description: "Latest iPhone with titanium design and A17 Pro chip",
    price: 999.99,
    category: "Electronics",
    brand: "Apple",
    sku: "IP15PRO-128-TIT",
    stock: 25,
    images: ["iphone-1.jpg", "iphone-2.jpg"],
    specifications: {
      processor: "A17 Pro",
      storage: "128GB",
      camera: "48MP Pro camera system",
      display: "6.1-inch Super Retina XDR",
    },
    tags: ["smartphone", "apple", "pro"],
    featured: true,
    rating: 4.9,
    reviewCount: 89,
  }),
];
let nextId = 3;

// GET /api/products - Get all products with advanced filtering
```

```
router.get("/", (req, res) => {
  try {
    const {
      page = 1,
      limit = 10,
      sortBy = "createdAt",
      sortOrder = "desc",
      category,
      brand,
      minPrice,
      maxPrice,
      inStock,
      featured,
      search,
      tags,
    } = req.query;

    let filteredProducts = products.filter((p) => p.active);

    // Apply filters
    if (category) {
      filteredProducts = filteredProducts.filter(
        (p) => p.category.toLowerCase() === category.toLowerCase()
      );
    }

    if (brand) {
      filteredProducts = filteredProducts.filter(
        (p) => p.brand.toLowerCase() === brand.toLowerCase()
      );
    }

    if (minPrice) {
      filteredProducts = filteredProducts.filter(
        (p) => p.price >= parseFloat(minPrice)
      );
    }

    if (maxPrice) {
      filteredProducts = filteredProducts.filter(
        (p) => p.price <= parseFloat(maxPrice)
      );
    }

    if (inStock === "true") {
      filteredProducts = filteredProducts.filter((p) => p.stock > 0);
    }

    if (featured === "true") {
      filteredProducts = filteredProducts.filter((p) => p.featured);
    }

    if (search) {
      const searchLower = search.toLowerCase();
```

```
    filteredProducts = filteredProducts.filter(
      (p) =>
        p.name.toLowerCase().includes(searchLower) ||
        p.description.toLowerCase().includes(searchLower) ||
        p.brand.toLowerCase().includes(searchLower) ||
        p.tags.some((tag) => tag.toLowerCase().includes(searchLower))
    );
  }

  if (tags) {
    const tagList = tags.split(",").map((tag) => tag.trim().toLowerCase());
    filteredProducts = filteredProducts.filter((p) =>
      tagList.some((tag) => p.tags.map((t) => t.toLowerCase()).includes(tag))
    );
  }

  // Apply sorting
  filteredProducts.sort((a, b) => {
    let aVal = a[sortBy];
    let bVal = b[sortBy];

    if (sortBy === "createdAt" || sortBy === "updatedAt") {
      aVal = new Date(aVal);
      bVal = new Date(bVal);
    }

    if (sortOrder === "desc") {
      return bVal > aVal ? 1 : -1;
    } else {
      return aVal > bVal ? 1 : -1;
    }
  });

  // Apply pagination
  const total = filteredProducts.length;
  const totalPages = Math.ceil(total / limit);
  const offset = (page - 1) * limit;
  const paginatedProducts = filteredProducts.slice(
    offset,
    offset + parseInt(limit)
  );

  // Get filter options for frontend
  const filterOptions = {
    categories: [...new Set(products.map((p) => p.category))],
    brands: [...new Set(products.map((p) => p.brand))],
    priceRange: {
      min: Math.min(...products.map((p) => p.price)),
      max: Math.max(...products.map((p) => p.price)),
    },
    tags: [...new Set(products.flatMap((p) => p.tags))],
  };

  res.json({
```

```
        success: true,
        data: paginatedProducts.map((p) => p.toPublic()),
        pagination: {
            page: parseInt(page),
            limit: parseInt(limit),
            total,
            totalPages,
            hasNext: page < totalPages,
            hasPrev: page > 1,
        },
        filters: {
            applied: {
                category,
                brand,
                minPrice,
                maxPrice,
                inStock,
                featured,
                search,
                tags,
            },
            available: filterOptions,
        },
        message: "Products retrieved successfully",
    });
} catch (error) {
    res.status(500).json({
        success: false,
        error: "Internal server error",
        message: error.message,
    });
}
});

// GET /api/products/featured - Get featured products
router.get("/featured", (req, res) => {
    try {
        const featuredProducts = products
            .filter((p) => p.featured && p.active)
            .sort((a, b) => b.rating - a.rating)
            .slice(0, parseInt(req.query.limit) || 6);

        res.json({
            success: true,
            data: featuredProducts.map((p) => p.toPublic()),
            count: featuredProducts.length,
            message: "Featured products retrieved successfully",
        });
    } catch (error) {
        res.status(500).json({
            success: false,
            error: "Internal server error",
            message: error.message,
        });
    }
});
```

```
    }
  });

  // GET /api/products/categories - Get all categories
  router.get("/categories", (req, res) => {
    try {
      const categories = [...new Set(products.map((p) => p.category))];
      const categoryStats = categories.map((category) => {
        const categoryProducts = products.filter(
          (p) => p.category === category && p.active
        );
        return {
          name: category,
          count: categoryProducts.length,
          averagePrice:
            categoryProducts.reduce((sum, p) => sum + p.price, 0) /
            categoryProducts.length,
        };
      });
    } catch (error) {
      res.status(500).json({
        success: false,
        error: "Internal server error",
        message: error.message,
      });
    }
  });

  // GET /api/products/:id - Get product by ID
  router.get("/:id", (req, res) => {
    try {
      const product = products.find(
        (p) => p.id === parseInt(req.params.id) && p.active
      );

      if (!product) {
        return res.status(404).json({
          success: false,
          error: "Product not found",
          message: `Product with ID ${req.params.id} does not exist`,
        });
      }

      // Get related products
      const relatedProducts = products
        .filter(
          (p) =>
            p.id !== product.id && p.category === product.category && p.active
        )
    }
  });
```

```
    )
    .sort((a, b) => b.rating - a.rating)
    .slice(0, 4);

res.json({
  success: true,
  data: {
    product: product.toPublic(),
    related: relatedProducts.map((p) => p.toPublic()),
  },
  message: "Product retrieved successfully",
});
} catch (error) {
  res.status(500).json({
    success: false,
    error: "Internal server error",
    message: error.message,
  });
}
});

// POST /api/products - Create new product
router.post("/", (req, res) => {
  try {
    // Check if SKU already exists
    const existingProduct = products.find((p) => p.sku === req.body.sku);
    if (existingProduct) {
      return res.status(409).json({
        success: false,
        error: "Conflict",
        message: "Product with this SKU already exists",
      });
    }

    const product = new Product({
      ...req.body,
      id: nextId++,
    });

    const validationErrors = product.validate();
    if (validationErrors.length > 0) {
      return res.status(422).json({
        success: false,
        error: "Validation error",
        message: `Validation failed: ${validationErrors.join(", ")}`,
      });
    }

    products.push(product);

    res.status(201).json({
      success: true,
      data: product.toPublic(),
      message: "Product created successfully",
    });
  }
});
```



```
});  
} catch (error) {  
  res.status(500).json({  
    success: false,  
    error: "Internal server error",  
    message: error.message,  
  });  
}  
});  
  
// PUT /api/products/:id - Update entire product  
router.put("/:id", (req, res) => {  
  try {  
    const productIndex = products.findIndex(  
      (p) => p.id === parseInt(req.params.id)  
    );  
  
    if (productIndex === -1) {  
      return res.status(404).json({  
        success: false,  
        error: "Product not found",  
        message: `Product with ID ${req.params.id} does not exist`,  
      });  
    }  
  
    // Check SKU conflict  
    if (req.body.sku) {  
      const conflictProduct = products.find(  
        (p) => p.id !== parseInt(req.params.id) && p.sku === req.body.sku  
      );  
      if (conflictProduct) {  
        return res.status(409).json({  
          success: false,  
          error: "Conflict",  
          message: "Product with this SKU already exists",  
        });  
      }  
    }  
  
    const updatedProduct = new Product({  
      ...req.body,  
      id: parseInt(req.params.id),  
      updatedAt: new Date(),  
    });  
  
    const validationErrors = updatedProduct.validate();  
    if (validationErrors.length > 0) {  
      return res.status(422).json({  
        success: false,  
        error: "Validation error",  
        message: `Validation failed: ${validationErrors.join(", ")}`,  
      });  
    }  
  }  
});
```

```
    products[productIndex] = updatedProduct;

    res.json({
      success: true,
      data: updatedProduct.toPublic(),
      message: "Product updated successfully",
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      error: "Internal server error",
      message: error.message,
    });
  }
});

// PATCH /api/products/:id/stock - Update stock only
router.patch("/:id/stock", (req, res) => {
  try {
    const productIndex = products.findIndex(
      (p) => p.id === parseInt(req.params.id)
    );

    if (productIndex === -1) {
      return res.status(404).json({
        success: false,
        error: "Product not found",
        message: `Product with ID ${req.params.id} does not exist`,
      });
    }

    const { stock } = req.body;

    if (stock === undefined || isNaN(parseInt(stock)) || parseInt(stock) < 0) {
      return res.status(400).json({
        success: false,
        error: "Bad Request",
        message: "Valid stock quantity (>= 0) is required",
      });
    }

    products[productIndex].stock = parseInt(stock);
    products[productIndex].updatedAt = new Date();

    res.json({
      success: true,
      data: products[productIndex].toPublic(),
      message: "Product stock updated successfully",
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      error: "Internal server error",
      message: error.message,
    });
  }
});
```

```

    });
  }
});

// DELETE /api/products/:id - Delete product (soft delete)
router.delete("/:id", (req, res) => {
  try {
    const productIndex = products.findIndex(
      (p) => p.id === parseInt(req.params.id)
    );

    if (productIndex === -1) {
      return res.status(404).json({
        success: false,
        error: "Product not found",
        message: `Product with ID ${req.params.id} does not exist`,
      });
    }

    // Soft delete - mark as inactive
    products[productIndex].active = false;
    products[productIndex].updatedAt = new Date();

    res.json({
      success: true,
      data: products[productIndex].toPublic(),
      message: "Product deleted successfully",
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      error: "Internal server error",
      message: error.message,
    });
  }
});


module.exports = router;

```

## Best Practices

### 1. Use Proper HTTP Methods

```

//  Correct usage
GET / api / users; // Get all users
GET / api / users / 123; // Get specific user
POST / api / users; // Create new user
PUT / api / users / 123; // Update entire user
PATCH / api / users / 123; // Partial update
DELETE / api / users / 123; // Delete user

```

```
// ✗ Incorrect usage
GET / api / getUsers; // Don't use verbs in URLs
POST /
  api /
  users /
  delete (
    // Use DELETE method instead
    GET
  ) /
  api /
  users /
  create; // Use POST for creation
```

## 2. Implement Consistent Response Format

```
// Success response format
{
  "success": true,
  "data": { /* response data */ },
  "message": "Operation completed successfully",
  "pagination": { /* pagination info if applicable */ }
}

// Error response format
{
  "success": false,
  "error": "Error Type",
  "message": "Detailed error message",
  "details": { /* additional error details */ }
}
```

## 3. Use Appropriate Status Codes

```
// Success responses
res.status(200).json(data); // OK - successful GET, PUT, PATCH
res.status(201).json(data); // Created - successful POST
res.status(204).end(); // No Content - successful DELETE

// Client error responses
res.status(400).json(error); // Bad Request - invalid syntax
res.status(401).json(error); // Unauthorized - authentication required
res.status(403).json(error); // Forbidden - access denied
res.status(404).json(error); // Not Found - resource doesn't exist
res.status(409).json(error); // Conflict - resource already exists
res.status(422).json(error); // Unprocessable Entity - validation errors

// Server error responses
res.status(500).json(error); // Internal Server Error
```

## 4. Implement Proper Error Handling

```
// Async error wrapper
const asyncHandler = (fn) => (req, res, next) => {
  Promise.resolve(fn(req, res, next)).catch(next);
};

// Usage
router.get(
  "/users",
  asyncHandler(async (req, res) => {
    const users = await userService.getAllUsers();
    res.json({ success: true, data: users });
  })
);
```

## 5. Add Input Validation

```
const { body, validationResult } = require("express-validator");

const validateUser = [
  body("username")
    .isLength({ min: 3 })
    .withMessage("Username must be at least 3 characters"),
  body("email").isEmail().withMessage("Valid email is required"),
  body("firstName").notEmpty().withMessage("First name is required"),
  body("lastName").notEmpty().withMessage("Last name is required"),

  (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(422).json({
        success: false,
        error: "Validation error",
        details: errors.array(),
      });
    }
    next();
  },
];
```

## Summary

RESTful API design principles provide a foundation for building scalable, maintainable web services:

### Core Principles:

- **Stateless:** Each request is independent
- **Resource-based:** URLs represent resources, not actions

- **HTTP methods:** Use appropriate methods for different operations
- **Status codes:** Return meaningful HTTP status codes
- **Consistent format:** Standardize request/response structures

**Best Practices:**

- Use nouns in URLs, not verbs
- Implement proper error handling
- Add input validation
- Use consistent response formats
- Follow HTTP status code conventions
- Implement pagination for large datasets
- Add filtering and sorting capabilities

**Benefits:**

- **Predictable:** Follows established conventions
- **Scalable:** Stateless nature enables horizontal scaling
- **Cacheable:** Responses can be cached for performance
- **Maintainable:** Clear structure and separation of concerns
- **Interoperable:** Works with any HTTP client

Mastering RESTful API design is essential for building modern web applications. Next, we'll explore CRUD operations with Express.js in more detail, building upon these REST principles.

## CRUD Operations with Express.js

---

### Overview

CRUD (Create, Read, Update, Delete) operations are the fundamental building blocks of any data-driven application. In Express.js, implementing CRUD operations involves creating endpoints that handle HTTP requests to manipulate data resources. This chapter covers how to build comprehensive CRUD APIs with proper validation, error handling, and real-world patterns.

### Key Concepts

#### CRUD Operations Mapping

**CRUD to HTTP Methods:**

- **Create:** POST requests to create new resources
- **Read:** GET requests to retrieve existing resources
- **Update:** PUT (full update) or PATCH (partial update) requests
- **Delete:** DELETE requests to remove resources

#### Data Persistence Patterns

**In-Memory Storage:** Simple arrays/objects for development and testing **File-Based Storage:** JSON files for lightweight persistence **Database Integration:** SQL/NoSQL databases for production applications

## Validation and Error Handling

**Input Validation:** Ensuring data integrity before processing **Business Logic Validation:** Enforcing domain-specific rules **Error Responses:** Consistent error formatting and appropriate status codes

## Advanced CRUD Features

**Filtering:** Query parameters to filter results **Sorting:** Order results by specific fields **Pagination:** Handle large datasets efficiently **Search:** Text-based search across multiple fields **Bulk Operations:** Handle multiple records in single requests

## Example Code

### Basic CRUD Implementation

```
// models/Task.js - Task Model
class Task {
  constructor(data) {
    this.id = data.id;
    this.title = data.title;
    this.description = data.description || "";
    this.status = data.status || "pending"; // pending, in-progress, completed
    this.priority = data.priority || "medium"; // low, medium, high, urgent
    this.dueDate = data.dueDate ? new Date(data.dueDate) : null;
    this.assignedTo = data.assignedTo || null;
    this.tags = data.tags || [];
    this.createdAt = data.createdAt || new Date();
    this.updatedAt = data.updatedAt || new Date();
    this.completedAt = data.completedAt || null;
  }

  validate() {
    const errors = [];

    if (!this.title || this.title.trim().length < 3) {
      errors.push("Title must be at least 3 characters long");
    }

    if (this.title && this.title.length > 100) {
      errors.push("Title cannot exceed 100 characters");
    }

    if (this.description && this.description.length > 500) {
      errors.push("Description cannot exceed 500 characters");
    }

    const validStatuses = ["pending", "in-progress", "completed"];
    if (!validStatuses.includes(this.status)) {
      errors.push("Status must be one of: pending, in-progress, completed");
    }

    const validPriorities = ["low", "medium", "high", "urgent"];
  }
}
```

```
    if (!validPriorities.includes(this.priority)) {
      errors.push("Priority must be one of: low, medium, high, urgent");
    }

    if (this.dueDate && this.dueDate < new Date()) {
      errors.push("Due date cannot be in the past");
    }

    return errors;
  }

  toJSON() {
    return {
      id: this.id,
      title: this.title,
      description: this.description,
      status: this.status,
      priority: this.priority,
      dueDate: this.dueDate,
      assignedTo: this.assignedTo,
      tags: this.tags,
      createdAt: this.createdAt,
      updatedAt: this.updatedAt,
      completedAt: this.completedAt,
      isOverdue:
        this.dueDate &&
        this.dueDate < new Date() &&
        this.status !== "completed",
    };
  }

  markCompleted() {
    this.status = "completed";
    this.completedAt = new Date();
    this.updatedAt = new Date();
  }

  updateStatus(newStatus) {
    const oldStatus = this.status;
    this.status = newStatus;
    this.updatedAt = new Date();

    if (newStatus === "completed" && oldStatus !== "completed") {
      this.completedAt = new Date();
    } else if (newStatus !== "completed") {
      this.completedAt = null;
    }
  }
}

module.exports = Task;
```



```
// services/TaskService.js - Business Logic Layer
const Task = require("../models/Task");
const fs = require("fs").promises;
const path = require("path");

class TaskService {
  constructor() {
    this.tasks = [];
    this.nextId = 1;
    this.dataFile = path.join(__dirname, "../data/tasks.json");
    this.loadTasks();
  }

  // Load tasks from file
  async loadTasks() {
    try {
      const data = await fs.readFile(this.dataFile, "utf8");
      const tasksData = JSON.parse(data);
      this.tasks = tasksData.tasks.map((taskData) => new Task(taskData));
      this.nextId = tasksData.nextId || 1;
    } catch (error) {
      // File doesn't exist or is invalid, start with empty array
      this.tasks = [];
      this.nextId = 1;
      await this.saveTasks();
    }
  }

  // Save tasks to file
  async saveTasks() {
    try {
      const dataDir = path.dirname(this.dataFile);
      await fs.mkdir(dataDir, { recursive: true });

      const data = {
        tasks: this.tasks.map((task) => task.toJSON()),
        nextId: this.nextId,
        lastUpdated: new Date().toISOString(),
      };

      await fs.writeFile(this.dataFile, JSON.stringify(data, null, 2));
    } catch (error) {
      console.error("Error saving tasks:", error);
      throw new Error("Failed to save tasks");
    }
  }

  // CREATE - Add new task
  async createTask(taskData) {
    const task = new Task({
      ...taskData,
      id: this.nextId++,
    });
  }
```

```
    const validationErrors = task.validate();
    if (validationErrors.length > 0) {
      throw new Error(`Validation failed: ${validationErrors.join(", ")}`);
    }

    this.tasks.push(task);
    await this.saveTasks();

    return task.toJSON();
  }

// READ - Get all tasks with filtering, sorting, and pagination
async getAllTasks(options = {}) {
  const {
    page = 1,
    limit = 10,
    sortBy = "createdAt",
    sortOrder = "desc",
    status,
    priority,
    assignedTo,
    search,
    tags,
    overdue,
    dueDateFrom,
    dueDateTo,
  } = options;

  let filteredTasks = [...this.tasks];

  // Apply filters
  if (status) {
    filteredTasks = filteredTasks.filter((task) => task.status === status);
  }

  if (priority) {
    filteredTasks = filteredTasks.filter(
      (task) => task.priority === priority
    );
  }

  if (assignedTo) {
    filteredTasks = filteredTasks.filter(
      (task) => task.assignedTo === assignedTo
    );
  }

  if (search) {
    const searchLower = search.toLowerCase();
    filteredTasks = filteredTasks.filter(
      (task) =>
        task.title.toLowerCase().includes(searchLower) ||
        task.description.toLowerCase().includes(searchLower) ||

```

```
        (task.assignedTo &&
          task.assignedTo.toLowerCase().includes(searchLower))
      );
    }

    if (tags) {
      const tagList = tags.split(",").map((tag) => tag.trim().toLowerCase());
      filteredTasks = filteredTasks.filter((task) =>
        tagList.some((tag) =>
          task.tags.map((t) => t.toLowerCase()).includes(tag)
        )
      );
    }

    if (overdue === "true") {
      const now = new Date();
      filteredTasks = filteredTasks.filter(
        (task) =>
          task.dueDate && task.dueDate < now && task.status !== "completed"
      );
    }

    if (dueDateFrom) {
      const fromDate = new Date(dueDateFrom);
      filteredTasks = filteredTasks.filter(
        (task) => task.dueDate && task.dueDate >= fromDate
      );
    }

    if (dueDateTo) {
      const toDate = new Date(dueDateTo);
      filteredTasks = filteredTasks.filter(
        (task) => task.dueDate && task.dueDate <= toDate
      );
    }

    // Apply sorting
    filteredTasks.sort((a, b) => {
      let aVal = a[sortBy];
      let bVal = b[sortBy];

      // Handle date fields
      if (
        ["createdAt", "updatedAt", "dueDate", "completedAt"].includes(sortBy)
      ) {
        aVal = aVal ? new Date(aVal) : new Date(0);
        bVal = bVal ? new Date(bVal) : new Date(0);
      }

      // Handle priority sorting
      if (sortBy === "priority") {
        const priorityOrder = { low: 1, medium: 2, high: 3, urgent: 4 };
        aVal = priorityOrder[aVal] || 0;
        bVal = priorityOrder[bVal] || 0;
      }
    });
  }
}
```

```
    }

    if (sortOrder === "desc") {
      return bVal > aVal ? 1 : -1;
    } else {
      return aVal > bVal ? 1 : -1;
    }
  });

  // Apply pagination
  const total = filteredTasks.length;
  const totalPages = Math.ceil(total / limit);
  const offset = (page - 1) * limit;
  const paginatedTasks = filteredTasks.slice(offset, offset + limit);

  // Calculate statistics
  const stats = {
    total: this.tasks.length,
    filtered: total,
    byStatus: {
      pending: this.tasks.filter((t) => t.status === "pending").length,
      "in-progress": this.tasks.filter((t) => t.status === "in-progress")
        .length,
      completed: this.tasks.filter((t) => t.status === "completed").length,
    },
    overdue: this.tasks.filter(
      (t) => t.dueDate && t.dueDate < new Date() && t.status !== "completed"
    ).length,
  };

  return {
    tasks: paginatedTasks.map((task) => task.toJSON()),
    pagination: {
      page: parseInt(page),
      limit: parseInt(limit),
      total,
      totalPages,
      hasNext: page < totalPages,
      hasPrev: page > 1,
    },
    stats,
  };
}

// READ - Get task by ID
async getTaskById(id) {
  const task = this.tasks.find((t) => t.id === parseInt(id));
  return task ? task.toJSON() : null;
}

// UPDATE - Update entire task
async updateTask(id, updateData) {
  const taskIndex = this.tasks.findIndex((t) => t.id === parseInt(id));
```

```
    if (taskIndex === -1) {
      return null;
    }

    const updatedTask = new Task({
      ...this.tasks[taskIndex],
      ...updateData,
      id: parseInt(id),
      updatedAt: new Date(),
    });

    const validationErrors = updatedTask.validate();
    if (validationErrors.length > 0) {
      throw new Error(`Validation failed: ${validationErrors.join(", ")}`);
    }

    this.tasks[taskIndex] = updatedTask;
    await this.saveTasks();

    return updatedTask.toJSON();
  }

  // UPDATE - Partial update
  async patchTask(id, patchData) {
    const taskIndex = this.tasks.findIndex((t) => t.id === parseInt(id));

    if (taskIndex === -1) {
      return null;
    }

    // Handle status change logic
    if (patchData.status && patchData.status !== this.tasks[taskIndex].status) {
      this.tasks[taskIndex].updateStatus(patchData.status);
    }

    // Apply other updates
    Object.keys(patchData).forEach((key) => {
      if (key !== "id" && key !== "createdAt" && patchData[key] !== undefined) {
        this.tasks[taskIndex][key] = patchData[key];
      }
    });

    this.tasks[taskIndex].updatedAt = new Date();

    const validationErrors = this.tasks[taskIndex].validate();
    if (validationErrors.length > 0) {
      throw new Error(`Validation failed: ${validationErrors.join(", ")}`);
    }

    await this.saveTasks();

    return this.tasks[taskIndex].toJSON();
  }
}
```

```
// DELETE - Remove task
async deleteTask(id) {
  const taskIndex = this.tasks.findIndex((t) => t.id === parseInt(id));

  if (taskIndex === -1) {
    return null;
  }

  const deletedTask = this.tasks.splice(taskIndex, 1)[0];
  await this.saveTasks();

  return deletedTask.toJSON();
}

// BULK OPERATIONS
async bulkUpdateTasks(ids, updateData) {
  const updatedTasks = [];
  const errors = [];

  for (const id of ids) {
    try {
      const updated = await this.patchTask(id, updateData);
      if (updated) {
        updatedTasks.push(updated);
      } else {
        errors.push(`Task with ID ${id} not found`);
      }
    } catch (error) {
      errors.push(`Task ${id}: ${error.message}`);
    }
  }

  return { updatedTasks, errors };
}

async bulkDeleteTasks(ids) {
  const deletedTasks = [];
  const errors = [];

  // Sort IDs in descending order to avoid index shifting issues
  const sortedIds = ids.sort((a, b) => b - a);

  for (const id of sortedIds) {
    const deleted = await this.deleteTask(id);
    if (deleted) {
      deletedTasks.push(deleted);
    } else {
      errors.push(`Task with ID ${id} not found`);
    }
  }

  return { deletedTasks: deletedTasks.reverse(), errors };
}
```

```
// ANALYTICS
async getTaskAnalytics() {
  const now = new Date();
  const oneWeekAgo = new Date(now.getTime() - 7 * 24 * 60 * 60 * 1000);
  const oneMonthAgo = new Date(now.getTime() - 30 * 24 * 60 * 60 * 1000);

  return {
    total: this.tasks.length,
    byStatus: {
      pending: this.tasks.filter((t) => t.status === "pending").length,
      "in-progress": this.tasks.filter((t) => t.status === "in-progress")
        .length,
      completed: this.tasks.filter((t) => t.status === "completed").length,
    },
    byPriority: {
      low: this.tasks.filter((t) => t.priority === "low").length,
      medium: this.tasks.filter((t) => t.priority === "medium").length,
      high: this.tasks.filter((t) => t.priority === "high").length,
      urgent: this.tasks.filter((t) => t.priority === "urgent").length,
    },
    overdue: this.tasks.filter(
      (t) => t.dueDate && t.dueDate < now && t.status !== "completed"
    ).length,
    completedThisWeek: this.tasks.filter(
      (t) => t.completedAt && t.completedAt >= oneWeekAgo
    ).length,
    completedThisMonth: this.tasks.filter(
      (t) => t.completedAt && t.completedAt >= oneMonthAgo
    ).length,
    averageCompletionTime: this.calculateAverageCompletionTime(),
    topAssignees: this.getTopAssignees(),
  };
}

calculateAverageCompletionTime() {
  const completedTasks = this.tasks.filter((t) => t.completedAt);
  if (completedTasks.length === 0) return 0;

  const totalTime = completedTasks.reduce((sum, task) => {
    const completionTime = task.completedAt - task.createdAt;
    return sum + completionTime;
  }, 0);

  return Math.round(
    totalTime / completedTasks.length / (1000 * 60 * 60 * 24)
  ); // days
}

getTopAssignees() {
  const assigneeCounts = {};
  this.tasks.forEach((task) => {
    if (task.assignedTo) {
      assigneeCounts[task.assignedTo] =
        (assigneeCounts[task.assignedTo] || 0) + 1;
    }
  });
}
```

```

    }
  });

  return Object.entries(assigneeCounts)
    .sort(([a], [b]) => b - a)
    .slice(0, 5)
    .map([name, count] => ({ name, count }));
}
}

module.exports = TaskService;

```

## CRUD Controllers

```

// controllers/TaskController.js - Request/Response Handling
const TaskService = require("../services/TaskService");

class TaskController {
  constructor() {
    this.taskService = new TaskService();
  }

  // CREATE - POST /api/tasks
  async createTask(req, res) {
    try {
      const task = await this.taskService.createTask(req.body);

      res.status(201).json({
        success: true,
        data: task,
        message: "Task created successfully",
      });
    } catch (error) {
      if (error.message.includes("Validation failed")) {
        return res.status(422).json({
          success: false,
          error: "Validation Error",
          message: error.message,
        });
      }

      res.status(500).json({
        success: false,
        error: "Internal Server Error",
        message: error.message,
      });
    }
  }

  // READ - GET /api/tasks
  async getAllTasks(req, res) {

```



```
    try {
      const options = {
        page: req.query.page,
        limit: req.query.limit,
        sortBy: req.query.sortBy,
        sortOrder: req.query.sortOrder,
        status: req.query.status,
        priority: req.query.priority,
        assignedTo: req.query.assignedTo,
        search: req.query.search,
        tags: req.query.tags,
        overdue: req.query.overdue,
        dueDateFrom: req.query.dueDateFrom,
        dueDateTo: req.query.dueDateTo,
      };

      const result = await this.taskService.getAllTasks(options);

      res.json({
        success: true,
        data: result.tasks,
        pagination: result.pagination,
        stats: result.stats,
        message: "Tasks retrieved successfully",
      });
    } catch (error) {
      res.status(500).json({
        success: false,
        error: "Internal Server Error",
        message: error.message,
      });
    }
  }

// READ - GET /api/tasks/:id
async getTaskById(req, res) {
  try {
    const task = await this.taskService.getTaskById(req.params.id);

    if (!task) {
      return res.status(404).json({
        success: false,
        error: "Not Found",
        message: `Task with ID ${req.params.id} does not exist`,
      });
    }

    res.json({
      success: true,
      data: task,
      message: "Task retrieved successfully",
    });
  } catch (error) {
    res.status(500).json({
```

```
        success: false,
        error: "Internal Server Error",
        message: error.message,
    });
}
}

// UPDATE - PUT /api/tasks/:id
async updateTask(req, res) {
    try {
        const task = await this.taskService.updateTask(req.params.id, req.body);

        if (!task) {
            return res.status(404).json({
                success: false,
                error: "Not Found",
                message: `Task with ID ${req.params.id} does not exist`,
            });
        }

        res.json({
            success: true,
            data: task,
            message: "Task updated successfully",
        });
    } catch (error) {
        if (error.message.includes("Validation failed")) {
            return res.status(422).json({
                success: false,
                error: "Validation Error",
                message: error.message,
            });
        }

        res.status(500).json({
            success: false,
            error: "Internal Server Error",
            message: error.message,
        });
    }
}

// UPDATE - PATCH /api/tasks/:id
async patchTask(req, res) {
    try {
        const task = await this.taskService.patchTask(req.params.id, req.body);

        if (!task) {
            return res.status(404).json({
                success: false,
                error: "Not Found",
                message: `Task with ID ${req.params.id} does not exist`,
            });
        }
    }
}
```

```
    res.json({
      success: true,
      data: task,
      message: "Task updated successfully",
    });
  } catch (error) {
    if (error.message.includes("Validation failed")) {
      return res.status(422).json({
        success: false,
        error: "Validation Error",
        message: error.message,
      });
    }

    res.status(500).json({
      success: false,
      error: "Internal Server Error",
      message: error.message,
    });
  }
}

// DELETE - DELETE /api/tasks/:id
async deleteTask(req, res) {
  try {
    const task = await this.taskService.deleteTask(req.params.id);

    if (!task) {
      return res.status(404).json({
        success: false,
        error: "Not Found",
        message: `Task with ID ${req.params.id} does not exist`,
      });
    }

    res.json({
      success: true,
      data: task,
      message: "Task deleted successfully",
    });
  } catch (error) {
    res.status(500).json({
      success: false,
      error: "Internal Server Error",
      message: error.message,
    });
  }
}

// BULK OPERATIONS

// PATCH /api/tasks/bulk
async bulkUpdateTasks(req, res) {
```

```
try {
  const { ids, updateData } = req.body;

  if (!ids || !Array.isArray(ids) || ids.length === 0) {
    return res.status(400).json({
      success: false,
      error: "Bad Request",
      message: "Array of task IDs is required",
    });
  }

  if (!updateData || Object.keys(updateData).length === 0) {
    return res.status(400).json({
      success: false,
      error: "Bad Request",
      message: "Update data is required",
    });
  }

  const result = await this.taskService.bulkUpdateTasks(ids, updateData);

  res.json({
    success: true,
    data: {
      updated: result.updatedTasks,
      errors: result.errors,
    },
    message: `${result.updatedTasks.length} tasks updated successfully`,
  });
} catch (error) {
  res.status(500).json({
    success: false,
    error: "Internal Server Error",
    message: error.message,
  });
}
}

// DELETE /api/tasks/bulk
async bulkDeleteTasks(req, res) {
  try {
    const { ids } = req.body;

    if (!ids || !Array.isArray(ids) || ids.length === 0) {
      return res.status(400).json({
        success: false,
        error: "Bad Request",
        message: "Array of task IDs is required",
      });
    }

    const result = await this.taskService.bulkDeleteTasks(ids);

    res.json({
```

```
        success: true,
        data: {
            deleted: result.deletedTasks,
            errors: result.errors,
        },
        message: `${result.deletedTasks.length} tasks deleted successfully`,
    });
} catch (error) {
    res.status(500).json({
        success: false,
        error: "Internal Server Error",
        message: error.message,
    });
}
}

// ANALYTICS - GET /api/tasks/analytics
async getTaskAnalytics(req, res) {
    try {
        const analytics = await this.taskService.getTaskAnalytics();

        res.json({
            success: true,
            data: analytics,
            message: "Task analytics retrieved successfully",
        });
    } catch (error) {
        res.status(500).json({
            success: false,
            error: "Internal Server Error",
            message: error.message,
        });
    }
}

// CUSTOM ACTIONS

// PATCH /api/tasks/:id/complete
async completeTask(req, res) {
    try {
        const task = await this.taskService.patchTask(req.params.id, {
            status: "completed",
        });

        if (!task) {
            return res.status(404).json({
                success: false,
                error: "Not Found",
                message: `Task with ID ${req.params.id} does not exist`,
            });
        }

        res.json({
            success: true,
```

```

        data: task,
        message: "Task marked as completed",
    });
} catch (error) {
    res.status(500).json({
        success: false,
        error: "Internal Server Error",
        message: error.message,
    });
}
}

// GET /api/tasks/overdue
async getOverdueTasks(req, res) {
    try {
        const options = {
            ...req.query,
            overdue: "true",
        };

        const result = await this.taskService.getAllTasks(options);

        res.json({
            success: true,
            data: result.tasks,
            pagination: result.pagination,
            count: result.tasks.length,
            message: "Overdue tasks retrieved successfully",
        });
    } catch (error) {
        res.status(500).json({
            success: false,
            error: "Internal Server Error",
            message: error.message,
        });
    }
}
}

module.exports = TaskController;

```

## CRUD Routes with Validation

```

// routes/tasks.js - Task Routes with Validation
const express = require("express");
const { body, query, param, validationResult } = require("express-validator");
const TaskController = require("../controllers/TaskController");
const router = express.Router();

const taskController = new TaskController();

```

```
// Validation middleware
const handleValidationErrors = (req, res, next) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(422).json({
      success: false,
      error: "Validation Error",
      message: "Invalid input data",
      details: errors.array(),
    });
  }
  next();
};

// Validation rules
const createTaskValidation = [
  body("title")
    .trim()
    .isLength({ min: 3, max: 100 })
    .withMessage("Title must be between 3 and 100 characters"),
  body("description")
    .optional()
    .isLength({ max: 500 })
    .withMessage("Description cannot exceed 500 characters"),
  body("status")
    .optional()
    .isIn(["pending", "in-progress", "completed"])
    .withMessage("Status must be one of: pending, in-progress, completed"),
  body("priority")
    .optional()
    .isIn(["low", "medium", "high", "urgent"])
    .withMessage("Priority must be one of: low, medium, high, urgent"),
  body("dueDate")
    .optional()
    .isISO8601()
    .withMessage("Due date must be a valid ISO 8601 date"),
  body("assignedTo")
    .optional()
    .trim()
    .isLength({ min: 1, max: 50 })
    .withMessage("Assigned to must be between 1 and 50 characters"),
  body("tags").optional().isArray().withMessage("Tags must be an array"),
  body("tags.*")
    .optional()
    .trim()
    .isLength({ min: 1, max: 20 })
    .withMessage("Each tag must be between 1 and 20 characters"),
];

const updateTaskValidation = [
  param("id")
    .isInt({ min: 1 })
    .withMessage("Task ID must be a positive integer"),
  ...createTaskValidation,
```

```
];

const patchTaskValidation = [
  param("id")
    .isInt({ min: 1 })
    .withMessage("Task ID must be a positive integer"),
  body("title")
    .optional()
    .trim()
    .isLength({ min: 3, max: 100 })
    .withMessage("Title must be between 3 and 100 characters"),
  body("description")
    .optional()
    .isLength({ max: 500 })
    .withMessage("Description cannot exceed 500 characters"),
  body("status")
    .optional()
    .isIn(["pending", "in-progress", "completed"])
    .withMessage("Status must be one of: pending, in-progress, completed"),
  body("priority")
    .optional()
    .isIn(["low", "medium", "high", "urgent"])
    .withMessage("Priority must be one of: low, medium, high, urgent"),
  body("dueDate")
    .optional()
    .isISO8601()
    .withMessage("Due date must be a valid ISO 8601 date"),
  body("assignedTo")
    .optional()
    .trim()
    .isLength({ min: 1, max: 50 })
    .withMessage("Assigned to must be between 1 and 50 characters"),
  body("tags").optional().isArray().withMessage("Tags must be an array"),
  body("tags.*")
    .optional()
    .trim()
    .isLength({ min: 1, max: 20 })
    .withMessage("Each tag must be between 1 and 20 characters"),
];

const queryValidation = [
  query("page")
    .optional()
    .isInt({ min: 1 })
    .withMessage("Page must be a positive integer"),
  query("limit")
    .optional()
    .isInt({ min: 1, max: 100 })
    .withMessage("Limit must be between 1 and 100"),
  query("sortBy")
    .optional()
    .isIn(["createdAt", "updatedAt", "title", "status", "priority", "dueDate"])
    .withMessage("Invalid sort field"),
  query("sortOrder")
```



```

    .optional()
    .isIn(["asc", "desc"])
    .withMessage("Sort order must be asc or desc"),
  query("status")
    .optional()
    .isIn(["pending", "in-progress", "completed"])
    .withMessage("Invalid status filter"),
  query("priority")
    .optional()
    .isIn(["low", "medium", "high", "urgent"])
    .withMessage("Invalid priority filter"),
  query("overdue")
    .optional()
    .isBoolean()
    .withMessage("Overdue must be true or false"),
  query("dueDateFrom")
    .optional()
    .isISO8601()
    .withMessage("Due date from must be a valid ISO 8601 date"),
  query("dueDateTo")
    .optional()
    .isISO8601()
    .withMessage("Due date to must be a valid ISO 8601 date"),
];

const bulkValidation = [
  body("ids")
    .isArray({ min: 1 })
    .withMessage("IDs array is required and must not be empty"),
  body("ids.*")
    .isInt({ min: 1 })
    .withMessage("Each ID must be a positive integer"),
];

// Request logging middleware
router.use((req, res, next) => {
  console.log(`${new Date().toISOString()} - ${req.method} ${req.originalUrl}`);
  if (req.body && Object.keys(req.body).length > 0) {
    console.log("Request body:", JSON.stringify(req.body, null, 2));
  }
  next();
});

// CRUD Routes

// CREATE - POST /api/tasks
router.post(
  "/",
  createTaskValidation,
  handleValidationErrors,
  taskController.createTask.bind(taskController)
);

// READ - GET /api/tasks

```

```
router.get(
  "/",
  queryValidation,
  handleValidationErrors,
  taskController.getAllTasks.bind(taskController)
);

// READ - GET /api/tasks/analytics (before /:id to avoid conflicts)
router.get("/analytics", taskController.getTaskAnalytics.bind(taskController));

// READ - GET /api/tasks/overdue
router.get(
  "/overdue",
  queryValidation,
  handleValidationErrors,
  taskController.getOverdueTasks.bind(taskController)
);

// READ - GET /api/tasks/:id
router.get(
  "/:id",
  param("id")
    .isInt({ min: 1 })
    .withMessage("Task ID must be a positive integer"),
  handleValidationErrors,
  taskController.getTaskById.bind(taskController)
);

// UPDATE - PUT /api/tasks/:id
router.put(
  "/:id",
  updateTaskValidation,
  handleValidationErrors,
  taskController.updateTask.bind(taskController)
);

// UPDATE - PATCH /api/tasks/:id
router.patch(
  "/:id",
  patchTaskValidation,
  handleValidationErrors,
  taskController.patchTask.bind(taskController)
);

// DELETE - DELETE /api/tasks/:id
router.delete(
  "/:id",
  param("id")
    .isInt({ min: 1 })
    .withMessage("Task ID must be a positive integer"),
  handleValidationErrors,
  taskController.deleteTask.bind(taskController)
);
```

```
// BULK OPERATIONS

// PATCH /api/tasks/bulk
router.patch(
  "/bulk",
  bulkValidation,
  body("updateData").isObject().withMessage("Update data object is required"),
  handleValidationErrors,
  taskController.bulkUpdateTasks.bind(taskController)
);

// DELETE /api/tasks/bulk
router.delete(
  "/bulk",
  bulkValidation,
  handleValidationErrors,
  taskController.bulkDeleteTasks.bind(taskController)
);

// CUSTOM ACTIONS

// PATCH /api/tasks/:id/complete
router.patch(
  "/*:id/complete",
  param("id")
    .isInt({ min: 1 })
    .withMessage("Task ID must be a positive integer"),
  handleValidationErrors,
  taskController.completeTask.bind(taskController)
);

// Error handling middleware for this router
router.use((error, req, res, next) => {
  console.error("Task router error:", error);
  res.status(500).json({
    success: false,
    error: "Internal Server Error",
    message: "An error occurred while processing the task request",
  });
});

module.exports = router;
```

## Real-World Use Case

### Complete Task Management API

```
// app.js - Complete CRUD Application
const express = require("express");
const cors = require("cors");
const helmet = require("helmet");
```

```
const morgan = require("morgan");
const rateLimit = require("express-rate-limit");
const compression = require("compression");

const app = express();

// Security and performance middleware
app.use(helmet());
app.use(compression());
app.use(
  cors({
    origin: process.env.ALLOWED_ORIGINS?.split(",") || [
      "http://localhost:3000",
    ],
    credentials: true,
  })
);

// Rate limiting
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests per windowMs
  message: {
    success: false,
    error: "Too Many Requests",
    message: "Rate limit exceeded. Please try again later.",
  },
  standardHeaders: true,
  legacyHeaders: false,
});
app.use("/api/", limiter);

// Logging
app.use(morgan("combined"));

// Body parsing
app.use(express.json({ limit: "10mb" }));
app.use(express.urlencoded({ extended: true, limit: "10mb" }));

// API versioning
const v1Router = express.Router();

// Import routes
const taskRoutes = require("./routes/tasks");
const userRoutes = require("./routes/users");
const projectRoutes = require("./routes/projects");

// Mount routes
v1Router.use("/tasks", taskRoutes);
v1Router.use("/users", userRoutes);
v1Router.use("/projects", projectRoutes);

// API info endpoint
v1Router.get("/", (req, res) => {
```

```
res.json({
  success: true,
  message: "Task Management API v1",
  version: "1.0.0",
  endpoints: {
    tasks: "/api/v1/tasks",
    users: "/api/v1/users",
    projects: "/api/v1/projects",
  },
  features: {
    crud: "Full CRUD operations",
    filtering: "Advanced filtering and search",
    pagination: "Cursor and offset pagination",
    sorting: "Multi-field sorting",
    bulk: "Bulk operations support",
    analytics: "Built-in analytics",
    validation: "Comprehensive input validation",
  },
  documentation: "/api/v1/docs",
  timestamp: new Date().toISOString(),
});
});

// Health check with detailed status
v1Router.get("/health", async (req, res) => {
  try {
    // Check database connectivity (if using database)
    // const dbStatus = await checkDatabaseConnection();

    const health = {
      status: "healthy",
      timestamp: new Date().toISOString(),
      uptime: process.uptime(),
      memory: {
        used: Math.round(process.memoryUsage().heapUsed / 1024 / 1024),
        total: Math.round(process.memoryUsage().heapTotal / 1024 / 1024),
        external: Math.round(process.memoryUsage().external / 1024 / 1024),
      },
      version: "1.0.0",
      environment: process.env.NODE_ENV || "development",
      // database: dbStatus
    };

    res.json({
      success: true,
      data: health,
    });
  } catch (error) {
    res.status(503).json({
      success: false,
      error: "Service Unavailable",
      message: "Health check failed",
      details: error.message,
    });
  }
});
```

```
    }
  });

  // Mount API version
  app.use("/api/v1", v1Router);

  // Root endpoint
  app.get("/", (req, res) => {
    res.json({
      success: true,
      message: "Task Management API Server",
      version: "1.0.0",
      api: {
        v1: "/api/v1",
        documentation: "/api/v1/docs",
        health: "/api/v1/health",
      },
      status: "running",
      timestamp: new Date().toISOString(),
    });
  });

  // 404 handler
  app.use("*", (req, res) => {
    res.status(404).json({
      success: false,
      error: "Not Found",
      message: `Route ${req.method} ${req.originalUrl} not found`,
      availableEndpoints: {
        api: "/api/v1",
        tasks: "/api/v1/tasks",
        users: "/api/v1/users",
        projects: "/api/v1/projects",
        health: "/api/v1/health",
      },
    });
  });

  // Global error handler
  app.use((err, req, res, next) => {
    console.error("Global error handler:", err);

    // Handle specific error types
    if (err.type === "entity.parse.failed") {
      return res.status(400).json({
        success: false,
        error: "Bad Request",
        message: "Invalid JSON in request body",
      });
    }

    if (err.type === "entity.too.large") {
      return res.status(413).json({
        success: false,
```

```

        error: "Payload Too Large",
        message: "Request body exceeds size limit",
    });
}

if (err.code === "ENOENT") {
    return res.status(500).json({
        success: false,
        error: "Internal Server Error",
        message: "File system error occurred",
    });
}

res.status(err.status || 500).json({
    success: false,
    error: err.status === 500 ? "Internal Server Error" : err.name || "Error",
    message:
        process.env.NODE_ENV === "production"
            ? "An error occurred while processing your request"
            : err.message,
    ...(process.env.NODE_ENV !== "production" && { stack: err.stack }),
});
});

// Graceful shutdown
process.on("SIGTERM", () => {
    console.log("SIGTERM received, shutting down gracefully");
    server.close(() => {
        console.log("Process terminated");
    });
});

process.on("SIGINT", () => {
    console.log("SIGINT received, shutting down gracefully");
    server.close(() => {
        console.log("Process terminated");
    });
});

const PORT = process.env.PORT || 3000;
const server = app.listen(PORT, () => {
    console.log(`🚀 Task Management API Server running on port ${PORT}`);
    console.log(`📖 API Documentation: http://localhost:${PORT}/api/v1/docs`);
    console.log(`🏥 Health Check: http://localhost:${PORT}/api/v1/health`);
    console.log(`📋 Tasks API: http://localhost:${PORT}/api/v1/tasks`);
});

module.exports = app;

```

## Best Practices

### 1. Implement Proper Validation

```
// Use express-validator for comprehensive validation
const { body, query, param } = require("express-validator");

// Input sanitization
body("title").trim().escape(),
body("email").normalizeEmail(),
// Custom validation
body("dueDate").custom((value) => {
  if (new Date(value) < new Date()) {
    throw new Error("Due date cannot be in the past");
  }
  return true;
});
```

## 2. Use Consistent Error Handling

```
// Async error wrapper
const asyncHandler = (fn) => (req, res, next) => {
  Promise.resolve(fn(req, res, next)).catch(next);
};

// Centralized error handling
class AppError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
    this.isOperational = true;
  }
}
```

## 3. Implement Pagination and Filtering

```
// Cursor-based pagination for large datasets
const paginateResults = (data, cursor, limit) => {
  const startIndex = cursor
    ? data.findIndex((item) => item.id === cursor) + 1
    : 0;
  const endIndex = startIndex + limit;
  const results = data.slice(startIndex, endIndex);

  return {
    data: results,
    hasNext: endIndex < data.length,
    nextCursor: results.length > 0 ? results[results.length - 1].id : null,
  };
};
```



## 4. Add Request/Response Logging

```
// Custom logging middleware
const requestLogger = (req, res, next) => {
  const start = Date.now();

  res.on("finish", () => {
    const duration = Date.now() - start;
    console.log(
      `${req.method} ${req.originalUrl} - ${res.statusCode} - ${duration}ms`
    );
  });

  next();
};
```

## 5. Implement Data Persistence

```
// File-based persistence with atomic writes
const fs = require("fs").promises;
const path = require("path");

class FileStorage {
  async saveData(filename, data) {
    const tempFile = `${filename}.tmp`;
    await fs.writeFile(tempFile, JSON.stringify(data, null, 2));
    await fs.rename(tempFile, filename);
  }

  async loadData(filename) {
    try {
      const data = await fs.readFile(filename, "utf8");
      return JSON.parse(data);
    } catch (error) {
      if (error.code === "ENOENT") {
        return null; // File doesn't exist
      }
      throw error;
    }
  }
}
```

## Summary

CRUD operations form the foundation of data-driven applications in Express.js:

### Core CRUD Operations:

- **Create:** POST requests with validation and error handling

- **Read:** GET requests with filtering, sorting, and pagination
- **Update:** PUT (full) and PATCH (partial) with validation
- **Delete:** DELETE requests with proper cleanup

#### Advanced Features:

- **Bulk operations:** Handle multiple records efficiently
- **Search and filtering:** Query parameters for data retrieval
- **Analytics:** Built-in reporting and statistics
- **Validation:** Comprehensive input validation and sanitization
- **Error handling:** Consistent error responses and logging

#### Best Practices:

- Use proper HTTP methods and status codes
- Implement comprehensive validation
- Add pagination for large datasets
- Use consistent response formats
- Handle errors gracefully
- Add request/response logging
- Implement data persistence strategies

#### Benefits:

- **Scalable:** Handle growing data requirements
- **Maintainable:** Clear separation of concerns
- **Robust:** Comprehensive error handling and validation
- **Flexible:** Support for various query patterns
- **Performance:** Efficient data operations and caching

Mastering CRUD operations is essential for building robust web applications. Next, we'll explore middleware in Express.js, which provides powerful ways to process requests and responses throughout the application lifecycle.

## Middleware in Express.js

---

### Overview

Middleware functions are the backbone of Express.js applications. They are functions that have access to the request object (**req**), response object (**res**), and the next middleware function in the application's request-response cycle. Middleware can execute code, modify request and response objects, end the request-response cycle, or call the next middleware in the stack. Understanding middleware is crucial for building scalable and maintainable Express applications.

### Key Concepts

What is Middleware?

**Middleware Function:** A function that sits between the raw HTTP request and the final route handler, processing the request and response objects.

**Middleware Stack:** The sequence of middleware functions that execute in order during request processing.

**Next Function:** A function that passes control to the next middleware function. If not called, the request will be left hanging.

## Types of Middleware

1. **Application-level middleware:** Bound to the app object
2. **Router-level middleware:** Bound to router instances
3. **Error-handling middleware:** Special middleware for handling errors
4. **Built-in middleware:** Provided by Express (e.g., `express.static`)
5. **Third-party middleware:** External packages (e.g., `cors`, `helmet`)

## Middleware Execution Order

Middleware executes in the order it's defined:

1. Application-level middleware
2. Router-level middleware
3. Route handlers
4. Error-handling middleware

## Example Code

### Basic Middleware Concepts

```
// middleware/logger.js - Custom Logging Middleware
const fs = require("fs").promises;
const path = require("path");

class Logger {
  constructor(options = {}) {
    this.logFile =
      options.logFile || path.join(__dirname, "../logs/access.log");
    this.format = options.format || "combined";
    this.enableConsole = options.enableConsole !== false;
    this.enableFile = options.enableFile !== false;
    this.ensureLogDirectory();
  }

  async ensureLogDirectory() {
    try {
      const logDir = path.dirname(this.logFile);
      await fs.mkdir(logDir, { recursive: true });
    } catch (error) {
      console.error("Failed to create log directory:", error);
    }
  }
}
```

```
formatLog(req, res, responseTime) {
  const timestamp = new Date().toISOString();
  const method = req.method;
  const url = req.originalUrl || req.url;
  const status = res.statusCode;
  const userAgent = req.get("User-Agent") || "-";
  const ip = req.ip || req.connection.remoteAddress;
  const contentLength = res.get("Content-Length") || "-";

  switch (this.format) {
    case "simple":
      return `${timestamp} ${method} ${url} ${status} ${responseTime}ms`;
    case "combined":
      return `${ip} - - [${timestamp}] "${method} ${url} HTTP/1.1" ${status}
${contentLength} "-" "${userAgent}" ${responseTime}ms`;
    case "json":
      return JSON.stringify({
        timestamp,
        method,
        url,
        status,
        responseTime,
        ip,
        userAgent,
        contentLength,
      });
    default:
      return `${timestamp} ${method} ${url} ${status} ${responseTime}ms`;
  }
}

middleware() {
  return (req, res, next) => {
    const startTime = Date.now();

    // Capture the original end function
    const originalEnd = res.end;

    // Override the end function to log when response is sent
    res.end = (...args) => {
      const responseTime = Date.now() - startTime;
      const logEntry = this.formatLog(req, res, responseTime);

      // Log to console
      if (this.enableConsole) {
        console.log(logEntry);
      }

      // Log to file
      if (this.enableFile) {
        this.writeToFile(logEntry);
      }
    };
  };
}
```

```
        // Call the original end function
        originalEnd.apply(res, args);
    };

    next();
};
}

async writeFile(logEntry) {
    try {
        await fs.appendFile(this.logFile, logEntry + "\n");
    } catch (error) {
        console.error("Failed to write to log file:", error);
    }
}
}

module.exports = Logger;
```

```
// middleware/auth.js - Authentication Middleware
const jwt = require("jsonwebtoken");

class AuthMiddleware {
    constructor(options = {}) {
        this.secret = options.secret || process.env.JWT_SECRET || "default-secret";
        this.algorithms = options.algorithms || ["HS256"];
        this.expiresIn = options.expiresIn || "24h";
        this.issuer = options.issuer || "task-api";
    }

    // Generate JWT token
    generateToken(payload) {
        return jwt.sign({ ...payload, iss: this.issuer }, this.secret, {
            expiresIn: this.expiresIn,
            algorithm: this.algorithms[0],
        });
    }

    // Verify JWT token
    verifyToken(token) {
        try {
            return jwt.verify(token, this.secret, {
                algorithms: this.algorithms,
                issuer: this.issuer,
            });
        } catch (error) {
            throw new Error(`Token verification failed: ${error.message}`);
        }
    }

    // Extract token from request
```

```
extractToken(req) {
  // Check Authorization header
  const authHeader = req.headers.authorization;
  if (authHeader && authHeader.startsWith("Bearer ")) {
    return authHeader.substring(7);
  }

  // Check query parameter
  if (req.query.token) {
    return req.query.token;
  }

  // Check cookies
  if (req.cookies && req.cookies.token) {
    return req.cookies.token;
  }

  return null;
}

// Authentication middleware
authenticate() {
  return (req, res, next) => {
    try {
      const token = this.extractToken(req);

      if (!token) {
        return res.status(401).json({
          success: false,
          error: "Unauthorized",
          message: "Access token is required",
        });
      }

      const decoded = this.verifyToken(token);
      req.user = decoded;
      next();
    } catch (error) {
      return res.status(401).json({
        success: false,
        error: "Unauthorized",
        message: error.message,
      });
    }
  };
}

// Authorization middleware (role-based)
authorize(roles = []) {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({
        success: false,
        error: "Unauthorized",
      });
    }
  };
}
```

```

        message: "Authentication required",
    });
}

if (roles.length > 0 && !roles.includes(req.user.role)) {
    return res.status(403).json({
        success: false,
        error: "Forbidden",
        message: "Insufficient permissions",
    });
}

next();
};
}

// Optional authentication (doesn't fail if no token)
optionalAuth() {
    return (req, res, next) => {
        try {
            const token = this.extractToken(req);

            if (token) {
                const decoded = this.verifyToken(token);
                req.user = decoded;
            }
        } catch (error) {
            // Ignore token errors for optional auth
            console.warn("Optional auth token error:", error.message);
        }

        next();
    };
}

module.exports = AuthMiddleware;

```

## Validation Middleware

```

// middleware/validation.js - Request Validation Middleware
const { validationResult } = require("express-validator");

class ValidationMiddleware {
    // Handle validation errors
    static handleValidationErrors(req, res, next) {
        const errors = validationResult(req);

        if (!errors.isEmpty()) {
            const formattedErrors = errors.array().map((error) => ({
                field: error.path || error.param,
            }));
        }
    }
}

```

```
        message: error.msg,
        value: error.value,
        location: error.location,
    }));

    return res.status(422).json({
        success: false,
        error: "Validation Error",
        message: "Invalid input data",
        details: formattedErrors,
    });
}

next();
}

// Sanitize request data
static sanitizeRequest(req, res, next) {
    // Remove undefined and null values from body
    if (req.body && typeof req.body === "object") {
        Object.keys(req.body).forEach((key) => {
            if (req.body[key] === undefined || req.body[key] === null) {
                delete req.body[key];
            }

            // Trim string values
            if (typeof req.body[key] === "string") {
                req.body[key] = req.body[key].trim();
            }
        });
    }

    next();
}

// Content type validation
static validateContentType(allowedTypes = ["application/json"]) {
    return (req, res, next) => {
        if (req.method === "GET" || req.method === "DELETE") {
            return next();
        }

        const contentType = req.get("Content-Type");

        if (!contentType) {
            return res.status(400).json({
                success: false,
                error: "Bad Request",
                message: "Content-Type header is required",
            });
        }

        const isValidType = allowedTypes.some((type) =>
            contentType.toLowerCase().includes(type.toLowerCase())
        );
    };
}
```



```
    );

    if (!isValidType) {
      return res.status(415).json({
        success: false,
        error: "Unsupported Media Type",
        message: `Content-Type must be one of: ${allowedTypes.join(", ")}`
      });
    }

    next();
  };
}

// Request size validation
static validateRequestSize(maxSize = "10mb") {
  return (req, res, next) => {
    const contentLength = req.get("Content-Length");

    if (contentLength) {
      const sizeInBytes = parseInt(contentLength);
      const maxSizeInBytes = this.parseSize(maxSize);

      if (sizeInBytes > maxSizeInBytes) {
        return res.status(413).json({
          success: false,
          error: "Payload Too Large",
          message: `Request size exceeds maximum allowed size of ${maxSize}`
        });
      }
    }

    next();
  };
}

static parseSize(size) {
  const units = {
    b: 1,
    kb: 1024,
    mb: 1024 * 1024,
    gb: 1024 * 1024 * 1024,
  };

  const match = size.toLowerCase().match(/^(\d+)([a-z]+)$/);
  if (!match) return parseInt(size);

  const [, number, unit] = match;
  return parseInt(number) * (units[unit] || 1);
}

module.exports = ValidationMiddleware;
```

## Error Handling Middleware

```
// middleware/errorHandler.js - Comprehensive Error Handling
class ErrorHandler {
  constructor(options = {}) {
    this.includeStack =
      options.includeStack || process.env.NODE_ENV !== "production";
    this.logErrors = options.logErrors !== false;
    this.logger = options.logger || console;
  }

  // Main error handling middleware
  handle() {
    return (err, req, res, next) => {
      // Log error
      if (this.logErrors) {
        this.logError(err, req);
      }

      // Handle different error types
      if (err.name === "ValidationError") {
        return this.handleValidationError(err, res);
      }

      if (err.name === "CastError") {
        return this.handleCastError(err, res);
      }

      if (err.code === 11000) {
        return this.handleDuplicateError(err, res);
      }

      if (err.name === "JsonWebTokenError") {
        return this.handleJWTError(err, res);
      }

      if (err.name === "TokenExpiredError") {
        return this.handleTokenExpiredError(err, res);
      }

      if (err.type === "entity.parse.failed") {
        return this.handleJSONParseError(err, res);
      }

      if (err.type === "entity.too.large") {
        return this.handlePayloadTooLargeError(err, res);
      }

      // Handle operational errors
      if (err.isOperational) {
        return this.handleOperationalError(err, res);
      }
    };
  }
}
```

```
    }

    // Handle programming errors
    this.handleProgrammingError(err, res);
  };
}

logError(err, req) {
  const errorInfo = {
    timestamp: new Date().toISOString(),
    method: req.method,
    url: req.originalUrl,
    ip: req.ip,
    userAgent: req.get("User-Agent"),
    error: {
      name: err.name,
      message: err.message,
      stack: err.stack,
    },
    user: req.user ? { id: req.user.id, role: req.user.role } : null,
  };

  this.logger.error("Application Error:", JSON.stringify(errorInfo, null, 2));
}

handleValidationError(err, res) {
  const errors = Object.values(err.errors).map((error) => ({
    field: error.path,
    message: error.message,
    value: error.value,
  }));

  return res.status(422).json({
    success: false,
    error: "Validation Error",
    message: "Invalid input data",
    details: errors,
  });
}

handleCastError(err, res) {
  return res.status(400).json({
    success: false,
    error: "Bad Request",
    message: `Invalid ${err.path}: ${err.value}`,
  });
}

handleDuplicateError(err, res) {
  const field = Object.keys(err.keyValue)[0];
  const value = err.keyValue[field];

  return res.status(409).json({
    success: false,
```

```
        error: "Conflict",
        message: `${field} '${value}' already exists`,
    });
}

handleJWTError(err, res) {
    return res.status(401).json({
        success: false,
        error: "Unauthorized",
        message: "Invalid authentication token",
    });
}

handleTokenExpiredError(err, res) {
    return res.status(401).json({
        success: false,
        error: "Unauthorized",
        message: "Authentication token has expired",
    });
}

handleJSONParseError(err, res) {
    return res.status(400).json({
        success: false,
        error: "Bad Request",
        message: "Invalid JSON in request body",
    });
}

handlePayloadTooLargeError(err, res) {
    return res.status(413).json({
        success: false,
        error: "Payload Too Large",
        message: "Request body exceeds size limit",
    });
}

handleOperationalError(err, res) {
    return res.status(err.statusCode || 500).json({
        success: false,
        error: err.name || "Operational Error",
        message: err.message,
        ...(this.includeStack && { stack: err.stack }),
    });
}

handleProgrammingError(err, res) {
    return res.status(500).json({
        success: false,
        error: "Internal Server Error",
        message: this.includeStack ? err.message : "Something went wrong",
        ...(this.includeStack && { stack: err.stack }),
    });
}
```

```

// 404 handler
notFound() {
  return (req, res, next) => {
    const error = new Error(
      `Route ${req.method} ${req.originalUrl} not found`
    );
    error.statusCode = 404;
    error.isOperational = true;
    next(error);
  };
}

module.exports = ErrorHandler;

```

## Security Middleware

```

// middleware/security.js - Security Middleware Collection
const rateLimit = require("express-rate-limit");
const slowDown = require("express-slow-down");
const helmet = require("helmet");

class SecurityMiddleware {
  // Rate limiting
  static createRateLimit(options = {}) {
    return rateLimit({
      windowMs: options.windowMs || 15 * 60 * 1000, // 15 minutes
      max: options.max || 100, // limit each IP to 100 requests per windowMs
      message: {
        success: false,
        error: "Too Many Requests",
        message: "Rate limit exceeded. Please try again later.",
        retryAfter: Math.ceil(options.windowMs / 1000) || 900,
      },
      standardHeaders: true,
      legacyHeaders: false,
      handler: (req, res) => {
        res.status(429).json({
          success: false,
          error: "Too Many Requests",
          message: "Rate limit exceeded. Please try again later.",
          retryAfter: Math.ceil(options.windowMs / 1000) || 900,
        });
      },
      skip: options.skip || (() => false),
    });
  }

  // Speed limiting (slow down responses)
  static createSpeedLimit(options = {}) {

```

```
    return slowDown({
      windowMs: options.windowMs || 15 * 60 * 1000, // 15 minutes
      delayAfter: options.delayAfter || 50, // allow 50 requests per windowMs
    });
  }

  // CORS configuration
  static configureCORS(options = {}) {
    return (req, res, next) => {
      const allowedOrigins = options.origins || [
        "http://localhost:3000",
        "http://localhost:3001",
        "https://yourdomain.com",
      ];

      const origin = req.headers.origin;

      if (allowedOrigins.includes(origin) || !origin) {
        res.setHeader("Access-Control-Allow-Origin", origin || "*");
      }

      res.setHeader(
        "Access-Control-Allow-Methods",
        "GET, POST, PUT, PATCH, DELETE, OPTIONS"
      );
      res.setHeader(
        "Access-Control-Allow-Headers",
        "Content-Type, Authorization, X-Requested-With"
      );
      res.setHeader("Access-Control-Allow-Credentials", "true");
      res.setHeader("Access-Control-Max-Age", "86400"); // 24 hours

      if (req.method === "OPTIONS") {
        return res.status(200).end();
      }

      next();
    };
  }

  // Security headers
  static securityHeaders() {
    return helmet({
      contentSecurityPolicy: {
        directives: {
          defaultSrc: ["'self'"],
          styleSrc: ["'self'", "unsafe-inline"],
          scriptSrc: ["'self'"],
        },
      },
    });
  }
}
```

```

        imgSrc: ["'self'", "data:", "https:"],
        connectSrc: ["'self'"],
        fontSrc: ["'self'"],
        objectSrc: ["'none'"],
        mediaSrc: ["'self'"],
        frameSrc: ["'none'"],
      },
    },
    crossOriginEmbedderPolicy: false,
    hsts: {
      maxAge: 31536000,
      includeSubDomains: true,
      preload: true,
    },
  });
}

// IP whitelist/blacklist
static ipFilter(options = {}) {
  const whitelist = options.whitelist || [];
  const blacklist = options.blacklist || [];

  return (req, res, next) => {
    const clientIP = req.ip || req.connection.remoteAddress;

    // Check blacklist first
    if (blacklist.length > 0 && blacklist.includes(clientIP)) {
      return res.status(403).json({
        success: false,
        error: "Forbidden",
        message: "Access denied from this IP address",
      });
    }

    // Check whitelist if defined
    if (whitelist.length > 0 && !whitelist.includes(clientIP)) {
      return res.status(403).json({
        success: false,
        error: "Forbidden",
        message: "Access denied from this IP address",
      });
    }

    next();
  };
}

// Request sanitization
static sanitizeRequest() {
  return (req, res, next) => {
    // Remove potentially dangerous characters
    const sanitize = (obj) => {
      if (typeof obj === "string") {
        return obj

```

```

        .replace(/<script[^>]*>.*?</script>/gi, "")
        .replace(/<[^>]*>/g, "")
        .replace(/javascript:/gi, "")
        .replace(/on\w+=/gi, "");
    }

    if (typeof obj === "object" && obj !== null) {
        for (const key in obj) {
            obj[key] = sanitize(obj[key]);
        }
    }

    return obj;
};

if (req.body) {
    req.body = sanitize(req.body);
}

if (req.query) {
    req.query = sanitize(req.query);
}

if (req.params) {
    req.params = sanitize(req.params);
}

next();
};
}
}

module.exports = SecurityMiddleware;

```

## Complete Middleware Application

```

// app.js - Application with Comprehensive Middleware
const express = require("express");
const cookieParser = require("cookie-parser");
const compression = require("compression");

// Custom middleware
const Logger = require("./middleware/logger");
const AuthMiddleware = require("./middleware/auth");
const ValidationMiddleware = require("./middleware/validation");
const ErrorHandler = require("./middleware/errorHandler");
const SecurityMiddleware = require("./middleware/security");

const app = express();

// Trust proxy (for accurate IP addresses behind reverse proxy)

```



```
app.set("trust proxy", 1);

// Initialize middleware instances
const logger = new Logger({
  format: "combined",
  enableConsole: true,
  enableFile: true,
});

const auth = new AuthMiddleware({
  secret: process.env.JWT_SECRET,
  expiresIn: "24h",
});

const errorHandler = new ErrorHandler({
  includeStack: process.env.NODE_ENV !== "production",
  logErrors: true,
});

// Security middleware (applied first)
app.use(SecurityMiddleware.securityHeaders());
app.use(
  SecurityMiddleware.configureCORS({
    origins: process.env.ALLOWED_ORIGINS?.split(",") || [
      "http://localhost:3000",
    ],
  })
);

// Rate limiting
app.use(
  "/api/",
  SecurityMiddleware.createRateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 100, // requests per window
  })
);

// Speed limiting for auth endpoints
app.use(
  "/api/auth/",
  SecurityMiddleware.createSpeedLimit({
    windowMs: 15 * 60 * 1000,
    delayAfter: 5,
    delayMs: 1000,
  })
);

// Compression
app.use(compression());

// Body parsing
app.use(express.json({ limit: "10mb" }));
app.use(express.urlencoded({ extended: true, limit: "10mb" }));
```

```
app.use(cookieParser());

// Request sanitization
app.use(SecurityMiddleware.sanitizeRequest());

// Content type validation for POST/PUT/PATCH
app.use(ValidationMiddleware.validateContentType(["application/json"]));

// Request size validation
app.use(ValidationMiddleware.validateRequestSize("10mb"));

// Request sanitization
app.use(ValidationMiddleware.sanitizeRequest);

// Logging
app.use(logger.middleware());

// Custom request context middleware
app.use((req, res, next) => {
  req.requestId = require("crypto").randomUUID();
  req.startTime = Date.now();

  // Add request ID to response headers
  res.setHeader("X-Request-ID", req.requestId);

  next();
});

// API versioning
const v1Router = express.Router();

// Authentication routes (no auth required)
v1Router.post("/auth/login", (req, res) => {
  // Mock login logic
  const { username, password } = req.body;

  if (username === "admin" && password === "password") {
    const token = auth.generateToken({
      id: 1,
      username: "admin",
      role: "admin",
    });

    res.json({
      success: true,
      data: {
        token,
        user: { id: 1, username: "admin", role: "admin" },
      },
      message: "Login successful",
    });
  } else {
    res.status(401).json({
      success: false,
```

```
        error: "Unauthorized",
        message: "Invalid credentials",
    });
    }
});

// Protected routes
const protectedRouter = express.Router();

// Apply authentication to all protected routes
protectedRouter.use(auth.authenticate());

// Import and mount route modules
const taskRoutes = require("./routes/tasks");
const userRoutes = require("./routes/users");

protectedRouter.use("/tasks", taskRoutes);
protectedRouter.use("/users", auth.authorize(["admin"]), userRoutes);

// Mount routers
v1Router.use("/", protectedRouter);

// API info endpoint
v1Router.get("/", (req, res) => {
    res.json({
        success: true,
        message: "Task Management API v1",
        version: "1.0.0",
        requestId: req.requestId,
        timestamp: new Date().toISOString(),
        user: req.user || null,
    });
});

// Mount API version
app.use("/api/v1", v1Router);

// Root endpoint
app.get("/", (req, res) => {
    res.json({
        success: true,
        message: "Task Management API Server",
        version: "1.0.0",
        requestId: req.requestId,
        endpoints: {
            api: "/api/v1",
            auth: "/api/v1/auth/login",
            tasks: "/api/v1/tasks",
            users: "/api/v1/users",
        },
    });
});

// Health check endpoint
```

```

app.get("/health", (req, res) => {
  const healthData = {
    status: "healthy",
    timestamp: new Date().toISOString(),
    uptime: process.uptime(),
    memory: {
      used: Math.round(process.memoryUsage().heapUsed / 1024 / 1024),
      total: Math.round(process.memoryUsage().heapTotal / 1024 / 1024),
    },
    requestId: req.requestId,
  };

  res.json({
    success: true,
    data: healthData,
  });
});

// 404 handler
app.use(errorHandler.notFound());

// Global error handler (must be last)
app.use(errorHandler.handle());

// Graceful shutdown
process.on("SIGTERM", () => {
  console.log("SIGTERM received, shutting down gracefully");
  server.close(() => {
    console.log("Process terminated");
  });
});

const PORT = process.env.PORT || 3000;
const server = app.listen(PORT, () => {
  console.log(`🚀 Server running on port ${PORT}`);
  console.log(`📖 API: http://localhost:${PORT}/api/v1`);
  console.log(`🏥 Health: http://localhost:${PORT}/health`);
});

module.exports = app;

```

## Real-World Use Case

### E-commerce API with Advanced Middleware

```

// middleware/analytics.js - Analytics and Monitoring Middleware
class AnalyticsMiddleware {
  constructor() {
    this.metrics = {
      requests: 0,
      errors: 0,

```

```
    responseTimeSum: 0,
    endpoints: {},
    statusCodes: {},
    userAgents: {},
    ips: {},
  };

  // Reset metrics every hour
  setInterval(() => {
    this.resetMetrics();
  }, 60 * 60 * 1000);
}

middleware() {
  return (req, res, next) => {
    const startTime = Date.now();

    // Track request
    this.metrics.requests++;

    // Track endpoint
    const endpoint = `${req.method} ${req.route?.path || req.path}`;
    this.metrics.endpoints[endpoint] =
      (this.metrics.endpoints[endpoint] || 0) + 1;

    // Track IP
    const ip = req.ip;
    this.metrics.ips[ip] = (this.metrics.ips[ip] || 0) + 1;

    // Track User Agent
    const userAgent = req.get("User-Agent") || "Unknown";
    this.metrics.userAgents[userAgent] =
      (this.metrics.userAgents[userAgent] || 0) + 1;

    // Override res.end to capture response metrics
    const originalEnd = res.end;
    res.end = (...args) => {
      const responseTime = Date.now() - startTime;

      // Track response time
      this.metrics.responseTimeSum += responseTime;

      // Track status codes
      const statusCode = res.statusCode;
      this.metrics.statusCodes[statusCode] =
        (this.metrics.statusCodes[statusCode] || 0) + 1;

      // Track errors
      if (statusCode >= 400) {
        this.metrics.errors++;
      }

      // Add performance headers
      res.setHeader("X-Response-Time", `${responseTime}ms`);
    };
  };
}
```

```

        originalEnd.apply(res, args);
    };

    next();
};
}

getMetrics() {
    const avgResponseTime =
        this.metrics.requests > 0
        ? Math.round(this.metrics.responseTimeSum / this.metrics.requests)
        : 0;

    return {
        ...this.metrics,
        averageResponseTime: avgResponseTime,
        errorRate:
            this.metrics.requests > 0
            ? Math.round((this.metrics.errors / this.metrics.requests) * 100)
            : 0,
        timestamp: new Date().toISOString(),
    };
}

resetMetrics() {
    this.metrics = {
        requests: 0,
        errors: 0,
        responseTimeSum: 0,
        endpoints: {},
        statusCodes: {},
        userAgents: {},
        ips: {},
    };
}
}

module.exports = AnalyticsMiddleware;

```

```

// middleware/cache.js - Response Caching Middleware
const NodeCache = require("node-cache");

class CacheMiddleware {
    constructor(options = {}) {
        this.cache = new NodeCache({
            stdTTL: options.ttl || 300, // 5 minutes default
            checkperiod: options.checkperiod || 60, // check for expired keys every 60
seconds
            useClones: false,
        });
    }
}

```

```
    this.defaultTTL = options.ttl || 300;
  }

  // Cache GET requests
  cacheGet(ttl = this.defaultTTL) {
    return (req, res, next) => {
      // Only cache GET requests
      if (req.method !== "GET") {
        return next();
      }

      // Create cache key from URL and query parameters
      const cacheKey = this.generateCacheKey(req);

      // Check if response is cached
      const cachedResponse = this.cache.get(cacheKey);

      if (cachedResponse) {
        res.setHeader("X-Cache", "HIT");
        res.setHeader("X-Cache-Key", cacheKey);
        return res.json(cachedResponse);
      }

      // Override res.json to cache the response
      const originalJson = res.json;
      res.json = (data) => {
        // Only cache successful responses
        if (res.statusCode >= 200 && res.statusCode < 300) {
          this.cache.set(cacheKey, data, ttl);
          res.setHeader("X-Cache", "MISS");
          res.setHeader("X-Cache-TTL", ttl.toString());
        } else {
          res.setHeader("X-Cache", "SKIP");
        }

        res.setHeader("X-Cache-Key", cacheKey);
        originalJson.call(res, data);
      };

      next();
    };
  }

  // Invalidate cache for specific patterns
  invalidate(pattern) {
    const keys = this.cache.keys();
    const keysToDelete = keys.filter((key) => key.includes(pattern));

    keysToDelete.forEach((key) => {
      this.cache.del(key);
    });

    return keysToDelete.length;
  }
}
```

```
}

// Clear all cache
clear() {
  this.cache.flushAll();
}

// Get cache statistics
getStats() {
  return {
    keys: this.cache.keys().length,
    hits: this.cache.getStats().hits,
    misses: this.cache.getStats().misses,
    ksize: this.cache.getStats().ksize,
    vsize: this.cache.getStats().vsize,
  };
}

generateCacheKey(req) {
  const url = req.originalUrl || req.url;
  const user = req.user ? req.user.id : "anonymous";
  return `${user}:${url}`;
}

// Middleware to add cache control headers
cacheControl(maxAge = 300) {
  return (req, res, next) => {
    if (req.method === "GET") {
      res.setHeader("Cache-Control", `public, max-age=${maxAge}`);
      res.setHeader("ETag", this.generateETag(req));
    } else {
      res.setHeader("Cache-Control", "no-cache, no-store, must-revalidate");
    }

    next();
  };
}

generateETag(req) {
  const crypto = require("crypto");
  const content = req.originalUrl + (req.user ? req.user.id : "");
  return crypto.createHash("md5").update(content).digest("hex");
}

module.exports = CacheMiddleware;
```

## Best Practices

### 1. Middleware Order Matters



```
// Correct order:
app.use(helmet()); // Security first
app.use(cors()); // CORS before other middleware
app.use(rateLimit); // Rate limiting early
app.use(express.json()); // Body parsing
app.use(authentication); // Auth before routes
app.use("/api", routes); // Routes
app.use(errorHandler); // Error handling last
```

## 2. Use Async Middleware Properly

```
// Async middleware wrapper
const asyncMiddleware = (fn) => (req, res, next) => {
  Promise.resolve(fn(req, res, next)).catch(next);
};

// Usage
app.use(
  "/api/users",
  asyncMiddleware(async (req, res, next) => {
    req.user = await getUserFromDatabase(req.headers.authorization);
    next();
  })
);
```

## 3. Create Reusable Middleware

```
// Factory function for creating middleware
const createAuthMiddleware = (options = {}) => {
  return (req, res, next) => {
    // Middleware logic with options
    next();
  };
};

// Usage
app.use("/admin", createAuthMiddleware({ role: "admin" }));
app.use("/api", createAuthMiddleware({ role: "user" }));
```

## 4. Handle Errors Properly

```
// Always call next() with error to trigger error handling
app.use((req, res, next) => {
  try {
    // Middleware logic
    next();
  } catch (err) {
    next(err);
  }
});
```

```
    } catch (error) {  
      next(error); // Pass error to error handler  
    }  
  });  
});
```

## 5. Use Conditional Middleware

```
// Apply middleware conditionally  
const conditionalMiddleware = (condition, middleware) => {  
  return (req, res, next) => {  
    if (condition(req)) {  
      return middleware(req, res, next);  
    }  
    next();  
  };  
};  
  
// Usage  
app.use(conditionalMiddleware((req) => req.path.startsWith("/api"), rateLimit));
```

## Summary

Middleware is the foundation of Express.js applications, providing a powerful way to process requests and responses:

### Key Concepts:

- **Middleware Stack:** Functions execute in order
- **Next Function:** Controls flow to next middleware
- **Error Handling:** Special middleware for error processing
- **Types:** Application, router, built-in, third-party, and error-handling

### Common Middleware Types:

- **Security:** Authentication, authorization, rate limiting
- **Logging:** Request/response logging and analytics
- **Validation:** Input validation and sanitization
- **Caching:** Response caching and cache control
- **Error Handling:** Comprehensive error processing

### Best Practices:

- Order middleware correctly
- Handle async operations properly
- Create reusable middleware factories
- Always handle errors
- Use conditional middleware when needed
- Keep middleware focused and single-purpose

**Benefits:**

- **Modularity:** Separate concerns into focused functions
- **Reusability:** Share middleware across routes and applications
- **Flexibility:** Apply middleware conditionally
- **Maintainability:** Clear separation of cross-cutting concerns
- **Testability:** Easy to test individual middleware functions

Mastering middleware is essential for building robust, secure, and maintainable Express.js applications. Next, we'll explore database integration with MongoDB and Mongoose, building upon the middleware foundation to create data-driven applications.

## Database Integration: MongoDB and Mongoose

---

### Overview

MongoDB is a popular NoSQL document database that stores data in flexible, JSON-like documents. Mongoose is an Object Document Mapping (ODM) library for MongoDB and Node.js that provides a schema-based solution to model application data. This chapter covers integrating MongoDB with Express.js applications using Mongoose, including schema design, CRUD operations, data validation, and advanced querying.

### Key Concepts

#### MongoDB Fundamentals

**Document Database:** MongoDB stores data in BSON (Binary JSON) documents within collections.

**Collections:** Groups of documents, similar to tables in relational databases.

**Documents:** Individual records stored as key-value pairs, similar to rows in relational databases.

**Schema-less:** Documents in a collection don't need to have the same structure.

#### Mongoose Benefits

**Schema Definition:** Define structure and validation rules for documents.

**Type Casting:** Automatic type conversion and validation.

**Middleware:** Pre and post hooks for document operations.

**Query Building:** Chainable query API with powerful features.

**Population:** Reference other documents and populate them automatically.

**Validation:** Built-in and custom validation rules.

### Example Code

#### Database Connection Setup

```
// config/database.js - Database Configuration
const mongoose = require("mongoose");

class DatabaseConnection {
  constructor() {
    this.connection = null;
    this.isConnected = false;
  }

  async connect(options = {}) {
    try {
      const mongoUri =
        options.uri ||
        process.env.MONGODB_URI ||
        "mongodb://localhost:27017/taskmanager";

      const connectionOptions = {
        useNewUrlParser: true,
        useUnifiedTopology: true,
        maxPoolSize: options.maxPoolSize || 10,
        serverSelectionTimeoutMS: options.serverSelectionTimeoutMS || 5000,
        socketTimeoutMS: options.socketTimeoutMS || 45000,
        bufferCommands: false,
        bufferMaxEntries: 0,
        ...options.mongooseOptions,
      };

      this.connection = await mongoose.connect(mongoUri, connectionOptions);
      this.isConnected = true;

      console.log(
        `✅ MongoDB connected:
        ${this.connection.connection.host}:${this.connection.connection.port}`
      );
      console.log(`📊 Database: ${this.connection.connection.name}`);

      // Connection event listeners
      mongoose.connection.on("error", this.handleError.bind(this));
      mongoose.connection.on(
        "disconnected",
        this.handleDisconnected.bind(this)
      );
      mongoose.connection.on("reconnected", this.handleReconnected.bind(this));

      return this.connection;
    } catch (error) {
      console.error("❌ MongoDB connection error:", error.message);
      throw error;
    }
  }

  async disconnect() {

```

```
    try {
      if (this.isConnected) {
        await mongoose.disconnect();
        this.isConnected = false;
        console.log("🔌 MongoDB disconnected");
      }
    } catch (error) {
      console.error("❌ MongoDB disconnection error:", error.message);
      throw error;
    }
  }

  handleError(error) {
    console.error("❌ MongoDB error:", error.message);
    this.isConnected = false;
  }

  handleDisconnected() {
    console.warn("⚠️ MongoDB disconnected");
    this.isConnected = false;
  }

  handleReconnected() {
    console.log("🔄 MongoDB reconnected");
    this.isConnected = true;
  }

  getConnectionStatus() {
    return {
      isConnected: this.isConnected,
      readyState: mongoose.connection.readyState,
      host: mongoose.connection.host,
      port: mongoose.connection.port,
      name: mongoose.connection.name,
    };
  }

  // Health check
  async healthCheck() {
    try {
      const adminDb = mongoose.connection.db.admin();
      const result = await adminDb.ping();
      return {
        status: "healthy",
        ping: result,
        timestamp: new Date().toISOString(),
      };
    } catch (error) {
      return {
        status: "unhealthy",
        error: error.message,
        timestamp: new Date().toISOString(),
      };
    }
  }
}
```

```
}  
}  
  
module.exports = new DatabaseConnection();
```

## Schema Definitions

```
// models/User.js - User Schema  
const mongoose = require("mongoose");  
const bcrypt = require("bcryptjs");  
const validator = require("validator");  
  
const userSchema = new mongoose.Schema(  
  {  
    username: {  
      type: String,  
      required: [true, "Username is required"],  
      unique: true,  
      trim: true,  
      minlength: [3, "Username must be at least 3 characters"],  
      maxlength: [30, "Username cannot exceed 30 characters"],  
      match: [  
        /^[a-zA-Z0-9_]+$/,  
        "Username can only contain letters, numbers, and underscores",  
      ],  
    },  
    email: {  
      type: String,  
      required: [true, "Email is required"],  
      unique: true,  
      lowercase: true,  
      trim: true,  
      validate: {  
        validator: validator.isEmail,  
        message: "Please provide a valid email address",  
      },  
    },  
    password: {  
      type: String,  
      required: [true, "Password is required"],  
      minlength: [8, "Password must be at least 8 characters"],  
      select: false, // Don't include password in queries by default  
    },  
    firstName: {  
      type: String,  
      required: [true, "First name is required"],  
      trim: true,  
      maxlength: [50, "First name cannot exceed 50 characters"],  
    },  
    lastName: {  
      type: String,
```

```
    required: [true, "Last name is required"],
    trim: true,
    maxlength: [50, "Last name cannot exceed 50 characters"],
  },
  avatar: {
    type: String,
    validate: {
      validator: function (v) {
        return !v || validator.isURL(v);
      },
      message: "Avatar must be a valid URL",
    },
  },
},
role: {
  type: String,
  enum: {
    values: ["user", "admin", "moderator"],
    message: "Role must be either user, admin, or moderator",
  },
  default: "user",
},
isActive: {
  type: Boolean,
  default: true,
},
lastLogin: {
  type: Date,
},
preferences: {
  theme: {
    type: String,
    enum: ["light", "dark", "auto"],
    default: "auto",
  },
  notifications: {
    email: { type: Boolean, default: true },
    push: { type: Boolean, default: true },
    sms: { type: Boolean, default: false },
  },
  timezone: {
    type: String,
    default: "UTC",
  },
},
},
metadata: {
  createdBy: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "User",
  },
},
tags: [
  {
    type: String,
    trim: true,
  },
],
```

```
    ],
    notes: String,
  },
},
{
  timestamps: true, // Adds createdAt and updatedAt
  toJSON: {
    virtuals: true,
    transform: function (doc, ret) {
      delete ret.password;
      delete ret.__v;
      return ret;
    },
  },
  toObject: { virtuals: true },
}
);

// Indexes
userSchema.index({ email: 1 });
userSchema.index({ username: 1 });
userSchema.index({ role: 1, isActive: 1 });
userSchema.index({ createdAt: -1 });
userSchema.index({ "metadata.tags": 1 });

// Virtual properties
userSchema.virtual("fullName").get(function () {
  return `${this.firstName} ${this.lastName}`;
});

userSchema.virtual("initials").get(function () {
  return `${this.firstName.charAt(0)}${this.lastName.charAt(0)}`.toUpperCase();
});

userSchema.virtual("tasksCount", {
  ref: "Task",
  localField: "_id",
  foreignField: "assignedTo",
  count: true,
});

// Pre-save middleware
userSchema.pre("save", async function (next) {
  // Only hash password if it's modified
  if (!this.isModified("password")) return next();

  try {
    // Hash password
    const salt = await bcrypt.genSalt(12);
    this.password = await bcrypt.hash(this.password, salt);
    next();
  } catch (error) {
    next(error);
  }
});
```



```
});

// Pre-save middleware for username uniqueness
userSchema.pre("save", async function (next) {
  if (!this.isModified("username")) return next();

  try {
    const existingUser = await this.constructor.findOne({
      username: this.username,
      _id: { $ne: this._id },
    });

    if (existingUser) {
      const error = new Error("Username already exists");
      error.name = "ValidationError";
      return next(error);
    }

    next();
  } catch (error) {
    next(error);
  }
});

// Instance methods
userSchema.methods.comparePassword = async function (candidatePassword) {
  try {
    return await bcrypt.compare(candidatePassword, this.password);
  } catch (error) {
    throw new Error("Password comparison failed");
  }
};

userSchema.methods.updateLastLogin = async function () {
  this.lastLogin = new Date();
  return await this.save({ validateBeforeSave: false });
};

userSchema.methods.toPublicJSON = function () {
  const userObject = this.toObject();
  delete userObject.password;
  delete userObject.__v;
  return userObject;
};

// Static methods
userSchema.statics.findByEmail = function (email) {
  return this.findOne({ email: email.toLowerCase() });
};

userSchema.statics.findActiveUsers = function () {
  return this.find({ isActive: true });
};
```

```

userSchema.statics.getUserStats = async function () {
  const stats = await this.aggregate([
    {
      $group: {
        _id: null,
        totalUsers: { $sum: 1 },
        activeUsers: {
          $sum: { $cond: [{ $eq: ["$isActive", true] }, 1, 0] },
        },
        adminUsers: {
          $sum: { $cond: [{ $eq: ["$role", "admin"] }, 1, 0] },
        },
        averageTasksPerUser: { $avg: "$tasksCount" },
      },
    },
  ]);

  return (
    stats[0] || {
      totalUsers: 0,
      activeUsers: 0,
      adminUsers: 0,
      averageTasksPerUser: 0,
    }
  );
};

module.exports = mongoose.model("User", userSchema);

```

```

// models/Task.js - Task Schema
const mongoose = require("mongoose");

const taskSchema = new mongoose.Schema(
  {
    title: {
      type: String,
      required: [true, "Task title is required"],
      trim: true,
      maxlength: [200, "Title cannot exceed 200 characters"],
    },
    description: {
      type: String,
      trim: true,
      maxlength: [2000, "Description cannot exceed 2000 characters"],
    },
    status: {
      type: String,
      enum: {
        values: ["pending", "in-progress", "completed", "cancelled"],
        message: "Status must be pending, in-progress, completed, or cancelled",
      },
    },
  },

```

```
    default: "pending",
  },
  priority: {
    type: String,
    enum: {
      values: ["low", "medium", "high", "urgent"],
      message: "Priority must be low, medium, high, or urgent",
    },
    default: "medium",
  },
  category: {
    type: String,
    required: [true, "Category is required"],
    trim: true,
    maxlength: [50, "Category cannot exceed 50 characters"],
  },
  tags: [
    {
      type: String,
      trim: true,
      maxlength: [30, "Tag cannot exceed 30 characters"],
    },
  ],
  assignedTo: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "User",
    required: [true, "Task must be assigned to a user"],
  },
  createdBy: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "User",
    required: [true, "Creator is required"],
  },
  dueDate: {
    type: Date,
    validate: {
      validator: function (v) {
        return !v || v > new Date();
      },
      message: "Due date must be in the future",
    },
  },
  completedAt: {
    type: Date,
  },
  estimatedHours: {
    type: Number,
    min: [0, "Estimated hours cannot be negative"],
    max: [1000, "Estimated hours cannot exceed 1000"],
  },
  actualHours: {
    type: Number,
    min: [0, "Actual hours cannot be negative"],
    max: [1000, "Actual hours cannot exceed 1000"],
  },
}
```

```
    },
    attachments: [
      {
        filename: {
          type: String,
          required: true,
        },
        originalName: {
          type: String,
          required: true,
        },
        mimetype: {
          type: String,
          required: true,
        },
        size: {
          type: Number,
          required: true,
        },
        url: {
          type: String,
          required: true,
        },
        uploadedAt: {
          type: Date,
          default: Date.now,
        },
      },
    ],
    comments: [
      {
        author: {
          type: mongoose.Schema.Types.ObjectId,
          ref: "User",
          required: true,
        },
        content: {
          type: String,
          required: [true, "Comment content is required"],
          trim: true,
          maxlength: [1000, "Comment cannot exceed 1000 characters"],
        },
        createdAt: {
          type: Date,
          default: Date.now,
        },
        editedAt: Date,
        isEdited: {
          type: Boolean,
          default: false,
        },
      },
    ],
    subtasks: [
```

```

    {
      title: {
        type: String,
        required: [true, "Subtask title is required"],
        trim: true,
        maxlength: [200, "Subtask title cannot exceed 200 characters"],
      },
      completed: {
        type: Boolean,
        default: false,
      },
      completedAt: Date,
      order: {
        type: Number,
        default: 0,
      },
    },
  ],
  isArchived: {
    type: Boolean,
    default: false,
  },
},
{
  timestamps: true,
  toJSON: { virtuals: true },
  toObject: { virtuals: true },
}
);

// Indexes
taskSchema.index({ assignedTo: 1, status: 1 });
taskSchema.index({ createdBy: 1 });
taskSchema.index({ status: 1, priority: 1 });
taskSchema.index({ category: 1 });
taskSchema.index({ tags: 1 });
taskSchema.index({ dueDate: 1 });
taskSchema.index({ createdAt: -1 });
taskSchema.index({ title: "text", description: "text" }); // Text search

// Virtual properties
taskSchema.virtual("isOverdue").get(function () {
  return (
    this.dueDate && this.dueDate < new Date() && this.status !== "completed"
  );
});

taskSchema.virtual("daysUntilDue").get(function () {
  if (!this.dueDate) return null;
  const diffTime = this.dueDate - new Date();
  return Math.ceil(diffTime / (1000 * 60 * 60 * 24));
});

taskSchema.virtual("completionPercentage").get(function () {

```

```
    if (this.subtasks.length === 0) {
      return this.status === "completed" ? 100 : 0;
    }

    const completedSubtasks = this.subtasks.filter(
      (subtask) => subtask.completed
    ).length;
    return Math.round((completedSubtasks / this.subtasks.length) * 100);
  });

taskSchema.virtual("timeSpent").get(function () {
  return this.actualHours || 0;
});

// Pre-save middleware
taskSchema.pre("save", function (next) {
  // Set completedAt when status changes to completed
  if (
    this.isModified("status") &&
    this.status === "completed" &&
    !this.completedAt
  ) {
    this.completedAt = new Date();
  }

  // Clear completedAt when status changes from completed
  if (
    this.isModified("status") &&
    this.status !== "completed" &&
    this.completedAt
  ) {
    this.completedAt = undefined;
  }

  next();
});

// Pre-save middleware for subtasks
taskSchema.pre("save", function (next) {
  if (this.isModified("subtasks")) {
    this.subtasks.forEach((subtask, index) => {
      if (subtask.completed && !subtask.completedAt) {
        subtask.completedAt = new Date();
      } else if (!subtask.completed && subtask.completedAt) {
        subtask.completedAt = undefined;
      }

      // Set order if not provided
      if (subtask.order === undefined) {
        subtask.order = index;
      }
    });
  }
});
}
```

```
    next();
  });

  // Instance methods
  taskSchema.methods.addComment = function (authorId, content) {
    this.comments.push({
      author: authorId,
      content: content,
    });
    return this.save();
  };

  taskSchema.methods.addSubtask = function (title) {
    this.subtasks.push({
      title: title,
      order: this.subtasks.length,
    });
    return this.save();
  };

  taskSchema.methods.toggleSubtask = function (subtaskId) {
    const subtask = this.subtasks.id(subtaskId);
    if (subtask) {
      subtask.completed = !subtask.completed;
      subtask.completedAt = subtask.completed ? new Date() : undefined;
      return this.save();
    }
    throw new Error("Subtask not found");
  };

  taskSchema.methods.archive = function () {
    this.isArchived = true;
    return this.save();
  };

  taskSchema.methods.unarchive = function () {
    this.isArchived = false;
    return this.save();
  };

  // Static methods
  taskSchema.statics.findByUser = function (userId) {
    return this.find({ assignedTo: userId, isArchived: false });
  };

  taskSchema.statics.findOverdue = function () {
    return this.find({
      dueDate: { $lt: new Date() },
      status: { $ne: "completed" },
      isArchived: false,
    });
  };

  taskSchema.statics.getTaskStats = async function (userId = null) {
```

```

const matchStage = userId
  ? { assignedTo: mongoose.Types.ObjectId(userId) }
  : {};

const stats = await this.aggregate([
  { $match: { ...matchStage, isArchived: false } },
  {
    $group: {
      _id: null,
      totalTasks: { $sum: 1 },
      pendingTasks: {
        $sum: { $cond: [{ $eq: ["$status", "pending"] }, 1, 0] },
      },
      inProgressTasks: {
        $sum: { $cond: [{ $eq: ["$status", "in-progress"] }, 1, 0] },
      },
      completedTasks: {
        $sum: { $cond: [{ $eq: ["$status", "completed"] }, 1, 0] },
      },
      overdueTasks: {
        $sum: {
          $cond: [
            {
              $and: [
                { $lt: ["$dueDate", new Date()] },
                { $ne: ["$status", "completed"] },
              ],
            },
            1,
            0,
          ],
        },
      },
      averageCompletionTime: {
        $avg: {
          $cond: [
            { $eq: ["$status", "completed"] },
            { $subtract: ["$completedAt", "$createdAt"] },
            null,
          ],
        },
      },
    },
  ],
  {});

return (
  stats[0] || {
    totalTasks: 0,
    pendingTasks: 0,
    inProgressTasks: 0,
    completedTasks: 0,
    overdueTasks: 0,
    averageCompletionTime: 0,
  }
);

```



```
    }  
  );  
};  
  
module.exports = mongoose.model("Task", taskSchema);
```

## Service Layer with Advanced Queries

```
// services/TaskService.js - Task Business Logic  
const Task = require("../models/Task");  
const User = require("../models/User");  
const mongoose = require("mongoose");  
  
class TaskService {  
  // Create a new task  
  async createTask(taskData, createdBy) {  
    try {  
      // Validate assignee exists  
      const assignee = await User.findById(taskData.assignedTo);  
      if (!assignee) {  
        throw new Error("Assigned user not found");  
      }  
  
      const task = new Task({  
        ...taskData,  
        createdBy,  
      });  
  
      await task.save();  
  
      // Populate references  
      await task.populate([  
        { path: "assignedTo", select: "username email firstName lastName" },  
        { path: "createdBy", select: "username email firstName lastName" },  
      ]);  
  
      return task;  
    } catch (error) {  
      throw new Error(`Failed to create task: ${error.message}`);  
    }  
  }  
  
  // Get tasks with advanced filtering and pagination  
  async getTasks(options = {}) {  
    try {  
      const {  
        page = 1,  
        limit = 10,  
        sortBy = "createdAt",  
        sortOrder = "desc",  
        status,
```

```
    priority,
    category,
    assignedTo,
    createdBy,
    tags,
    search,
    dueDateFrom,
    dueDateTo,
    isOverdue,
    includeArchived = false,
  } = options;

  // Build query
  const query = {};

  if (!includeArchived) {
    query.isArchived = false;
  }

  if (status) {
    if (Array.isArray(status)) {
      query.status = { $in: status };
    } else {
      query.status = status;
    }
  }

  if (priority) {
    if (Array.isArray(priority)) {
      query.priority = { $in: priority };
    } else {
      query.priority = priority;
    }
  }

  if (category) {
    query.category = new RegExp(category, "i");
  }

  if (assignedTo) {
    query.assignedTo = assignedTo;
  }

  if (createdBy) {
    query.createdBy = createdBy;
  }

  if (tags && tags.length > 0) {
    query.tags = { $in: tags };
  }

  if (search) {
    query.$text = { $search: search };
  }
}
```

```
// Date range filtering
if (dueDateFrom || dueDateTo) {
  query.dueDate = {};
  if (dueDateFrom) {
    query.dueDate.$gte = new Date(dueDateFrom);
  }
  if (dueDateTo) {
    query.dueDate.$lte = new Date(dueDateTo);
  }
}

// Overdue filtering
if (isOverdue === true) {
  query.dueDate = { $lt: new Date() };
  query.status = { $ne: "completed" };
}

// Build sort object
const sort = {};
sort[sortBy] = sortOrder === "desc" ? -1 : 1;

// Calculate pagination
const skip = (page - 1) * limit;

// Execute query with population
const [tasks, totalCount] = await Promise.all([
  Task.find(query)
    .populate("assignedTo", "username email firstName lastName avatar")
    .populate("createdBy", "username email firstName lastName")
    .populate("comments.author", "username firstName lastName avatar")
    .sort(sort)
    .skip(skip)
    .limit(parseInt(limit))
    .lean(),
  Task.countDocuments(query),
]);

// Calculate pagination info
const totalPages = Math.ceil(totalCount / limit);
const hasNextPage = page < totalPages;
const hasPrevPage = page > 1;

return {
  tasks,
  pagination: {
    currentPage: parseInt(page),
    totalPages,
    totalCount,
    hasNextPage,
    hasPrevPage,
    limit: parseInt(limit),
  },
  filters: {
```

```
        status,
        priority,
        category,
        assignedTo,
        createdBy,
        tags,
        search,
        dueDateFrom,
        dueDateTo,
        isOverdue,
      },
    ];
  } catch (error) {
    throw new Error(`Failed to get tasks: ${error.message}`);
  }
}

// Get task by ID with full population
async getTaskById(taskId, userId = null) {
  try {
    if (!mongoose.Types.ObjectId.isValid(taskId)) {
      throw new Error("Invalid task ID");
    }

    const query = { _id: taskId };

    // If userId provided, ensure user has access to the task
    if (userId) {
      query.$or = [{ assignedTo: userId }, { createdBy: userId }];
    }

    const task = await Task.findOne(query)
      .populate("assignedTo", "username email firstName lastName avatar role")
      .populate("createdBy", "username email firstName lastName avatar")
      .populate("comments.author", "username firstName lastName avatar")
      .lean();

    if (!task) {
      throw new Error("Task not found or access denied");
    }

    return task;
  } catch (error) {
    throw new Error(`Failed to get task: ${error.message}`);
  }
}

// Update task
async updateTask(taskId, updateData, userId) {
  try {
    if (!mongoose.Types.ObjectId.isValid(taskId)) {
      throw new Error("Invalid task ID");
    }
  }
}
```

```
// Find task and check permissions
const task = await Task.findOne({
  _id: taskId,
  $or: [{ assignedTo: userId }, { createdBy: userId }],
});

if (!task) {
  throw new Error("Task not found or access denied");
}

// Validate assignee if being updated
if (
  updateData.assignedTo &&
  updateData.assignedTo !== task.assignedTo.toString()
) {
  const assignee = await User.findById(updateData.assignedTo);
  if (!assignee) {
    throw new Error("Assigned user not found");
  }
}

// Update task
Object.assign(task, updateData);
await task.save();

// Populate and return updated task
await task.populate([
  {
    path: "assignedTo",
    select: "username email firstName lastName avatar",
  },
  { path: "createdBy", select: "username email firstName lastName" },
  {
    path: "comments.author",
    select: "username firstName lastName avatar",
  },
]);

return task;
} catch (error) {
  throw new Error(`Failed to update task: ${error.message}`);
}
}

// Delete task
async deleteTask(taskId, userId) {
  try {
    if (!mongoose.Types.ObjectId.isValid(taskId)) {
      throw new Error("Invalid task ID");
    }

    const task = await Task.findOneAndDelete({
      _id: taskId,
      $or: [{ assignedTo: userId }, { createdBy: userId }],
    });
  }
}
```

```
    });

    if (!task) {
      throw new Error("Task not found or access denied");
    }

    return { message: "Task deleted successfully" };
  } catch (error) {
    throw new Error(`Failed to delete task: ${error.message}`);
  }
}

// Bulk operations
async bulkUpdateTasks(taskIds, updateData, userId) {
  try {
    const validIds = taskIds.filter((id) =>
      mongoose.Types.ObjectId.isValid(id)
    );

    if (validIds.length === 0) {
      throw new Error("No valid task IDs provided");
    }

    const result = await Task.updateMany(
      {
        _id: { $in: validIds },
        $or: [{ assignedTo: userId }, { createdBy: userId }],
      },
      updateData
    );

    return {
      matchedCount: result.matchedCount,
      modifiedCount: result.modifiedCount,
      message: `Updated ${result.modifiedCount} tasks`,
    };
  } catch (error) {
    throw new Error(`Failed to bulk update tasks: ${error.message}`);
  }
}

// Get task analytics
async getTaskAnalytics(userId = null, dateRange = null) {
  try {
    const matchStage = {};

    if (userId) {
      matchStage.$or = [
        { assignedTo: mongoose.Types.ObjectId(userId) },
        { createdBy: mongoose.Types.ObjectId(userId) },
      ];
    }

    if (dateRange) {

```

```

    matchStage.createdAt = {
      $gte: new Date(dateRange.from),
      $lte: new Date(dateRange.to),
    };
  }

  const analytics = await Task.aggregate([
    { $match: { ...matchStage, isArchived: false } },
    {
      $facet: {
        statusDistribution: [
          {
            $group: {
              _id: "$status",
              count: { $sum: 1 },
            },
          ],
        ],
        priorityDistribution: [
          {
            $group: {
              _id: "$priority",
              count: { $sum: 1 },
            },
          ],
        ],
        categoryDistribution: [
          {
            $group: {
              _id: "$category",
              count: { $sum: 1 },
            },
          ],
          { $sort: { count: -1 } },
          { $limit: 10 },
        ],
        completionTrend: [
          {
            $match: { status: "completed" },
          ],
          {
            $group: {
              _id: {
                year: { $year: "$completedAt" },
                month: { $month: "$completedAt" },
                day: { $dayOfMonth: "$completedAt" },
              },
              count: { $sum: 1 },
            },
          ],
          { $sort: { "_id.year": 1, "_id.month": 1, "_id.day": 1 } },
        ],
        averageCompletionTime: [
          {

```

```

        $match: { status: "completed", completedAt: { $exists: true } },
      },
    {
      $group: {
        _id: null,
        avgTime: {
          $avg: {
            $subtract: ["$completedAt", "$createdAt"],
          },
        },
      },
    },
  ],
},
]);

return analytics[0];
} catch (error) {
  throw new Error(`Failed to get task analytics: ${error.message}`);
}
}
}

module.exports = new TaskService();

```

## Database Middleware and Hooks

```

// middleware/database.js - Database Middleware
const mongoose = require("mongoose");
const database = require("../config/database");

class DatabaseMiddleware {
  // Connection middleware
  static async ensureConnection(req, res, next) {
    try {
      if (!database.isConnected) {
        await database.connect();
      }
      next();
    } catch (error) {
      res.status(503).json({
        success: false,
        error: "Service Unavailable",
        message: "Database connection failed",
      });
    }
  }
}

// Transaction middleware
static withTransaction() {

```



```
return async (req, res, next) => {
  const session = await mongoose.startSession();

  try {
    await session.withTransaction(async () => {
      req.dbSession = session;
      await new Promise((resolve, reject) => {
        const originalEnd = res.end;
        res.end = (...args) => {
          if (res.statusCode >= 400) {
            reject(new Error("Transaction failed due to error response"));
          } else {
            resolve();
          }
        };
        originalEnd.apply(res, args);
      });

      next();
    });
  } catch (error) {
    console.error("Transaction failed:", error);
    if (!res.headersSent) {
      res.status(500).json({
        success: false,
        error: "Internal Server Error",
        message: "Transaction failed",
      });
    }
  } finally {
    await session.endSession();
  }
};

// Database health check middleware
static healthCheck() {
  return async (req, res, next) => {
    try {
      const health = await database.healthCheck();
      req.dbHealth = health;
      next();
    } catch (error) {
      req.dbHealth = {
        status: "unhealthy",
        error: error.message,
        timestamp: new Date().toISOString(),
      };
      next();
    }
  };
}

// Query performance monitoring
```

```

static queryMonitor() {
  return (req, res, next) => {
    const originalQuery = mongoose.Query.prototype.exec;
    const queries = [];

    mongoose.Query.prototype.exec = function () {
      const startTime = Date.now();
      const query = this.getQuery();
      const collection = this.mongooseCollection.name;

      return originalQuery.call(this).then((result) => {
        const duration = Date.now() - startTime;
        queries.push({
          collection,
          query,
          duration,
          timestamp: new Date().toISOString(),
        });

        return result;
      });
    };

    // Restore original exec after request
    const originalEnd = res.end;
    res.end = (...args) => {
      mongoose.Query.prototype.exec = originalQuery;

      // Add query info to response headers (for debugging)
      if (process.env.NODE_ENV === "development") {
        res.setHeader("X-DB-Queries", queries.length.toString());
        res.setHeader(
          "X-DB-Query-Time",
          queries.reduce((sum, q) => sum + q.duration, 0).toString() + "ms"
        );
      }

      req.dbQueries = queries;
      originalEnd.apply(res, args);
    };

    next();
  };
}

module.exports = DatabaseMiddleware;

```

## Real-World Use Case

### Complete Task Management API with MongoDB

```
// app.js - Complete Application with MongoDB Integration
const express = require("express");
const cors = require("cors");
const helmet = require("helmet");
const compression = require("compression");
const rateLimit = require("express-rate-limit");

// Database
const database = require("../config/database");
const DatabaseMiddleware = require("../middleware/database");

// Routes
const authRoutes = require("../routes/auth");
const taskRoutes = require("../routes/tasks");
const userRoutes = require("../routes/users");
const analyticsRoutes = require("../routes/analytics");

const app = express();

// Trust proxy
app.set("trust proxy", 1);

// Security middleware
app.use(helmet());
app.use(
  cors({
    origin: process.env.ALLOWED_ORIGINS?.split(",") || [
      "http://localhost:3000",
    ],
    credentials: true,
  })
);

// Rate limiting
app.use(
  rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 100, // requests per window
  })
);

// Compression
app.use(compression());

// Body parsing
app.use(express.json({ limit: "10mb" }));
app.use(express.urlencoded({ extended: true, limit: "10mb" }));

// Database middleware
app.use(DatabaseMiddleware.ensureConnection);
app.use(DatabaseMiddleware.queryMonitor());

// Routes
```

```
app.use("/api/v1/auth", authRoutes);
app.use("/api/v1/tasks", taskRoutes);
app.use("/api/v1/users", userRoutes);
app.use("/api/v1/analytics", analyticsRoutes);

// Health check with database status
app.get("/health", DatabaseMiddleware.healthCheck(), (req, res) => {
  const healthData = {
    status: "healthy",
    timestamp: new Date().toISOString(),
    uptime: process.uptime(),
    memory: {
      used: Math.round(process.memoryUsage().heapUsed / 1024 / 1024),
      total: Math.round(process.memoryUsage().heapTotal / 1024 / 1024),
    },
    database: req.dbHealth,
    queries: req.dbQueries?.length || 0,
  };

  const status = req.dbHealth?.status === "healthy" ? 200 : 503;

  res.status(status).json({
    success: status === 200,
    data: healthData,
  });
});

// Root endpoint
app.get("/", (req, res) => {
  res.json({
    success: true,
    message: "Task Management API with MongoDB",
    version: "1.0.0",
    database: database.getConnectionStatus(),
    endpoints: {
      auth: "/api/v1/auth",
      tasks: "/api/v1/tasks",
      users: "/api/v1/users",
      analytics: "/api/v1/analytics",
      health: "/health",
    },
  });
});

// 404 handler
app.use((req, res) => {
  res.status(404).json({
    success: false,
    error: "Not Found",
    message: `Route ${req.method} ${req.originalUrl} not found`,
  });
});

// Global error handler
```

```
app.use((err, req, res, next) => {
  console.error("Application Error:", err);

  // Mongoose validation error
  if (err.name === "ValidationError") {
    const errors = Object.values(err.errors).map((e) => ({
      field: e.path,
      message: e.message,
    }));

    return res.status(422).json({
      success: false,
      error: "Validation Error",
      details: errors,
    });
  }

  // Mongoose cast error
  if (err.name === "CastError") {
    return res.status(400).json({
      success: false,
      error: "Bad Request",
      message: `Invalid ${err.path}: ${err.value}`,
    });
  }

  // MongoDB duplicate key error
  if (err.code === 11000) {
    const field = Object.keys(err.keyValue)[0];
    return res.status(409).json({
      success: false,
      error: "Conflict",
      message: `${field} already exists`,
    });
  }

  // Default error
  res.status(500).json({
    success: false,
    error: "Internal Server Error",
    message:
      process.env.NODE_ENV === "production"
        ? "Something went wrong"
        : err.message,
  });
});

// Graceful shutdown
process.on("SIGTERM", async () => {
  console.log("SIGTERM received, shutting down gracefully");

  try {
    await database.disconnect();
    server.close(() => {
```

```

        console.log("Process terminated");
    });
} catch (error) {
    console.error("Error during shutdown:", error);
    process.exit(1);
}
});

// Start server
const PORT = process.env.PORT || 3000;

async function startServer() {
    try {
        // Connect to database
        await database.connect({
            uri: process.env.MONGODB_URI,
            maxPoolSize: 10,
            serverSelectionTimeoutMS: 5000,
        });

        // Start HTTP server
        const server = app.listen(PORT, () => {
            console.log(`🚀 Server running on port ${PORT}`);
            console.log(`📄 API: http://localhost:${PORT}/api/v1`);
            console.log(`🏥 Health: http://localhost:${PORT}/health`);
            console.log(`🗄 Database: ${database.getConnectionStatus().name}`);
        });

        return server;
    } catch (error) {
        console.error("Failed to start server:", error);
        process.exit(1);
    }
}

if (require.main === module) {
    startServer();
}

module.exports = app;

```

## Best Practices

### 1. Schema Design

```

// Use appropriate data types and validation
const schema = new mongoose.Schema({
    email: {
        type: String,
        required: true,
        unique: true,
    },

```

```
    lowercase: true,
    validate: [validator.isEmail, "Invalid email"],
  },
  createdAt: {
    type: Date,
    default: Date.now,
    index: true, // Add indexes for frequently queried fields
  },
});
```

## 2. Use Indexes Effectively

```
// Compound indexes for complex queries
schema.index({ userId: 1, status: 1, createdAt: -1 });

// Text indexes for search
schema.index({ title: "text", description: "text" });

// Sparse indexes for optional fields
schema.index({ email: 1 }, { sparse: true });
```

## 3. Handle Errors Properly

```
// Use try-catch with async/await
try {
  const user = await User.findById(id);
  if (!user) {
    throw new Error("User not found");
  }
  return user;
} catch (error) {
  throw new Error(`Database operation failed: ${error.message}`);
}
```

## 4. Use Transactions for Related Operations

```
// Use transactions for operations that must succeed together
const session = await mongoose.startSession();
try {
  await session.withTransaction(async () => {
    await User.create([userData], { session });
    await Task.create([taskData], { session });
  });
} finally {
  await session.endSession();
}
```

## 5. Optimize Queries

```
// Use lean() for read-only operations
const users = await User.find().lean();

// Use select() to limit fields
const users = await User.find().select("name email");

// Use populate() efficiently
const tasks = await Task.find()
  .populate("assignedTo", "name email")
  .populate("comments.author", "name");
```

## Summary

MongoDB with Mongoose provides a powerful foundation for Node.js applications:

### Key Features:

- **Schema Definition:** Structure and validation for documents
- **Type Casting:** Automatic data type conversion
- **Middleware:** Pre and post hooks for document operations
- **Query Building:** Chainable, powerful query API
- **Population:** Reference and populate related documents
- **Validation:** Built-in and custom validation rules

### Advanced Capabilities:

- **Aggregation:** Complex data processing and analytics
- **Indexing:** Performance optimization for queries
- **Transactions:** ACID compliance for related operations
- **Change Streams:** Real-time data change notifications
- **GridFS:** Large file storage and retrieval

### Best Practices:

- Design schemas with proper validation and indexes
- Use transactions for related operations
- Optimize queries with `lean()`, `select()`, and `populate()`
- Handle errors appropriately
- Monitor query performance
- Use connection pooling and proper configuration

### Benefits:

- **Flexibility:** Schema-less design with optional structure
- **Scalability:** Horizontal scaling capabilities
- **Performance:** Optimized for read-heavy workloads



- **Developer Experience:** Intuitive API and powerful features
- **Rich Ecosystem:** Extensive tooling and community support

MongoDB and Mongoose integration enables building robust, scalable applications with complex data requirements. Next, we'll explore authentication and authorization patterns, building upon the database foundation to create secure user management systems.

# Authentication and Authorization in Express.js

---

## Overview

Authentication and authorization are critical security components in web applications. Authentication verifies who a user is (identity), while authorization determines what a user can do (permissions). This chapter covers implementing secure authentication using JWT (JSON Web Tokens), session-based authentication, password hashing, role-based access control (RBAC), and advanced security patterns in Express.js applications.

## Key Concepts

### Authentication vs Authorization

**Authentication:** The process of verifying a user's identity ("Who are you?")

- Login with username/password
- Token verification
- Multi-factor authentication
- Social login (OAuth)

**Authorization:** The process of determining what an authenticated user can access ("What can you do?")

- Role-based access control (RBAC)
- Permission-based access control
- Resource-level permissions
- Attribute-based access control (ABAC)

### JWT (JSON Web Tokens)

**Structure:** Header.Payload.Signature

- **Header:** Token type and signing algorithm
- **Payload:** Claims (user data, permissions, expiration)
- **Signature:** Verification of token integrity

**Benefits:** Stateless, scalable, cross-domain support **Considerations:** Token size, security, refresh strategies

### Session-Based Authentication

**Server-side sessions:** Store session data on server **Session cookies:** Client stores session identifier **Benefits:** Server control, easy revocation **Considerations:** Scalability, memory usage

## Example Code

## JWT Authentication Service

```
// services/AuthService.js - Comprehensive Authentication Service
const jwt = require("jsonwebtoken");
const bcrypt = require("bcryptjs");
const crypto = require("crypto");
const User = require("../models/User");
const RefreshToken = require("../models/RefreshToken");

class AuthService {
  constructor() {
    this.jwtSecret = process.env.JWT_SECRET || "your-secret-key";
    this.jwtRefreshSecret =
      process.env.JWT_REFRESH_SECRET || "your-refresh-secret";
    this.accessTokenExpiry = process.env.JWT_EXPIRES_IN || "15m";
    this.refreshTokenExpiry = process.env.JWT_REFRESH_EXPIRES_IN || "7d";
    this.issuer = process.env.JWT_ISSUER || "task-api";
    this.audience = process.env.JWT_AUDIENCE || "task-app";
  }

  // Generate access token
  generateAccessToken(user) {
    const payload = {
      sub: user._id.toString(), // Subject (user ID)
      username: user.username,
      email: user.email,
      role: user.role,
      permissions: this.getUserPermissions(user.role),
      iss: this.issuer, // Issuer
      aud: this.audience, // Audience
      iat: Math.floor(Date.now() / 1000), // Issued at
      type: "access",
    };
    return jwt.sign(payload, this.jwtSecret, {
      expiresIn: this.accessTokenExpiry,
      algorithm: "HS256",
    });
  }

  // Generate refresh token
  async generateRefreshToken(userId) {
    const token = crypto.randomBytes(64).toString("hex");
    const expiresAt = new Date();
    expiresAt.setTime(expiresAt.getTime() + 7 * 24 * 60 * 60 * 1000); // 7 days

    // Store refresh token in database
    const refreshToken = new RefreshToken({
      token,
      userId,
      expiresAt,
      isActive: true,
    });
  }
}
```

```
});

    await refreshToken.save();
    return token;
}

// Verify access token
verifyAccessToken(token) {
    try {
        const decoded = jwt.verify(token, this.jwtSecret, {
            issuer: this.issuer,
            audience: this.audience,
            algorithms: ["HS256"],
        });

        if (decoded.type !== "access") {
            throw new Error("Invalid token type");
        }

        return decoded;
    } catch (error) {
        throw new Error(`Token verification failed: ${error.message}`);
    }
}

// Verify refresh token
async verifyRefreshToken(token) {
    try {
        const refreshToken = await RefreshToken.findOne({
            token,
            isActive: true,
            expiresAt: { $gt: new Date() },
        }).populate("userId");

        if (!refreshToken) {
            throw new Error("Invalid or expired refresh token");
        }

        return refreshToken;
    } catch (error) {
        throw new Error(`Refresh token verification failed: ${error.message}`);
    }
}

// Register new user
async register(userData) {
    try {
        const { username, email, password, firstName, lastName } = userData;

        // Check if user already exists
        const existingUser = await User.findOne({
            $or: [{ email }, { username }],
        });
    }
}
```

```
    if (existingUser) {
      throw new Error("User already exists with this email or username");
    }

    // Create new user
    const user = new User({
      username,
      email,
      password, // Will be hashed by pre-save middleware
      firstName,
      lastName,
      role: "user", // Default role
    });

    await user.save();

    // Generate tokens
    const accessToken = this.generateAccessToken(user);
    const refreshToken = await this.generateRefreshToken(user._id);

    return {
      user: user.toPublicJSON(),
      tokens: {
        accessToken,
        refreshToken,
        expiresIn: this.accessTokenExpiry,
      },
    };
  } catch (error) {
    throw new Error(`Registration failed: ${error.message}`);
  }
}

// Login user
async login(credentials) {
  try {
    const { identifier, password, rememberMe = false } = credentials;

    // Find user by email or username
    const user = await User.findOne({
      $or: [{ email: identifier.toLowerCase() }, { username: identifier }],
      isActive: true,
    }).select("+password"); // Include password field

    if (!user) {
      throw new Error("Invalid credentials");
    }

    // Verify password
    const isValid = await user.comparePassword(password);
    if (!isValid) {
      throw new Error("Invalid credentials");
    }
  }
}
```

```
// Update last login
await user.updateLastLogin();

// Generate tokens
const accessToken = this.generateAccessToken(user);
const refreshToken = await this.generateRefreshToken(user._id);

// Revoke old refresh tokens if not "remember me"
if (!rememberMe) {
  await this.revokeUserRefreshTokens(user._id);
}

return {
  user: user.toPublicJSON(),
  tokens: {
    accessToken,
    refreshToken,
    expiresIn: this.accessTokenExpiry,
  },
};
} catch (error) {
  throw new Error(`Login failed: ${error.message}`);
}
}

// Refresh access token
async refreshAccessToken(refreshToken) {
  try {
    const tokenDoc = await this.verifyRefreshToken(refreshToken);
    const user = tokenDoc.userId;

    if (!user.isActive) {
      throw new Error("User account is deactivated");
    }

    // Generate new access token
    const newAccessToken = this.generateAccessToken(user);

    // Optionally rotate refresh token
    let newRefreshToken = refreshToken;
    if (process.env.ROTATE_REFRESH_TOKENS === "true") {
      // Revoke old refresh token
      await this.revokeRefreshToken(refreshToken);
      // Generate new refresh token
      newRefreshToken = await this.generateRefreshToken(user._id);
    }

    return {
      accessToken: newAccessToken,
      refreshToken: newRefreshToken,
      expiresIn: this.accessTokenExpiry,
    };
  } catch (error) {
    throw new Error(`Token refresh failed: ${error.message}`);
  }
}
```

```
    }  
  }  
  
  // Logout user  
  async logout(refreshToken) {  
    try {  
      if (refreshToken) {  
        await this.revokeRefreshToken(refreshToken);  
      }  
      return { message: "Logged out successfully" };  
    } catch (error) {  
      throw new Error(`Logout failed: ${error.message}`);  
    }  
  }  
  
  // Logout from all devices  
  async logoutAll(userId) {  
    try {  
      await this.revokeUserRefreshTokens(userId);  
      return { message: "Logged out from all devices" };  
    } catch (error) {  
      throw new Error(`Logout all failed: ${error.message}`);  
    }  
  }  
  
  // Revoke refresh token  
  async revokeRefreshToken(token) {  
    await RefreshToken.updateOne(  
      { token },  
      { isActive: false, revokedAt: new Date() }  
    );  
  }  
  
  // Revoke all user refresh tokens  
  async revokeUserRefreshTokens(userId) {  
    await RefreshToken.updateMany(  
      { userId, isActive: true },  
      { isActive: false, revokedAt: new Date() }  
    );  
  }  
  
  // Change password  
  async changePassword(userId, currentPassword, newPassword) {  
    try {  
      const user = await User.findById(userId).select("+password");  
      if (!user) {  
        throw new Error("User not found");  
      }  
  
      // Verify current password  
      const isCurrentPasswordValid = await user.comparePassword(  
        currentPassword  
      );  
      if (!isCurrentPasswordValid) {
```

```
        throw new Error("Current password is incorrect");
    }

    // Update password
    user.password = newPassword;
    await user.save();

    // Revoke all refresh tokens to force re-login
    await this.revokeUserRefreshTokens(userId);

    return { message: "Password changed successfully" };
} catch (error) {
    throw new Error(`Password change failed: ${error.message}`);
}
}

// Reset password
async resetPassword(email) {
    try {
        const user = await User.findOne({ email: email.toLowerCase() });
        if (!user) {
            // Don't reveal if email exists
            return { message: "If the email exists, a reset link has been sent" };
        }

        // Generate reset token
        const resetToken = crypto.randomBytes(32).toString("hex");
        const resetTokenExpiry = new Date(Date.now() + 60 * 60 * 1000); // 1 hour

        // Save reset token to user
        user.passwordResetToken = crypto
            .createHash("sha256")
            .update(resetToken)
            .digest("hex");
        user.passwordResetExpires = resetTokenExpiry;
        await user.save({ validateBeforeSave: false });

        // TODO: Send email with reset link
        // await emailService.sendPasswordResetEmail(user.email, resetToken);

        return {
            message: "Password reset link sent to email",
            resetToken, // Remove in production
        };
    } catch (error) {
        throw new Error(`Password reset failed: ${error.message}`);
    }
}

// Confirm password reset
async confirmPasswordReset(resetToken, newPassword) {
    try {
        const hashedToken = crypto
            .createHash("sha256")
```

```
        .update(resetToken)
        .digest("hex");

    const user = await User.findOne({
        passwordResetToken: hashedToken,
        passwordResetExpires: { $gt: Date.now() },
    });

    if (!user) {
        throw new Error("Invalid or expired reset token");
    }

    // Update password and clear reset token
    user.password = newPassword;
    user.passwordResetToken = undefined;
    user.passwordResetExpires = undefined;
    await user.save();

    // Revoke all refresh tokens
    await this.revokeUserRefreshTokens(user._id);

    return { message: "Password reset successfully" };
} catch (error) {
    throw new Error(`Password reset confirmation failed: ${error.message}`);
}
}

// Get user permissions based on role
getUserPermissions(role) {
    const permissions = {
        user: [
            "tasks:read:own",
            "tasks:create",
            "tasks:update:own",
            "tasks:delete:own",
            "profile:read:own",
            "profile:update:own",
        ],
        moderator: [
            "tasks:read:all",
            "tasks:create",
            "tasks:update:all",
            "tasks:delete:own",
            "users:read:all",
            "profile:read:own",
            "profile:update:own",
        ],
        admin: ["tasks:*", "users:*", "analytics:read", "system:manage"],
    };

    return permissions[role] || permissions.user;
}

// Clean up expired tokens (run periodically)
```



```

    async cleanupExpiredTokens() {
      try {
        const result = await RefreshToken.deleteMany({
          $or: [
            { expiresAt: { $lt: new Date() } },
            {
              isActive: false,
              revokedAt: { $lt: new Date(Date.now() - 30 * 24 * 60 * 60 * 1000) },
            }, // 30 days old
          ],
        });

        console.log(`Cleaned up ${result.deletedCount} expired tokens`);
        return result.deletedCount;
      } catch (error) {
        console.error("Token cleanup failed:", error);
        throw error;
      }
    }
  }

  module.exports = new AuthService();

```

## Refresh Token Model

```

// models/RefreshToken.js - Refresh Token Schema
const mongoose = require("mongoose");

const refreshTokenSchema = new mongoose.Schema(
  {
    token: {
      type: String,
      required: true,
      unique: true,
      index: true,
    },
    userId: {
      type: mongoose.Schema.Types.ObjectId,
      ref: "User",
      required: true,
      index: true,
    },
    isActive: {
      type: Boolean,
      default: true,
      index: true,
    },
    expiresAt: {
      type: Date,
      required: true,
      index: { expireAfterSeconds: 0 }, // MongoDB TTL index
    },
  },

```

```

    },
    revokedAt: {
      type: Date,
    },
    deviceInfo: {
      userAgent: String,
      ip: String,
      platform: String,
      browser: String,
    },
    lastUsed: {
      type: Date,
      default: Date.now,
    },
  },
  {
    timestamps: true,
  }
);

// Compound indexes
refreshTokenSchema.index({ userId: 1, isActive: 1 });
refreshTokenSchema.index({ token: 1, isActive: 1 });

// Update lastUsed when token is accessed
refreshTokenSchema.methods.updateLastUsed = function () {
  this.lastUsed = new Date();
  return this.save({ validateBeforeSave: false });
};

// Static method to find active tokens for user
refreshTokenSchema.statics.findActiveTokensForUser = function (userId) {
  return this.find({
    userId,
    isActive: true,
    expiresAt: { $gt: new Date() },
  }).sort({ lastUsed: -1 });
};

module.exports = mongoose.model("RefreshToken", refreshTokenSchema);

```

## Authentication Middleware

```

// middleware/auth.js - Authentication and Authorization Middleware
const AuthService = require("../services/AuthService");
const User = require("../models/User");

class AuthMiddleware {
  // Extract token from request
  static extractToken(req) {
    // Check Authorization header (Bearer token)

```

```
const authHeader = req.headers.authorization;
if (authHeader && authHeader.startsWith("Bearer ")) {
  return authHeader.substring(7);
}

// Check query parameter
if (req.query.token) {
  return req.query.token;
}

// Check cookies
if (req.cookies && req.cookies.accessToken) {
  return req.cookies.accessToken;
}

return null;
}

// Authentication middleware - requires valid token
static authenticate() {
  return async (req, res, next) => {
    try {
      const token = AuthMiddleware.extractToken(req);

      if (!token) {
        return res.status(401).json({
          success: false,
          error: "Unauthorized",
          message: "Access token is required",
        });
      }

      // Verify token
      const decoded = AuthService.verifyAccessToken(token);

      // Get fresh user data
      const user = await User.findById(decoded.sub);
      if (!user || !user.isActive) {
        return res.status(401).json({
          success: false,
          error: "Unauthorized",
          message: "User not found or inactive",
        });
      }

      // Attach user and token info to request
      req.user = user;
      req.token = decoded;
      req.permissions = decoded.permissions || [];

      next();
    } catch (error) {
      return res.status(401).json({
        success: false,
```

```
        error: "Unauthorized",
        message: error.message,
    });
    }
};
}

// Optional authentication - doesn't fail if no token
static optionalAuth() {
    return async (req, res, next) => {
        try {
            const token = AuthMiddleware.extractToken(req);

            if (token) {
                const decoded = AuthService.verifyAccessToken(token);
                const user = await User.findById(decoded.sub);

                if (user && user.isActive) {
                    req.user = user;
                    req.token = decoded;
                    req.permissions = decoded.permissions || [];
                }
            }
        } catch (error) {
            // Ignore token errors for optional auth
            console.warn("Optional auth token error:", error.message);
        }

        next();
    };
}

// Role-based authorization
static authorize(allowedRoles = []) {
    return (req, res, next) => {
        if (!req.user) {
            return res.status(401).json({
                success: false,
                error: "Unauthorized",
                message: "Authentication required",
            });
        }

        if (allowedRoles.length > 0 && !allowedRoles.includes(req.user.role)) {
            return res.status(403).json({
                success: false,
                error: "Forbidden",
                message: "Insufficient permissions",
            });
        }

        next();
    };
}
```

```
// Permission-based authorization
static requirePermission(permission) {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({
        success: false,
        error: "Unauthorized",
        message: "Authentication required",
      });
    }

    if (!AuthMiddleware.hasPermission(req.permissions, permission)) {
      return res.status(403).json({
        success: false,
        error: "Forbidden",
        message: `Permission required: ${permission}`,
      });
    }

    next();
  };
}

// Resource ownership authorization
static requireOwnership(resourceField = "userId") {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({
        success: false,
        error: "Unauthorized",
        message: "Authentication required",
      });
    }

    // Admin can access any resource
    if (req.user.role === "admin") {
      return next();
    }

    // Check if user owns the resource
    const resourceUserId =
      req.params[resourceField] || req.body[resourceField];

    if (resourceUserId && resourceUserId !== req.user._id.toString()) {
      return res.status(403).json({
        success: false,
        error: "Forbidden",
        message: "Access denied to this resource",
      });
    }

    next();
  };
}
```

```
}

// Check if user has specific permission
static hasPermission(userPermissions, requiredPermission) {
  if (!userPermissions || userPermissions.length === 0) {
    return false;
  }

  // Check for exact match
  if (userPermissions.includes(requiredPermission)) {
    return true;
  }

  // Check for wildcard permissions
  const [resource, action, scope] = requiredPermission.split(":");

  // Check for resource-level wildcard (e.g., 'tasks:*')
  if (userPermissions.includes(`${resource}:*`)) {
    return true;
  }

  // Check for action-level wildcard (e.g., 'tasks:read:*')
  if (scope && userPermissions.includes(`${resource}:${action}:*`)) {
    return true;
  }

  return false;
}

// Rate limiting for auth endpoints
static createAuthRateLimit() {
  const rateLimit = require("express-rate-limit");

  return rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 5, // 5 attempts per window
    message: {
      success: false,
      error: "Too Many Requests",
      message: "Too many authentication attempts. Please try again later.",
    },
    standardHeaders: true,
    legacyHeaders: false,
    skipSuccessfulRequests: true,
  });
}

// Account lockout middleware
static accountLockout() {
  return async (req, res, next) => {
    try {
      const { identifier } = req.body;

      if (identifier) {
```

```

        const user = await User.findOne({
          $or: [
            { email: identifier.toLowerCase() },
            { username: identifier },
          ],
        });

        if (user && user.isLocked) {
          return res.status(423).json({
            success: false,
            error: "Account Locked",
            message:
              "Account is temporarily locked due to multiple failed login
attempts",
          });
        }

        next();
      } catch (error) {
        next(error);
      }
    };
  }
}

module.exports = AuthMiddleware;

```

## Session-Based Authentication Alternative

```

// middleware/sessionAuth.js - Session-Based Authentication
const session = require("express-session");
const MongoStore = require("connect-mongo");
const User = require("../models/User");

class SessionAuth {
  // Configure session middleware
  static configureSession() {
    return session({
      secret: process.env.SESSION_SECRET || "your-session-secret",
      resave: false,
      saveUninitialized: false,
      store: MongoStore.create({
        mongoUrl: process.env.MONGODB_URI,
        touchAfter: 24 * 3600, // lazy session update
        ttl: 7 * 24 * 60 * 60, // 7 days
      }),
      cookie: {
        secure: process.env.NODE_ENV === "production", // HTTPS only in production
        httpOnly: true, // Prevent XSS
        maxAge: 7 * 24 * 60 * 60 * 1000, // 7 days
      },
    });
  }
}

```

```
    sameSite: "strict", // CSRF protection
  },
  name: "sessionId", // Don't use default session name
});
}

// Login user (session-based)
static async login(req, res, next) {
  try {
    const { identifier, password } = req.body;

    // Find user
    const user = await User.findOne({
      $or: [{ email: identifier.toLowerCase() }, { username: identifier }],
      isActive: true,
    }).select("+password");

    if (!user || !(await user.comparePassword(password))) {
      return res.status(401).json({
        success: false,
        error: "Unauthorized",
        message: "Invalid credentials",
      });
    }

    // Create session
    req.session.userId = user._id;
    req.session.role = user.role;
    req.session.loginTime = new Date();

    // Update last login
    await user.updateLastLogin();

    res.json({
      success: true,
      data: {
        user: user.toPublicJSON(),
        sessionId: req.sessionID,
      },
      message: "Login successful",
    });
  } catch (error) {
    next(error);
  }
}

// Logout user (session-based)
static logout(req, res, next) {
  req.session.destroy((err) => {
    if (err) {
      return next(err);
    }

    res.clearCookie("sessionId");
  });
}
```



```
    res.json({
      success: true,
      message: "Logout successful",
    });
  });
}

// Authentication middleware (session-based)
static authenticate() {
  return async (req, res, next) => {
    try {
      if (!req.session.userId) {
        return res.status(401).json({
          success: false,
          error: "Unauthorized",
          message: "Please log in to access this resource",
        });
      }

      // Get user data
      const user = await User.findById(req.session.userId);
      if (!user || !user.isActive) {
        req.session.destroy(() => {});
        return res.status(401).json({
          success: false,
          error: "Unauthorized",
          message: "User not found or inactive",
        });
      }

      req.user = user;
      next();
    } catch (error) {
      next(error);
    }
  };
}

// Authorization middleware (session-based)
static authorize(allowedRoles = []) {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({
        success: false,
        error: "Unauthorized",
        message: "Authentication required",
      });
    }

    if (allowedRoles.length > 0 && !allowedRoles.includes(req.user.role)) {
      return res.status(403).json({
        success: false,
        error: "Forbidden",
        message: "Insufficient permissions",
      });
    }
  };
}
```

```

    });
  }

  next();
};
}
}

module.exports = SessionAuth;

```

## Authentication Routes

```

// routes/auth.js - Authentication Routes
const express = require("express");
const { body, validationResult } = require("express-validator");
const AuthService = require("../services/AuthService");
const AuthMiddleware = require("../middleware/auth");
const ValidationMiddleware = require("../middleware/validation");

const router = express.Router();

// Validation rules
const registerValidation = [
  body("username")
    .isLength({ min: 3, max: 30 })
    .matches(/^[a-zA-Z0-9_]+$/)
    .withMessage(
      "Username must be 3-30 characters and contain only letters, numbers, and underscores"
    ),
  body("email")
    .isEmail()
    .normalizeEmail()
    .withMessage("Please provide a valid email"),
  body("password")
    .isLength({ min: 8 })
    .matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]/)
    .withMessage(
      "Password must be at least 8 characters with uppercase, lowercase, number, and special character"
    ),
  body("firstName")
    .trim()
    .isLength({ min: 1, max: 50 })
    .withMessage("First name is required and must be less than 50 characters"),
  body("lastName")
    .trim()
    .isLength({ min: 1, max: 50 })
    .withMessage("Last name is required and must be less than 50 characters"),
];

```

```
const loginValidation = [
  body("identifier").notEmpty().withMessage("Email or username is required"),
  body("password").notEmpty().withMessage("Password is required"),
];

const changePasswordValidation = [
  body("currentPassword")
    .notEmpty()
    .withMessage("Current password is required"),
  body("newPassword")
    .isLength({ min: 8 })
    .matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]/)
    .withMessage(
      "New password must be at least 8 characters with uppercase, lowercase,
      number, and special character"
    ),
];

// Apply rate limiting to auth routes
router.use(AuthMiddleware.createAuthRateLimit());

// Register
router.post(
  "/register",
  registerValidation,
  ValidationMiddleware.handleValidationErrors,
  async (req, res, next) => {
    try {
      const result = await AuthService.register(req.body);

      // Set refresh token as httpOnly cookie
      res.cookie("refreshToken", result.tokens.refreshToken, {
        httpOnly: true,
        secure: process.env.NODE_ENV === "production",
        sameSite: "strict",
        maxAge: 7 * 24 * 60 * 60 * 1000, // 7 days
      });

      res.status(201).json({
        success: true,
        data: {
          user: result.user,
          accessToken: result.tokens.accessToken,
          expiresIn: result.tokens.expiresIn,
        },
        message: "Registration successful",
      });
    } catch (error) {
      next(error);
    }
  }
);

// Login
```

```
router.post(
  "/login",
  loginValidation,
  ValidationMiddleware.handleValidationErrors,
  AuthMiddleware.accountLockout(),
  async (req, res, next) => {
    try {
      const result = await AuthService.login(req.body);

      // Set refresh token as httpOnly cookie
      res.cookie("refreshToken", result.tokens.refreshToken, {
        httpOnly: true,
        secure: process.env.NODE_ENV === "production",
        sameSite: "strict",
        maxAge: 7 * 24 * 60 * 60 * 1000, // 7 days
      });

      res.json({
        success: true,
        data: {
          user: result.user,
          accessToken: result.tokens.accessToken,
          expiresIn: result.tokens.expiresIn,
        },
        message: "Login successful",
      });
    } catch (error) {
      next(error);
    }
  }
);

// Refresh token
router.post("/refresh", async (req, res, next) => {
  try {
    const refreshToken = req.cookies.refreshToken || req.body.refreshToken;

    if (!refreshToken) {
      return res.status(401).json({
        success: false,
        error: "Unauthorized",
        message: "Refresh token is required",
      });
    }

    const result = await AuthService.refreshAccessToken(refreshToken);

    // Update refresh token cookie if rotated
    if (result.refreshToken !== refreshToken) {
      res.cookie("refreshToken", result.refreshToken, {
        httpOnly: true,
        secure: process.env.NODE_ENV === "production",
        sameSite: "strict",
        maxAge: 7 * 24 * 60 * 60 * 1000, // 7 days
      });
    }
  }
});
```

```
    });
  }

  res.json({
    success: true,
    data: {
      accessToken: result.accessToken,
      expiresIn: result.expiresIn,
    },
    message: "Token refreshed successfully",
  });
} catch (error) {
  // Clear invalid refresh token
  res.clearCookie("refreshToken");
  next(error);
}
});

// Logout
router.post(
  "/logout",
  AuthMiddleware.optionalAuth(),
  async (req, res, next) => {
    try {
      const refreshToken = req.cookies.refreshToken || req.body.refreshToken;

      await AuthService.logout(refreshToken);

      // Clear refresh token cookie
      res.clearCookie("refreshToken");

      res.json({
        success: true,
        message: "Logout successful",
      });
    } catch (error) {
      next(error);
    }
  }
);

// Logout from all devices
router.post(
  "/logout-all",
  AuthMiddleware.authenticate(),
  async (req, res, next) => {
    try {
      await AuthService.logoutAll(req.user._id);

      // Clear refresh token cookie
      res.clearCookie("refreshToken");

      res.json({
        success: true,
```

```
        message: "Logged out from all devices",
    });
} catch (error) {
    next(error);
}
}
);

// Change password
router.post(
    "/change-password",
    AuthMiddleware.authenticate(),
    changePasswordValidation,
    ValidationMiddleware.handleValidationErrors,
    async (req, res, next) => {
        try {
            const { currentPassword, newPassword } = req.body;

            await AuthService.changePassword(
                req.user._id,
                currentPassword,
                newPassword
            );

            // Clear refresh token cookie to force re-login
            res.clearCookie("refreshToken");

            res.json({
                success: true,
                message: "Password changed successfully. Please log in again.",
            });
        } catch (error) {
            next(error);
        }
    }
);

// Request password reset
router.post(
    "/forgot-password",
    body("email")
        .isEmail()
        .normalizeEmail()
        .withMessage("Please provide a valid email"),
    ValidationMiddleware.handleValidationErrors,
    async (req, res, next) => {
        try {
            const { email } = req.body;

            const result = await AuthService.resetPassword(email);

            res.json({
                success: true,
                message: result.message,
            });
        } catch (error) {
            next(error);
        }
    }
);
```

```

        ...(process.env.NODE_ENV === "development" && {
            resetToken: result.resetToken,
        }),
    });
} catch (error) {
    next(error);
}
}
);

// Confirm password reset
router.post(
    "/reset-password",
    body("resetToken").notEmpty().withMessage("Reset token is required"),
    body("newPassword")
        .isLength({ min: 8 })
        .matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]/)
        .withMessage(
            "Password must be at least 8 characters with uppercase, lowercase, number, and special character"
        ),
    ValidationMiddleware.handleValidationErrors,
    async (req, res, next) => {
        try {
            const { resetToken, newPassword } = req.body;

            await AuthService.confirmPasswordReset(resetToken, newPassword);

            res.json({
                success: true,
                message:
                    "Password reset successfully. Please log in with your new password.",
            });
        } catch (error) {
            next(error);
        }
    }
);

// Get current user profile
router.get("/me", AuthMiddleware.authenticate(), (req, res) => {
    res.json({
        success: true,
        data: {
            user: req.user.toPublicJSON(),
            permissions: req.permissions,
            tokenInfo: {
                issuedAt: new Date(req.token.iat * 1000),
                expiresAt: new Date(req.token.exp * 1000),
            },
        },
    });
});
});

```

```
// Verify token (for client-side validation)
router.post("/verify", AuthMiddleware.authenticate(), (req, res) => {
  res.json({
    success: true,
    data: {
      valid: true,
      user: req.user.toPublicJSON(),
      permissions: req.permissions,
    },
    message: "Token is valid",
  });
});

module.exports = router;
```

## Real-World Use Case

### Multi-Factor Authentication (MFA)

```
// services/MFAService.js - Multi-Factor Authentication
const speakeasy = require("speakeasy");
const QRCode = require("qrcode");
const crypto = require("crypto");
const User = require("../models/User");

class MFAService {
  // Generate TOTP secret for user
  async generateTOTPSecret(userId) {
    try {
      const user = await User.findById(userId);
      if (!user) {
        throw new Error("User not found");
      }

      const secret = speakeasy.generateSecret({
        name: `TaskApp (${user.email})`,
        issuer: "TaskApp",
        length: 32,
      });

      // Store secret temporarily (not activated until verified)
      user.mfa = {
        secret: secret.base32,
        enabled: false,
        backupCodes: this.generateBackupCodes(),
      };

      await user.save();

      // Generate QR code
      const qrCodeUrl = await QRCode.toDataURL(secret.otpauth_url);
```



```
    return {
      secret: secret.base32,
      qrCode: qrCodeUrl,
      backupCodes: user.mfa.backupCodes,
    };
  } catch (error) {
    throw new Error(`MFA setup failed: ${error.message}`);
  }
}

// Verify and enable TOTP
async enableTOTP(userId, token) {
  try {
    const user = await User.findById(userId);
    if (!user || !user.mfa || !user.mfa.secret) {
      throw new Error("MFA not set up");
    }

    const verified = speakeasy.totp.verify({
      secret: user.mfa.secret,
      encoding: "base32",
      token,
      window: 2, // Allow 2 time steps (60 seconds)
    });

    if (!verified) {
      throw new Error("Invalid verification code");
    }

    // Enable MFA
    user.mfa.enabled = true;
    await user.save();

    return {
      message: "MFA enabled successfully",
      backupCodes: user.mfa.backupCodes,
    };
  } catch (error) {
    throw new Error(`MFA verification failed: ${error.message}`);
  }
}

// Verify TOTP token
async verifyTOTP(userId, token) {
  try {
    const user = await User.findById(userId);
    if (!user || !user.mfa || !user.mfa.enabled) {
      throw new Error("MFA not enabled");
    }

    // Check if it's a backup code
    if (user.mfa.backupCodes.includes(token)) {
      // Remove used backup code
    }
  }
}
```

```
        user.mfa.backupCodes = user.mfa.backupCodes.filter(
            (code) => code !== token
        );
        await user.save();
        return true;
    }

    // Verify TOTP
    const verified = speakeasy.totp.verify({
        secret: user.mfa.secret,
        encoding: "base32",
        token,
        window: 2,
    });

    return verified;
} catch (error) {
    throw new Error(`MFA verification failed: ${error.message}`);
}
}

// Disable MFA
async disableMFA(userId, password) {
    try {
        const user = await User.findById(userId).select("+password");
        if (!user) {
            throw new Error("User not found");
        }

        // Verify password
        const isPasswordValid = await user.comparePassword(password);
        if (!isPasswordValid) {
            throw new Error("Invalid password");
        }

        // Disable MFA
        user.mfa = {
            enabled: false,
            secret: null,
            backupCodes: [],
        };

        await user.save();

        return { message: "MFA disabled successfully" };
    } catch (error) {
        throw new Error(`MFA disable failed: ${error.message}`);
    }
}

// Generate backup codes
generateBackupCodes(count = 10) {
    const codes = [];
    for (let i = 0; i < count; i++) {
```

```
        codes.push(crypto.randomBytes(4).toString("hex").toUpperCase());
    }
    return codes;
}

// Regenerate backup codes
async regenerateBackupCodes(userId) {
    try {
        const user = await User.findById(userId);
        if (!user || !user.mfa || !user.mfa.enabled) {
            throw new Error("MFA not enabled");
        }

        user.mfa.backupCodes = this.generateBackupCodes();
        await user.save();

        return {
            message: "Backup codes regenerated",
            backupCodes: user.mfa.backupCodes,
        };
    } catch (error) {
        throw new Error(`Backup code regeneration failed: ${error.message}`);
    }
}

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

## Best Practices

### 1. Password Security

```
// Use strong hashing with salt
const bcrypt = require("bcryptjs");
const saltRounds = 12; // Increase for better security

// Hash password
const hashedPassword = await bcrypt.hash(password, saltRounds);

// Verify password
const isValid = await bcrypt.compare(password, hashedPassword);
```

### 2. JWT Security

```
// Use strong secrets and proper algorithms
const jwt = require("jsonwebtoken");

const token = jwt.sign(
    { userId, role },
```

```
process.env.JWT_SECRET, // Use strong, random secret
{
  expiresIn: "15m", // Short expiry for access tokens
  algorithm: "HS256", // Specify algorithm
  issuer: "your-app",
  audience: "your-users",
}
);
```

### 3. Rate Limiting

```
// Implement aggressive rate limiting for auth endpoints
const authRateLimit = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 5, // 5 attempts per window
  skipSuccessfulRequests: true,
  message: "Too many authentication attempts",
});
```

### 4. Input Validation

```
// Validate all authentication inputs
const { body } = require("express-validator");

const loginValidation = [
  body("email").isEmail().normalizeEmail(),
  body("password")
    .isLength({ min: 8 })
    .matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/),
];
```

### 5. Secure Headers

```
// Set security headers
app.use(
  helmet({
    contentSecurityPolicy: {
      directives: {
        defaultSrc: ["'self'"],
        scriptSrc: ["'self'", "'unsafe-inline'"],
      },
    },
  },
  hsts: {
    maxAge: 31536000,
    includeSubDomains: true,
  },
);
```

```
  })  
);
```

## Summary

Authentication and authorization are fundamental security components:

### Authentication Methods:

- **JWT:** Stateless, scalable, cross-domain support
- **Sessions:** Server-side control, easy revocation
- **Multi-Factor:** Enhanced security with TOTP/backup codes
- **OAuth:** Third-party authentication integration

### Authorization Patterns:

- **Role-Based Access Control (RBAC):** Simple role hierarchy
- **Permission-Based:** Granular permission system
- **Resource Ownership:** User-specific resource access
- **Attribute-Based:** Context-aware authorization

### Security Best Practices:

- Use strong password hashing (bcrypt with high salt rounds)
- Implement proper JWT security (strong secrets, short expiry)
- Apply rate limiting to authentication endpoints
- Validate and sanitize all inputs
- Use secure headers and HTTPS
- Implement account lockout mechanisms
- Log security events for monitoring

### Advanced Features:

- Token refresh and rotation
- Multi-factor authentication
- Password reset flows
- Session management
- Device tracking
- Security event logging

Proper authentication and authorization implementation ensures application security while providing a smooth user experience. Next, we'll explore file upload and handling, building upon the secure foundation to manage user-generated content safely.