The Complete Node.js Ecosystem Guide

> A comprehensive reference for Node.js, Express, MongoDB, and template engines

Table of Contents

Introduction

This guide aims to provide a comprehensive overview of the Node.js ecosystem, focusing on the most common and powerful tools used in modern Node.js development. Whether you're relearning these technologies or starting fresh, this guide will take you from fundamentals to advanced concepts with practical examples and best practices.

The Node.js ecosystem has evolved into a robust platform for building server-side applications, APIs, and full-stack web applications. This guide covers:

- **Node.js core concepts** and how to leverage its asynchronous, event-driven architecture
- **Express.js framework** for building web applications and APIs
- **MongoDB** for data storage and **Mongoose** for elegant MongoDB object modeling
- **Template engines** for server-side rendering and their integration with Express
- **Full-stack integration** patterns and practices
- **Comparison with React** and modern front-end approaches

Let's begin our journey into the Node.js ecosystem!

Node.js Fundamentals

Core Concepts

What is Node.js?

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Unlike traditional server environments where each connection spawns a new thread (consuming system RAM), Node.js operates on a single-threaded event loop, using non-blocking I/O calls. This architecture allows it to handle thousands of concurrent connections with minimal overhead.

The Event Loop

The event loop is the secret behind Node.js's non-blocking I/O model. It works as follows:

1. Node.js initializes the event loop
2. Processes the input script which may make async API calls, schedule timers, or call `process.nextTick()`
3. Processes the event loop:
    - Executes callbacks in the timers phase (setTimeout, setInterval)
    - Executes I/O callbacks
    - Enters the poll phase (retrieves new I/O events)
    - Executes setImmediate() callbacks
    - Processes close callbacks

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

// setTimeout is added to the timer queue
setTimeout(() => {
  console.log('Timeout callback executed');
}, 0);

// nextTick is added to the nextTick queue, which runs after the current operation
// and before the event loop continues
process.nextTick(() => {
  console.log('Next tick executed');
});

// This is part of the current synchronous operation
console.log('End');

// Output:
// Start
// End
// Next tick executed
// Timeout callback executed
```

Understanding the event loop is crucial for writing performant Node.js applications, as it explains how Node.js can be single-threaded yet handle concurrent operations efficiently.

Non-blocking I/O

In traditional blocking I/O operations, a thread is occupied until the operation completes. For example, when reading a file, the thread waits until the file is fully read before continuing execution.

Node.js uses non-blocking I/O operations, which means it can start an I/O operation and immediately return to the event loop to handle other tasks. When the I/O operation completes, a callback function is triggered to handle the result.

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

// Blocking (synchronous) approach
try {
```

```
  const data = fs.readFileSync('/path/to/file', 'utf8');
  console.log(data);
  // Code here waits until file is read
  console.log('File reading complete');
} catch (err) {
  console.error(err);
}

// Non-blocking (asynchronous) approach
fs.readFile('/path/to/file', 'utf8', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data);
  console.log('File reading complete');
});
// Code here continues executing while file is being read
console.log('Reading file...');
```

Modules in Node.js

Node.js uses the CommonJS module system (though ES modules are now supported too). Modules help organize code into reusable, encapsulated units.

CommonJS Module Pattern:

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

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

// Alternatively:
module.exports = {
  add: function (a, b) {
    return a + b;
  },
  subtract: function (a, b) {
    return a - b;
  },
};
```

Using modules:

```
// app.js
const math = require('./lib/math');
console.log(math.add(5, 3)); // 8
```

ES Modules in Node.js:

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

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

Using ES modules:

```
// app.mjs
import { add, subtract } from './lib/math.mjs';
console.log(add(5, 3)); // 8
```

To use ES modules, either use the `.mjs` extension or add `"type": "module"` to your `package.json`.

Setting Up Node.js Projects

Installing Node.js

Download and install Node.js from nodejs.org. It's recommended to use a version manager like `nvm` (Node Version Manager) to easily switch between Node.js versions.

Installing nvm:

```
# For Unix/macOS systems
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.3/install.sh | bash

# For Windows, use nvm-windows:
# https://github.com/coreybutler/nvm-windows
```

Installing Node.js with nvm:

```
nvm install 16  # Install Node.js v16
nvm use 16      # Use Node.js v16
```

npm: Node Package Manager

npm is installed with Node.js and is used to manage packages and dependencies.

Key npm commands:

```
npm init                # Initialize a new Node.js project
npm install <package>   # Install a package locally
npm install -g <package> # Install a package globally
npm install --save-dev <package> # Install a development dependency
npm uninstall <package> # Uninstall a package
npm update              # Update packages
npm run <script>        # Run a script defined in package.json
```

Understanding package.json

The `package.json` file is the heart of a Node.js project, containing metadata about the project and its dependencies.

Example package.json:

```
{
  "name": "my-node-app",
  "version": "1.0.0",
  "description": "A sample Node.js application",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "dev": "nodemon index.js",
    "test": "jest"
  },
  "keywords": ["node", "example"],
  "author": "Your Name",
  "license": "MIT",
  "dependencies": {
    "express": "^4.18.2",
    "mongoose": "^7.0.0"
  },
  "devDependencies": {
    "nodemon": "^2.0.20",
    "jest": "^29.3.1"
  }
}
```

Understanding package versioning:

- `^4.18.2`: Compatible with version 4.18.2 up to, but not including, version 5.0.0
- `~4.18.2`: Compatible with version 4.18.2 up to, but not including, version 4.19.0
- `4.18.2`: Exactly version 4.18.2

npm scripts

npm scripts are custom commands defined in the `scripts` section of `package.json`.

```
"scripts": {
  "start": "node index.js",
  "dev": "nodemon index.js",
  "build": "webpack",
  "test": "jest",
  "lint": "eslint ."
}
```

Run these scripts with `npm run <script-name>` (or just `npm start` for the "start" script).

Managing Dependencies

package.json vs package-lock.json:

- `package.json`: Declares the dependencies and their acceptable version ranges
- `package-lock.json`: Locks down the exact versions of all dependencies and their dependencies

Dependencies vs devDependencies:

- `dependencies`: Packages required in production
- `devDependencies`: Packages needed only for development and testing

Built-in Modules

Node.js comes with several built-in modules that provide essential functionality.

File System (fs)

The `fs` module provides an API for interacting with the file system.

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

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

// Writing to a file asynchronously
fs.writeFile('file.txt', 'Hello, Node.js!', (err) => {
  if (err) {
    console.error('Error writing file:', err);
    return;
```

```
  }
  console.log('File written successfully');
});

// Promise-based API (Node.js v10+)
const fsPromises = require('fs').promises;

async function readAndWriteFile() {
  try {
    const data = await fsPromises.readFile('file.txt', 'utf8');
    console.log(data);

    await fsPromises.writeFile('new-file.txt', 'New content');
    console.log('File written successfully');
  } catch (err) {
    console.error('Error:', err);
  }
}

readAndWriteFile();
```

Path

The `path` module provides utilities for working with file and directory paths.

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

// Joining path components
const fullPath = path.join(__dirname, 'public', 'index.html');
console.log(fullPath); // /path/to/current/directory/public/index.html

// Getting the file extension
const ext = path.extname('file.txt');
console.log(ext); // .txt

// Parsing a path into components
const pathInfo = path.parse('/home/user/file.txt');
console.log(pathInfo);
// {
//   root: '/',
//   dir: '/home/user',
//   base: 'file.txt',
//   ext: '.txt',
//   name: 'file'
// }

// Resolving an absolute path
const absPath = path.resolve('some/relative/path');
console.log(absPath); // /current/working/directory/some/relative/path
```

HTTP/HTTPS

The `http` and `https` modules allow Node.js to transfer data over HTTP/HTTPS.

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

// Creating a basic HTTP server
const server = http.createServer((req, res) => {
  // Get request URL and method
  const { url, method } = req;

  // Set response headers
  res.setHeader('Content-Type', 'text/html');

  // Route handling
  if (url === '/') {
    res.statusCode = 200;
    res.end('<h1>Home Page</h1>');
  } else if (url === '/about') {
    res.statusCode = 200;
    res.end('<h1>About Page</h1>');
  } else {
    res.statusCode = 404;
    res.end('<h1>Page Not Found</h1>');
  }
});

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

// Making an HTTP request
http
  .get('http://example.com', (res) => {
    let data = '';

    // Receive data chunks
    res.on('data', (chunk) => {
      data += chunk;
    });

    // Complete response received
    res.on('end', () => {
      console.log(data);
    });
  })
  .on('error', (err) => {
    console.error('Error:', err.message);
  });
```

Events

The `events` module provides an event-driven architecture implementation.

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

// Create a custom event emitter
class MyEmitter extends EventEmitter {}
const myEmitter = new MyEmitter();

// Register an event listener
myEmitter.on('event', (a, b) => {
  console.log('Event triggered:', a, b);
});

// Emit an event
myEmitter.emit('event', 'arg1', 'arg2');
// Output: Event triggered: arg1 arg2

// One-time event listener
myEmitter.once('oneTimeEvent', () => {
  console.log('This will be triggered only once');
});

myEmitter.emit('oneTimeEvent'); // Listener triggered
myEmitter.emit('oneTimeEvent'); // No output, listener already removed
```

URL

The `url` module provides utilities for URL resolution and parsing.

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

// Legacy API
const myUrl = url.parse('https://example.com:8080/path?query=string#hash');
console.log(myUrl);
// {
//   protocol: 'https:',
//   hostname: 'example.com',
//   port: '8080',
//   pathname: '/path',
//   search: '?query=string',
//   hash: '#hash',
//   ...
// }

// WHATWG URL API (modern)
const newUrl = new URL('https://example.com:8080/path?query=string#hash');
console.log(newUrl.hostname); // example.com
console.log(newUrl.searchParams.get('query')); // string
```

Other Important Built-in Modules

- `os`: Operating system-related utilities
- `util`: Utility functions
- `crypto`: Cryptographic functionality
- `stream`: For working with streaming data
- `buffer`: For handling binary data
- `child_process`: For spawning child processes

Asynchronous Programming Patterns

Node.js is built around asynchronous programming. Understanding different patterns for handling asynchronous code is essential.

Callbacks

Callbacks are the most basic way to handle asynchronous operations in Node.js.

```
function fetchData(id, callback) {
  // Simulating an async operation
  setTimeout(() => {
    const data = { id, name: `Item ${id}` };
    callback(null, data); // Success: error is null, data is returned
  }, 1000);
}

fetchData(123, (err, data) => {
  if (err) {
    console.error('Error:', err);
    return;
  }
  console.log('Data:', data);
});
```

Callback Hell and How to Avoid It:

Nested callbacks can lead to "callback hell" (also known as the "pyramid of doom"):

```
fetchUser(userId, (err, user) => {
  if (err) {
    console.error(err);
    return;
  }

  fetchUserPosts(user.id, (err, posts) => {
    if (err) {
      console.error(err);
```

```
      return;
    }

    fetchPostComments(posts[0].id, (err, comments) => {
      if (err) {
        console.error(err);
        return;
      }

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

Solutions to callback hell:

1. Named functions instead of anonymous functions
2. Modularization (breaking code into smaller functions)
3. Using Promises or async/await

Promises

Promises provide a more elegant way to handle asynchronous operations.

```
function fetchData(id) {
  return new Promise((resolve, reject) => {
    // Simulate async operation
    setTimeout(() => {
      const data = { id, name: `Item ${id}` };
      if (data) {
        resolve(data); // Success
      } else {
        reject(new Error('Data not found')); // Error
      }
    }, 1000);
  });
}

// Using the Promise
fetchData(123)
  .then((data) => {
    console.log('Data:', data);
    return fetchData(456); // Chain another Promise
  })
  .then((moreData) => {
    console.log('More data:', moreData);
  })
  .catch((err) => {
    console.error('Error:', err);
  })
  .finally(() => {
```

```
    console.log('Operation complete');
  });

// Parallel Promise execution
Promise.all([fetchData(1), fetchData(2), fetchData(3)])
  .then((results) => {
    console.log('All results:', results);
  })
  .catch((err) => {
    console.error('One of the promises failed:', err);
  });

// Get first resolved Promise
Promise.race([fetchData(1), fetchData(2), fetchData(3)])
  .then((firstResult) => {
    console.log('First result:', firstResult);
  })
  .catch((err) => {
    console.error('Error:', err);
  });
```

Async/Await

Async/await (introduced in Node.js 7.6) provides a more intuitive way to work with Promises.

```
// Function declaration with async
async function getData() {
  try {
    // await can only be used inside async functions
    const data = await fetchData(123);
    console.log('Data:', data);

    const moreData = await fetchData(456);
    console.log('More data:', moreData);

    return { data, moreData };
  } catch (err) {
    console.error('Error:', err);
    throw err; // Re-throw or handle error
  } finally {
    console.log('Operation complete');
  }
}

// Using an async function
getData()
  .then((result) => {
    console.log('Result:', result);
  })
  .catch((err) => {
    console.error('Caught error:', err);
```

```
  });

// Parallel execution with async/await
async function getMultipleData() {
  try {
    // Start all async operations in parallel
    const dataPromise1 = fetchData(1);
    const dataPromise2 = fetchData(2);
    const dataPromise3 = fetchData(3);

    // Await all results
    const data1 = await dataPromise1;
    const data2 = await dataPromise2;
    const data3 = await dataPromise3;

    return [data1, data2, data3];
  } catch (err) {
    console.error('Error:', err);
    throw err;
  }
}

// Cleaner parallel execution using Promise.all with async/await
async function getAllData() {
  try {
    const results = await Promise.all([
      fetchData(1),
      fetchData(2),
      fetchData(3),
    ]);
    return results;
  } catch (err) {
    console.error('Error:', err);
    throw err;
  }
}
```

Error Handling Strategies

Proper error handling is crucial in Node.js applications to prevent crashes and provide meaningful feedback.

Handling Synchronous Errors

```
try {
  const result = JSON.parse('{"invalid": json}');
  console.log(result);
} catch (err) {
  console.error('Error parsing JSON:', err.message);
}
```

Handling Asynchronous Errors with Callbacks

```
fs.readFile('nonexistent-file.txt', (err, data) => {
  if (err) {
    console.error('Error reading file:', err.message);
    return;
  }
  console.log(data);
});
```

Handling Asynchronous Errors with Promises

```
fetchData(123)
  .then((data) => {
    console.log('Data:', data);
  })
  .catch((err) => {
    console.error('Error fetching data:', err.message);
  });
```

Handling Asynchronous Errors with Async/Await

```
async function getData() {
  try {
    const data = await fetchData(123);
    console.log('Data:', data);
  } catch (err) {
    console.error('Error fetching data:', err.message);
  }
}
```

Uncaught Exceptions and Unhandled Rejections

```
// Global handler for uncaught exceptions
process.on('uncaughtException', (err) => {
  console.error('Uncaught Exception:', err);
  // Perform cleanup, log the error, then exit
  process.exit(1); // Exit with failure code
});

// Global handler for unhandled promise rejections
process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled Rejection at:', promise, 'reason:', reason);
  // Optionally, exit the process
```

```
  // process.exit(1);
});
```

Custom Error Classes

```
class DatabaseError extends Error {
  constructor(message, code) {
    super(message);
    this.name = 'DatabaseError';
    this.code = code;
    Error.captureStackTrace(this, this.constructor);
  }
}

class ValidationError extends Error {
  constructor(message, field) {
    super(message);
    this.name = 'ValidationError';
    this.field = field;
    Error.captureStackTrace(this, this.constructor);
  }
}

// Usage
function validateUser(user) {
  if (!user.email) {
    throw new ValidationError('Email is required', 'email');
  }
  // More validation...
}

try {
  validateUser({ name: 'John' }); // Missing email
} catch (err) {
  if (err instanceof ValidationError) {
    console.error(`Validation error on field ${err.field}: ${err.message}`);
  } else {
    console.error('Unknown error:', err);
  }
}
```

Performance Optimization and Debugging

Node.js Performance Best Practices

1. **Avoid blocking the event loop**

```
// Bad: Blocking operation
const result = heavyComputation(data);

// Better: Offload to worker thread or chunk the operation
const { Worker } = require('worker_threads');
const worker = new Worker('./heavy-computation-worker.js');
worker.on('message', (result) => {
  console.log('Computation result:', result);
});
worker.postMessage(data);
```

2. **Use asynchronous methods**

```
// Bad: Blocking I/O
const data = fs.readFileSync('large-file.txt');

// Better: Non-blocking I/O
fs.readFile('large-file.txt', (err, data) => {
  if (err) throw err;
  // Process data
});
```

3. **Implement proper caching**

```
const cache = new Map();

function getUser(id) {
  // Check if data is in cache
  if (cache.has(id)) {
    return Promise.resolve(cache.get(id));
  }

  // If not, fetch from database
  return database.getUser(id).then((user) => {
    // Store in cache for future requests
    cache.set(id, user);
    return user;
  });
}
```

4. **Use streams for large data**

```
// Bad: Reading entire file into memory
fs.readFile('large-file.csv', (err, data) => {
  if (err) throw err;
  processData(data);
```

```
});

// Better: Using streams
const readStream = fs.createReadStream('large-file.csv');
const transformStream = new Transform({
  transform(chunk, encoding, callback) {
    const processed = processChunk(chunk);
    callback(null, processed);
  },
});

readStream.pipe(transformStream).pipe(process.stdout);
```

Debugging Node.js Applications

Using console methods:

```
console.log('Basic logging');
console.error('Error message');
console.warn('Warning message');
console.time('Timer');
// Operations to time
console.timeEnd('Timer'); // Outputs: Timer: 1.234ms
console.table([
  { name: 'John', age: 30 },
  { name: 'Jane', age: 25 },
]);
```

Using the built-in debugger:

```
# Start node with the inspector
node --inspect app.js

# Break on the first line
node --inspect-brk app.js
```

Adding debugger statements in code:

```
function someFunction() {
  const a = 5;
  const b = 10;
  debugger; // Execution will pause here when running with inspector
  return a + b;
}
```

Using Node.js Inspector Client:

```
const inspector = require('inspector');
const session = new inspector.Session();
session.connect();

session.post('Runtime.enable');
session.post('Debugger.enable');

// Set a breakpoint
session.post('Debugger.setBreakpointByUrl', {
  lineNumber: 10,
  url: 'file:///path/to/app.js',
});
```

Using debugging tools:

- Chrome DevTools: Open `chrome://inspect` in Chrome after starting Node.js with `--inspect`
- Visual Studio Code: Set breakpoints and use the debugging panel
- Node.js debugging packages like `debug`

```
// Using the debug package
const debug = require('debug')('app:server');

debug('Server starting up...');
// Only output when DEBUG=app:server environment variable is set
```

Security Best Practices

Input Validation

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

// Validate email
const email = req.body.email;
if (!validator.isEmail(email)) {
  return res.status(400).json({ error: 'Invalid email address' });
}

// Sanitize input
const sanitizedInput = validator.escape(req.body.comment);
```

Preventing Common Vulnerabilities

1. SQL Injection (using parameterized queries)

```
// Bad (vulnerable to SQL injection)
const query = `SELECT * FROM users WHERE username = '${username}'`;

// Good (using parameterized queries with mysql2)
const [rows] = await connection.execute(
  'SELECT * FROM users WHERE username = ?',
  [username]
);
```

2. Cross-Site Scripting (XSS)

```
// Using a package like helmet for Express
const helmet = require('helmet');
app.use(helmet()); // Sets various HTTP headers for security

// Content Security Policy
app.use(
  helmet.contentSecurityPolicy({
    directives: {
      defaultSrc: ["'self'"],
      scriptSrc: ["'self'", 'trusted-scripts.com'],
    },
  })
);
```

3. Cross-Site Request Forgery (CSRF)

```
const csrf = require('csurf');
const csrfProtection = csrf({ cookie: true });

app.use(csrfProtection);

app.get('/form', (req, res) => {
  // Send CSRF token to form
  res.render('form', { csrfToken: req.csrfToken() });
});

app.post('/process', csrfProtection, (req, res) => {
  // CSRF token is validated automatically
  res.send('Form processed');
});
```

4. Secure Cookies

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

```
app.use(
  session({
    secret: 'your-secret-key',
    resave: false,
    saveUninitialized: false,
    cookie: {
      secure: process.env.NODE_ENV === 'production', // Use HTTPS in production
      httpOnly: true, // Prevents client-side JS from reading the cookie
      maxAge: 1000 * 60 * 60 * 24, // 1 day
      sameSite: 'strict', // Restricts cross-site cookie usage
    },
  })
);
```

5. Rate Limiting

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

const apiLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // Limit each IP to 100 requests per windowMs
  message: 'Too many requests from this IP, please try again later',
});

// Apply to all API routes
app.use('/api/', apiLimiter);
```

6. Handling Sensitive Data

```
// Environment variables for sensitive data
require('dotenv').config();

const dbUser = process.env.DB_USER;
const dbPassword = process.env.DB_PASSWORD;

// Hashing passwords with bcrypt
const bcrypt = require('bcrypt');

async function hashPassword(password) {
  const salt = await bcrypt.genSalt(10);
  return bcrypt.hash(password, salt);
}

async function verifyPassword(password, hashedPassword) {
  return bcrypt.compare(password, hashedPassword);
}
```

7. Security Headers

```
app.use((req, res, next) => {
  // Disable X-Powered-By header
  res.removeHeader('X-Powered-By');

  // Add additional security headers
  res.setHeader('X-XSS-Protection', '1; mode=block');
  res.setHeader('X-Content-Type-Options', 'nosniff');
  res.setHeader('X-Frame-Options', 'DENY');

  next();
});
```

8. Dependency Security

```
# Audit dependencies for vulnerabilities
npm audit

# Fix vulnerabilities automatically where possible
npm audit fix

# Use tools like Snyk or npm audit for continuous monitoring
```

Express Framework

Framework Architecture and Core Concepts

What is Express?

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It's designed to make the process of building web applications and APIs much simpler and more intuitive.

Express Application Structure

```
// Basic Express application structure
const express = require('express');
const app = express();

// Middleware
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(express.static('public'));

// Routes
app.get('/', (req, res) => {
  res.send('Hello World!');
});
```

```
app.get('/about', (req, res) => {
  res.send('About page');
});

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

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

Express Application Generator

The Express Application Generator is a tool to quickly create an application skeleton.

```
# Install the generator globally
npm install -g express-generator

# Generate a new Express application
express --view=ejs myapp

# Navigate into the app and install dependencies
cd myapp
npm install

# Start the server
npm start
```

MVC Pattern with Express

Express can be structured following the Model-View-Controller (MVC) pattern:

```
project/
├── models/          # Data models
│   └── user.js      # User model
├── views/           # Templates (EJS, Pug, etc.)
│   ├── index.ejs
│   └── user.ejs
├── controllers/     # Route handlers
│   ├── homeController.js
│   └── userController.js
├── routes/          # Route definitions
│   ├── index.js
```

```
│   └── users.js
├── public/          # Static assets
│   ├── css/
│   ├── js/
│   └── images/
├── middleware/      # Custom middleware
│   └── auth.js
├── config/          # Configuration files
│   └── database.js
├── app.js           # Application entry point
└── package.json
```

Example implementation:

```
// models/user.js
class User {
  constructor(id, name, email) {
    this.id = id;
    this.name = name;
    this.email = email;
  }

  static findById(id) {
    // Database logic to find user
    return new User(id, 'Sample User', 'user@example.com');
  }
}

module.exports = User;

// controllers/userController.js
const User = require('../models/user');

exports.getUser = (req, res) => {
  const user = User.findById(req.params.id);
  res.render('user', { user });
};

// routes/users.js
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');

router.get('/:id', userController.getUser);

module.exports = router;

// app.js
const express = require('express');
const app = express();
const userRoutes = require('./routes/users');
```

```
app.set('view engine', 'ejs');
app.use('/users', userRoutes);

app.listen(3000);
```

Creating and Configuring an Express Application

Basic Setup

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

// Basic configuration
app.set('port', process.env.PORT || 3000);
app.set('view engine', 'ejs');
app.set('views', './views');

// Start server
app.listen(app.get('port'), () => {
  console.log(`Server running on port ${app.get('port')}`);
});
```

Environment-specific Configuration

```
// config/index.js
const development = require('./development');
const production = require('./production');
const test = require('./test');

const env = process.env.NODE_ENV || 'development';

const configs = {
  development,
  production,
  test,
};

module.exports = configs[env];

// Using the configuration
const config = require('./config');
console.log(`Database URL: ${config.database.url}`);
```

Using Environment Variables

```
// Install dotenv
// npm install dotenv

// Load environment variables
require('dotenv').config();

// Access environment variables
const dbUser = process.env.DB_USER;
const dbPassword = process.env.DB_PASSWORD;
const apiKey = process.env.API_KEY;

// Example .env file
// DB_USER=admin
// DB_PASSWORD=secret
// API_KEY=abc123
```

Configuring Express with Middleware

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

const app = express();

// Request logging
app.use(morgan('dev')); // 'combined' for production

// Security headers
app.use(helmet());

// CORS configuration
app.use(
  cors({
    origin: ['https://example.com', 'https://www.example.com'],
    methods: ['GET', 'POST', 'PUT', 'DELETE'],
    allowedHeaders: ['Content-Type', 'Authorization'],
  })
);

// Parse JSON requests
app.use(express.json());

// Parse URL-encoded requests
app.use(express.urlencoded({ extended: true }));

// Compress responses
app.use(compression());
```

```
// Serve static files
app.use(express.static('public'));

// Routes
app.use('/api/users', require('./routes/users'));
app.use('/api/products', require('./routes/products'));

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

// Start server
app.listen(3000);
```

Routing

Basic Routing

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

// Basic route with path and callback function
app.get('/', (req, res) => {
  res.send('Home Page');
});

// Route with path parameter
app.get('/users/:id', (req, res) => {
  res.send(`User ID: ${req.params.id}`);
});

// Route with multiple path parameters
app.get('/users/:userId/posts/:postId', (req, res) => {
  res.send(`User: ${req.params.userId}, Post: ${req.params.postId}`);
});

// Route with query parameters
// URL: /search?q=javascript&limit=10
app.get('/search', (req, res) => {
  const query = req.query.q;
  const limit = req.query.limit;
  res.send(`Search for: ${query}, Limit: ${limit}`);
});

// Different HTTP methods
app.post('/users', (req, res) => {
  res.send('Create a new user');
});
```

```
app.put('/users/:id', (req, res) => {
  res.send(`Update user ${req.params.id}`);
});

app.delete('/users/:id', (req, res) => {
  res.send(`Delete user ${req.params.id}`);
});

// Route handler with multiple callbacks
app.get(
  '/complex',
  (req, res, next) => {
    console.log('First handler');
    next(); // Pass control to the next handler
  },
  (req, res) => {
    res.send('Complex route processed');
  }
);
```

Express Router

```
// routes/users.js
const express = require('express');
const router = express.Router();

// Define routes on the router
router.get('/', (req, res) => {
  res.send('All users');
});

router.get('/:id', (req, res) => {
  res.send(`User ${req.params.id}`);
});

router.post('/', (req, res) => {
  res.send('Create user');
});

module.exports = router;

// app.js
const express = require('express');
const app = express();
const userRoutes = require('./routes/users');

// Mount the router at a specific path
app.use('/api/users', userRoutes);

// All routes in userRoutes will be prefixed with '/api/users'
// e.g., GET /api/users/123
```

Route Grouping and Organization

```
// routes/index.js
const express = require('express');
const router = express.Router();
const userRoutes = require('./users');
const productRoutes = require('./products');
const orderRoutes = require('./orders');

// API routes
router.use('/users', userRoutes);
router.use('/products', productRoutes);
router.use('/orders', orderRoutes);

module.exports = router;

// app.js
const express = require('express');
const app = express();
const apiRoutes = require('./routes');

// All API routes are prefixed with '/api'
app.use('/api', apiRoutes);
```

Route-specific Middleware

```
// Middleware to check if user is authenticated
function isAuthenticated(req, res, next) {
  if (req.isAuthenticated()) {
    return next();
  }
  res.status(401).send('Unauthorized');
}

// Middleware to check if user is an admin
function isAdmin(req, res, next) {
  if (req.user && req.user.role === 'admin') {
    return next();
  }
  res.status(403).send('Forbidden');
}

// Apply middleware to specific routes
app.get('/profile', isAuthenticated, (req, res) => {
  res.send('User profile');
});

app.get('/admin', isAuthenticated, isAdmin, (req, res) => {
```

```
  res.send('Admin panel');
});

// Apply middleware to an entire router
const adminRouter = express.Router();
adminRouter.use(isAuthenticated, isAdmin);

adminRouter.get('/dashboard', (req, res) => {
  res.send('Admin dashboard');
});

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

app.use('/admin', adminRouter);
```

Advanced Routing Patterns

Route parameter preprocessing:

```
app.param('userId', (req, res, next, id) => {
  // Convert ID to number or fetch user from database
  req.user = { id: parseInt(id), name: 'User ' + id };
  next();
});

app.get('/users/:userId', (req, res) => {
  // req.user is already populated
  res.send(req.user);
});
```

Regular expression routes:

```
// Match paths like /file.html, /file.txt
app.get(/^\/file\.(html|txt)$/, (req, res) => {
  const fileType = req.path.split('.')[1];
  res.send(`Requested a ${fileType} file`);
});
```

Optional parameters:

```
// Route with optional parameter
// Both /users and /users/123 will match
app.get('/users/:id?', (req, res) => {
  if (req.params.id) {
    res.send(`User ${req.params.id}`);
```

```
  } else {
    res.send('All users');
  }
});
```

Route versioning:

```
// Version 1 API
const v1Router = express.Router();
v1Router.get('/users', (req, res) => {
  res.send('Users API v1');
});

// Version 2 API
const v2Router = express.Router();
v2Router.get('/users', (req, res) => {
  res.send('Users API v2');
});

app.use('/api/v1', v1Router);
app.use('/api/v2', v2Router);
```

Middleware

Middleware functions are functions that have access to the request object (`req`), the response object (`res`), and the next middleware function in the application's request-response cycle.

Types of Middleware

1. **Application-level middleware**

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

// Middleware without a path (applies to all requests)
app.use((req, res, next) => {
  console.log('Time:', Date.now());
  next();
});

// Middleware with a path (applies to specific routes)
app.use('/user/:id', (req, res, next) => {
  console.log('Request Type:', req.method);
  next();
});
```

2. **Router-level middleware**

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

router.use((req, res, next) => {
  console.log('Router Middleware');
  next();
});

router.get('/user/:id', (req, res, next) => {
  res.send('User Info');
});

app.use('/api', router);
```

3. **Error-handling middleware**

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

4. **Built-in middleware**

```
// Parse JSON bodies
app.use(express.json());

// Parse URL-encoded bodies
app.use(express.urlencoded({ extended: true }));

// Serve static files
app.use(express.static('public'));
```

5. **Third-party middleware**

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

app.use(morgan('dev')); // Request logging
app.use(helmet()); // Set security headers
app.use(compression()); // Compress responses
```

Creating Custom Middleware

```

```

```
// Custom authentication middleware
function authenticate(req, res, next) {
  const authHeader = req.headers.authorization;

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

  const token = authHeader.split(' ')[1];

  try {
    // Verify token (example using jsonwebtoken)
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded; // Attach user info to request
    next();
  } catch (err) {
    return res.status(401).json({ message: 'Invalid token' });
  }
}

// Middleware to log requests
function requestLogger(req, res, next) {
  console.log(`${new Date().toISOString()} - ${req.method} ${req.url}`);
  next();
}

// Middleware to handle 404 errors
function notFound(req, res, next) {
  const error = new Error(`Not Found - ${req.originalUrl}`);
  error.status = 404;
  next(error);
}

// Error middleware (must be defined last)
function errorHandler(err, req, res, next) {
  const statusCode = err.status || 500;
  res.status(statusCode);
  res.json({
    error: {
      message: err.message,
      status: statusCode,
      stack: process.env.NODE_ENV === 'production' ? '🥞' : err.stack,
    },
  });
}

// Usage
app.use(requestLogger);
app.get('/protected', authenticate, (req, res) => {
  res.json({ message: 'Protected data', user: req.user });
});
app.use(notFound);
app.use(errorHandler);
```

Middleware Execution Order

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

app.use((req, res, next) => {
  console.log('Second middleware');
  next();
});

app.get('/', (req, res, next) => {
  console.log('Route handler');
  res.send('Hello');
});

app.use((req, res, next) => {
  console.log('This middleware will not run for "/" route');
  next();
});

// Output for GET /
// First middleware
// Second middleware
// Route handler
```

Middleware Best Practices

1. **Keep middleware focused and small** Each middleware should do one thing well, following the single responsibility principle.
2. **Use next() appropriately** Always call `next()` unless you're ending the request-response cycle with `res.end()`, `res.send()`, etc.
3. **Handle errors properly** Pass errors to `next(err)` to be handled by error-handling middleware.
4. **Order matters** Middleware executes in the order it's defined, so place them accordingly.
5. **Use middleware composition** Combine multiple middleware functions into a single route handler when appropriate.

```
function validateUser(req, res, next) {
  // Validation logic
  next();
}

function checkPermissions(req, res, next) {
```

```
    // Permission check
    next();
  }

  app.post('/users', [validateUser, checkPermissions], (req, res) => {
    // Handler code
  });
```

Request/Response Handling

Request Object

The `req` object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, etc.

```
app.get('/example/:id', (req, res) => {
  // URL path parameters
  console.log(req.params.id);

  // Query string parameters (?name=John&age=30)
  console.log(req.query.name);
  console.log(req.query.age);

  // Request body (for POST, PUT, etc.)
  console.log(req.body); // Requires middleware like express.json()

  // HTTP headers
  console.log(req.headers['user-agent']);

  // Request method
  console.log(req.method); // GET, POST, etc.

  // Request URL
  console.log(req.url); // /example/123?name=John
  console.log(req.path); // /example/123

  // Cookies
  console.log(req.cookies); // Requires cookie-parser middleware

  // Session
  console.log(req.session); // Requires express-session middleware

  // IP address
  console.log(req.ip);

  // Is the request AJAX/XHR?
  console.log(req.xhr);

  // Protocol
  console.log(req.protocol); // http or https
```

```
  // Host
  console.log(req.hostname); // example.com

  // Secure connection?
  console.log(req.secure); // true if https

  res.send('Request details logged');
});
```

Response Object

The `res` object represents the HTTP response that an Express app sends when it receives an HTTP request.

```
app.get('/response-examples', (req, res) => {
  // Send a simple response
  // res.send('Plain text response');

  // Send JSON response
  // res.json({ message: 'JSON response', success: true });

  // Set status code
  // res.status(201).send('Created successfully');

  // Send a file
  // res.sendFile('/path/to/file.pdf');

  // Download a file
  // res.download('/path/to/file.pdf', 'report.pdf');

  // Render a template
  // res.render('index', { title: 'Home', message: 'Welcome' });

  // Redirect
  // res.redirect('/new-page');
  // res.redirect(301, '/permanent-new-page');

  // Set response headers
  // res.set('Content-Type', 'text/html');
  // res.set({
  //   'Content-Type': 'application/json',
  //   'X-Custom-Header': 'Custom Value'
  // });

  // Set a cookie
  // res.cookie('username', 'john', { maxAge: 900000, httpOnly: true });

  // Clear a cookie
  // res.clearCookie('username');

  // End the response without data
  // res.end();
```

```
  // Sending a response with chained methods
  res
    .status(200)
    .set('Content-Type', 'application/json')
    .cookie('visited', 'true', { maxAge: 60000 })
    .json({ message: 'Response examples' });
});
```

File Uploads

File uploads in Express are typically handled using the `multer` middleware:

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

// Configure storage
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'uploads/'); // Uploads directory
  },
  filename: (req, file, cb) => {
    // Create unique filename
    cb(null, `${Date.now()}-${file.originalname}`);
  },
});

// File filter
const fileFilter = (req, file, cb) => {
  // Accept images only
  if (!file.originalname.match(/\.(jpg|jpeg|png|gif)$/)) {
    return cb(new Error('Only image files are allowed!'), false);
  }
  cb(null, true);
};

// Configure multer
const upload = multer({
  storage: storage,
  limits: {
    fileSize: 1024 * 1024 * 5, // 5MB
  },
  fileFilter: fileFilter,
});

// Single file upload
app.post('/upload', upload.single('photo'), (req, res) => {
  if (!req.file) {
    return res.status(400).send('No file uploaded.');
```

```
  }

  res.json({
    message: 'File uploaded successfully',
    file: {
      name: req.file.originalname,
      type: req.file.mimetype,
      size: req.file.size,
      path: req.file.path,
    },
  });
});

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

  const fileDetails = req.files.map((file) => ({
    name: file.originalname,
    type: file.mimetype,
    size: file.size,
    path: file.path,
  }));

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

// Multiple fields with different names
const multiUpload = upload.fields([
  { name: 'avatar', maxCount: 1 },
  { name: 'gallery', maxCount: 5 },
]);

app.post('/upload-fields', multiUpload, (req, res) => {
  res.json({
    avatar: req.files.avatar ? req.files.avatar[0] : null,
    gallery: req.files.gallery || [],
  });
});

// Error handling for multer
app.use((err, req, res, next) => {
  if (err instanceof multer.MulterError) {
    // A Multer error occurred when uploading
    return res.status(400).json({
      error: true,
      message: err.message,
    });
  }
```

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

JSON and URL-encoded Data

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

app.post('/api/data', (req, res) => {
  // Access data from request body
  const { name, email, age } = req.body;

  // Validate data
  if (!name || !email) {
    return res.status(400).json({ error: 'Name and email are required' });
  }

  // Process data
  res.status(201).json({
    message: 'Data received successfully',
    user: { name, email, age },
  });
});
```

Content Negotiation

```
app.get('/api/users/:id', (req, res) => {
  const user = {
    id: req.params.id,
    name: 'John Doe',
    email: 'john@example.com',
  };

  // Respond with different formats based on Accept header
  res.format({
    'text/plain': () => {
      res.send(`User ${user.id}: ${user.name} (${user.email})`);
    },
    'text/html': () => {
      res.send(`
        <h1>User Details</h1>
        <p><strong>ID:</strong> ${user.id}</p>
        <p><strong>Name:</strong> ${user.name}</p>
        <p><strong>Email:</strong> ${user.email}</p>
      `);
    },
```

```
    'application/json': () => {
      res.json(user);
    },
    default: () => {
      // Default to JSON
      res.status(406).json({ error: 'Not Acceptable' });
    },
  });
});
```

Error Handling in Express

Global Error Handler

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

// Regular route handlers
app.get('/', (req, res) => {
  res.send('Home Page');
});

app.get('/error', (req, res, next) => {
  // Simulating an error
  const err = new Error('This is a simulated error');
  err.statusCode = 500;
  next(err); // Pass error to error handler
});

app.get('/async-error', async (req, res, next) => {
  try {
    // Simulating an async operation that fails
    await Promise.reject(new Error('Async operation failed'));
  } catch (err) {
    next(err); // Pass error to error handler
  }
});

// 404 handler - must be defined after all other routes
app.use((req, res, next) => {
  const error = new Error('Not Found');
  error.statusCode = 404;
  next(error);
});

// Global error handler - must be defined last
app.use((err, req, res, next) => {
  const statusCode = err.statusCode || 500;
  const message = err.message || 'Internal Server Error';

  // In development, include stack trace
```

```
  const error = {
    message,
    status: statusCode,
    stack: process.env.NODE_ENV === 'production' ? undefined : err.stack,
  };

  // Log error
  console.error(`[${new Date().toISOString()}] ${err.stack}`);

  // Send error response
  res.status(statusCode).json({ error });
});

app.listen(3000);
```

Async Error Handling

```
// Helper to catch async errors
const catchAsync = (fn) => {
  return (req, res, next) => {
    fn(req, res, next).catch(next);
  };
};

app.get(
  '/users/:id',
  catchAsync(async (req, res) => {
    const user = await User.findById(req.params.id);

    if (!user) {
      const error = new Error('User not found');
      error.statusCode = 404;
      throw error;
    }

    res.json(user);
  })
);

// Alternative using middleware
const asyncHandler = require('express-async-handler');

app.get(
  '/products/:id',
  asyncHandler(async (req, res) => {
    const product = await Product.findById(req.params.id);

    if (!product) {
      res.status(404);
      throw new Error('Product not found');
    }
```

```
    res.json(product);
  })
);
```

Custom Error Classes

```
class AppError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
    this.status = `${statusCode}`.startsWith('4') ? 'fail' : 'error';
    this.isOperational = true; // Marks error as operational (expected)

    Error.captureStackTrace(this, this.constructor);
  }
}

class NotFoundError extends AppError {
  constructor(message = 'Resource not found') {
    super(message, 404);
  }
}

class ValidationError extends AppError {
  constructor(message = 'Validation failed') {
    super(message, 400);
  }
}

class UnauthorizedError extends AppError {
  constructor(message = 'Unauthorized access') {
    super(message, 401);
  }
}

class ForbiddenError extends AppError {
  constructor(message = 'Forbidden') {
    super(message, 403);
  }
}

// Usage
app.get('/protected', (req, res, next) => {
  if (!req.isAuthenticated()) {
    return next(new UnauthorizedError('You must be logged in'));
  }

  if (!req.user.isAdmin) {
    return next(new ForbiddenError('Requires admin privileges'));
  }
```

```
  res.send('Protected content');
});

// Error handler
app.use((err, req, res, next) => {
  if (err instanceof AppError) {
    // Handle operational errors
    return res.status(err.statusCode).json({
      status: err.status,
      message: err.message,
    });
  }

  // Handle unexpected errors
  console.error('UNEXPECTED ERROR:', err);
  res.status(500).json({
    status: 'error',
    message: 'Something went wrong',
  });
});
```

Authentication and Authorization

Basic Authentication

```
// Basic authentication middleware
function basicAuth(req, res, next) {
  // Get authorization header
  const authHeader = req.headers.authorization;

  if (!authHeader || !authHeader.startsWith('Basic ')) {
    res.set('WWW-Authenticate', 'Basic realm="Authentication Required"');
    return res.status(401).send('Authentication required');
  }

  // Decode base64 credentials
  const base64Credentials = authHeader.split(' ')[1];
  const credentials = Buffer.from(base64Credentials, 'base64').toString('utf8');
  const [username, password] = credentials.split(':');

  // Verify credentials (replace with database lookup)
  if (username === 'admin' && password === 'password') {
    req.user = { username }; // Attach user to request
    return next();
  }

  res.set('WWW-Authenticate', 'Basic realm="Authentication Failed"');
  res.status(401).send('Invalid authentication credentials');
}
```

```
// Apply to specific routes
app.get('/api/protected', basicAuth, (req, res) => {
  res.json({ message: 'Protected data', user: req.user });
});

// Or apply to all routes in a router
const protectedRouter = express.Router();
protectedRouter.use(basicAuth);

protectedRouter.get('/data', (req, res) => {
  res.json({ message: 'Protected data' });
});

app.use('/protected', protectedRouter);
```

JWT Authentication

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

app.use(express.json());

// Secret key for JWT (use environment variable in production)
const JWT_SECRET = process.env.JWT_SECRET || 'your-secret-key';

// Sample user database (replace with real database)
const users = [
  { id: 1, username: 'user1', password: 'password1', role: 'user' },
  { id: 2, username: 'admin', password: 'admin123', role: 'admin' },
];

// Login route
app.post('/api/login', (req, res) => {
  const { username, password } = req.body;

  // Find user
  const user = users.find(
    (u) => u.username === username && u.password === password
  );

  if (!user) {
    return res.status(401).json({ message: 'Invalid credentials' });
  }

  // Generate JWT
  const token = jwt.sign(
    { userId: user.id, username: user.username, role: user.role },
    JWT_SECRET,
    { expiresIn: '1h' } // Token expires in 1 hour
  );
```

```
  res.json({
    message: 'Authentication successful',
    token,
  });
});

// JWT authentication middleware
function authenticateJWT(req, res, next) {
  const authHeader = req.headers.authorization;

  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    return res.status(401).json({ message: 'Authorization header required' });
  }

  const token = authHeader.split(' ')[1];

  try {
    const decoded = jwt.verify(token, JWT_SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    if (err.name === 'TokenExpiredError') {
      return res.status(401).json({ message: 'Token expired' });
    }
    return res.status(403).json({ message: 'Invalid token' });
  }
}

// Role-based authorization middleware
function authorizeRole(role) {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({ message: 'User not authenticated' });
    }

    if (req.user.role !== role) {
      return res.status(403).json({ message: 'Insufficient permissions' });
    }

    next();
  };
}

// Protected route with authentication
app.get('/api/profile', authenticateJWT, (req, res) => {
  res.json({
    message: 'Protected profile data',
    user: {
      id: req.user.userId,
      username: req.user.username,
      role: req.user.role,
    },
  });
```

```
});

// Route with role-based authorization
app.get('/api/admin', authenticateJWT, authorizeRole('admin'), (req, res) => {
  res.json({ message: 'Admin panel data' });
});

// Refresh token route
app.post('/api/refresh-token', authenticateJWT, (req, res) => {
  // Generate new token with fresh expiration
  const token = jwt.sign(
    {
      userId: req.user.userId,
      username: req.user.username,
      role: req.user.role,
    },
    JWT_SECRET,
    { expiresIn: '1h' }
  );

  res.json({ token });
});

app.listen(3000);
```

Session-based Authentication

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

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

// Configure session middleware
app.use(
  session({
    secret: process.env.SESSION_SECRET || 'secret-key',
    resave: false,
    saveUninitialized: false,
    cookie: {
      secure: process.env.NODE_ENV === 'production', // HTTPS in production
      httpOnly: true,
      maxAge: 24 * 60 * 60 * 1000, // 1 day
    },
  })
);

// Sample user database (replace with real database)
const users = [
```

```
  {
    id: 1,
    username: 'user1',
    password: '$2b$10$i8J7Lw3oKQy6Hz7vkJZF5OQRFBmFzk78b9SW.Y9C/eC3r5h29HfKO',
  }, // "password1"
  {
    id: 2,
    username: 'admin',
    password: '$2b$10$qPpXHFl8bCj2mK6BETzRYOz8OKmwWUEjQXGAZp4xc6xXc/MQHfoBu',
  }, // "admin123"
];

// Login route
app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  // Find user
  const user = users.find((u) => u.username === username);

  if (!user) {
    return res.status(401).json({ message: 'Invalid credentials' });
  }

  // Compare passwords
  const isPasswordValid = await bcrypt.compare(password, user.password);

  if (!isPasswordValid) {
    return res.status(401).json({ message: 'Invalid credentials' });
  }

  // Store user info in session
  req.session.user = {
    id: user.id,
    username: user.username,
  };

  res.json({ message: 'Login successful' });
});

// Logout route
app.post('/logout', (req, res) => {
  req.session.destroy((err) => {
    if (err) {
      return res.status(500).json({ message: 'Logout failed' });
    }
    res.clearCookie('connect.sid');
    res.json({ message: 'Logout successful' });
  });
});

// Authentication middleware
function isAuthenticated(req, res, next) {
  if (req.session.user) {
    return next();
```

```
  }
  res.status(401).json({ message: 'Unauthorized' });
}

// Protected route
app.get('/profile', isAuthenticated, (req, res) => {
  res.json({
    message: 'Profile data',
    user: req.session.user,
  });
});

app.listen(3000);
```

OAuth Authentication

```
const express = require('express');
const passport = require('passport');
const GoogleStrategy = require('passport-google-oauth20').Strategy;
const session = require('express-session');
const app = express();

// Configure session
app.use(
  session({
    secret: process.env.SESSION_SECRET || 'secret',
    resave: false,
    saveUninitialized: false,
  })
);

// Initialize Passport
app.use(passport.initialize());
app.use(passport.session());

// Configure Google Strategy
passport.use(
  new GoogleStrategy(
    {
      clientID: process.env.GOOGLE_CLIENT_ID,
      clientSecret: process.env.GOOGLE_CLIENT_SECRET,
      callbackURL: '/auth/google/callback',
    },
    (accessToken, refreshToken, profile, done) => {
      // In a real app, you would find or create a user in your database
      // For this example, we'll just use the profile
      return done(null, profile);
    }
  )
);
```

```
// Serialize/deserialize user
passport.serializeUser((user, done) => {
  done(null, user);
});

passport.deserializeUser((obj, done) => {
  done(null, obj);
});

// Route to start Google authentication
app.get(
  '/auth/google',
  passport.authenticate('google', {
    scope: ['profile', 'email'],
  })
);

// Google callback route
app.get(
  '/auth/google/callback',
  passport.authenticate('google', { failureRedirect: '/login' }),
  (req, res) => {
    // Successful authentication
    res.redirect('/profile');
  }
);

// Check if user is authenticated
function isAuthenticated(req, res, next) {
  if (req.isAuthenticated()) {
    return next();
  }
  res.redirect('/login');
}

// Protected route
app.get('/profile', isAuthenticated, (req, res) => {
  res.send(`
    <h1>Profile</h1>
    <p>ID: ${req.user.id}</p>
    <p>Name: ${req.user.displayName}</p>
    <p>Email: ${req.user.emails?.[0]?.value}</p>
    <a href="/logout">Logout</a>
  `);
});

// Logout route
app.get('/logout', (req, res) => {
  req.logout();
  res.redirect('/');
});

// Login page
app.get('/login', (req, res) => {
```

```
  res.send(`
    <h1>Login</h1>
    <a href="/auth/google">Login with Google</a>
  `);
});

// Home page
app.get('/', (req, res) => {
  res.send(`
    <h1>OAuth Example</h1>
    ${
      req.isAuthenticated()
        ? `<p>Logged in as ${req.user.displayName}</p><a
href="/profile">Profile</a> | <a href="/logout">Logout</a>`
        : `<a href="/login">Login</a>`
    }
  `);
});

app.listen(3000);
```

File Uploads and Static File Serving

Handling File Uploads with Multer

```
const express = require('express');
const multer = require('multer');
const path = require('path');
const fs = require('fs');
const app = express();

// Ensure uploads directory exists
const uploadDir = path.join(__dirname, 'uploads');
if (!fs.existsSync(uploadDir)) {
  fs.mkdirSync(uploadDir, { recursive: true });
}

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

// File filter
```

```
const fileFilter = (req, file, cb) => {
  // Accept images and PDFs
  const allowedFileTypes = /jpeg|jpg|png|gif|pdf/;
  const ext = path.extname(file.originalname).toLowerCase();
  const mimetype = allowedFileTypes.test(file.mimetype);
  const extname = allowedFileTypes.test(ext);

  if (mimetype && extname) {
    return cb(null, true);
  }

  cb(
    new Error(
      'Invalid file type. Only JPEG, PNG, GIF and PDF files are allowed.'
    )
  );
};

// Configure multer
const upload = multer({
  storage: storage,
  fileFilter: fileFilter,
  limits: {
    fileSize: 10 * 1024 * 1024, // 10MB
  },
});

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

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

// Multiple files 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 fileDetails = req.files.map((file) => ({
    originalname: file.originalname,
    filename: file.filename,
```

```
    path: file.path,
    size: file.size,
    mimetype: file.mimetype,
  }));

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

// Multiple fields with different file types
const profileUpload = upload.fields([
  { name: 'avatar', maxCount: 1 },
  { name: 'documents', maxCount: 3 },
]);

app.post('/upload/profile', profileUpload, (req, res) => {
  const response = {
    message: 'Files uploaded successfully',
    avatar: req.files.avatar ? req.files.avatar[0] : null,
    documents: req.files.documents || [],
  };

  res.json(response);
});

// Error handling for multer
app.use((err, req, res, next) => {
  if (err instanceof multer.MulterError) {
    // A Multer error occurred when uploading
    let message = 'File upload error';

    switch (err.code) {
      case 'LIMIT_FILE_SIZE':
        message = 'File is too large';
        break;
      case 'LIMIT_FILE_COUNT':
        message = 'Too many files uploaded';
        break;
      case 'LIMIT_UNEXPECTED_FILE':
        message = `Unexpected field: ${err.field}`;
        break;
      default:
        message = err.message;
    }

    return res.status(400).json({ error: message });
  }

  // Other errors
  res.status(500).json({ error: err.message });
});
```

```
app.listen(3000);
```

Memory Storage for File Uploads

```
// Using memory storage instead of disk storage
const memoryStorage = multer.memoryStorage();
const memUpload = multer({
  storage: memoryStorage,
  limits: { fileSize: 5 * 1024 * 1024 }, // 5MB
});

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

  // req.file.buffer contains the file data
  const fileSize = req.file.buffer.length;

  // Process the file in memory
  // For example, upload to cloud storage

  res.json({
    message: 'File processed',
    file: {
      originalname: req.file.originalname,
      size: fileSize,
      mimetype: req.file.mimetype,
    },
  });
});
```

Serving Static Files

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

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

// Serve static files from multiple directories
app.use(express.static('assets'));
app.use(express.static('uploads'));

// Serve static files with a URL prefix
app.use('/static', express.static('public'));
```

```
// Serve with absolute path
app.use('/files', express.static(path.join(__dirname, 'files')));

// Serve with options
app.use(
  express.static('public', {
    maxAge: '1d', // Cache for 1 day
    etag: true, // Generate ETag headers
    lastModified: true, // Set Last-Modified headers
    index: 'index.html', // Default file to serve
    dotfiles: 'ignore', // Ignore dotfiles
    extensions: ['html', 'htm'], // Try these extensions for extensionless URLs
  })
);

// Serving uploaded files
app.use('/uploads', express.static('uploads'));

// Restrict access to certain static files
app.use(
  '/protected',
  (req, res, next) => {
    if (req.isAuthenticated()) {
      return next();
    }
    res.status(403).send('Access denied');
  },
  express.static('protected-files')
);

app.listen(3000);
```

File Download

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

// Direct download with res.download()
app.get('/download/:filename', (req, res) => {
  const filename = req.params.filename;
  const filepath = path.join(__dirname, 'uploads', filename);

  // Check if file exists
  fs.access(filepath, fs.constants.F_OK, (err) => {
    if (err) {
      return res.status(404).send('File not found');
    }
```

```
    // Set content disposition header for download
    res.download(filepath, filename, (err) => {
      if (err) {
        // Handle error but keep in mind the response may be partially sent
        console.error('Download error:', err);
        if (!res.headersSent) {
          res.status(500).send('Download failed');
        }
      }
    });
  });
});

// Stream large files
app.get('/stream/:filename', (req, res) => {
  const filename = req.params.filename;
  const filepath = path.join(__dirname, 'uploads', filename);

  // Check if file exists
  fs.access(filepath, fs.constants.F_OK, (err) => {
    if (err) {
      return res.status(404).send('File not found');
    }

    // Get file stats
    fs.stat(filepath, (err, stats) => {
      if (err) {
        return res.status(500).send('Error retrieving file');
      }

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

      // Create read stream and pipe to response
      const fileStream = fs.createReadStream(filepath);
      fileStream.pipe(res);

      // Handle errors
      fileStream.on('error', (err) => {
        console.error('Stream error:', err);
        if (!res.headersSent) {
          res.status(500).send('Stream error');
        } else {
          res.end();
        }
      });
    });
  });
});
```

```
// Download with range support (for partial content/resumable downloads)
app.get('/range-download/:filename', (req, res) => {
  const filename = req.params.filename;
  const filepath = path.join(__dirname, 'uploads', filename);

  fs.stat(filepath, (err, stats) => {
    if (err) {
      if (err.code === 'ENOENT') {
        return res.status(404).send('File not found');
      }
      return res.status(500).send('Error retrieving file');
    }

    const fileSize = stats.size;
    const range = req.headers.range;

    if (range) {
      // Parse range header
      const parts = range.replace(/bytes=/, '').split('-');
      const start = parseInt(parts[0], 10);
      const end = parts[1] ? parseInt(parts[1], 10) : fileSize - 1;
      const chunkSize = end - start + 1;

      // Set headers for partial content
      res.writeHead(206, {
        'Content-Range': `bytes ${start}-${end}/${fileSize}`,
        'Accept-Ranges': 'bytes',
        'Content-Length': chunkSize,
        'Content-Type': 'application/octet-stream',
        'Content-Disposition': `attachment; filename="${filename}"`,
      });

      // Create stream for specified range
      const fileStream = fs.createReadStream(filepath, { start, end });
      fileStream.pipe(res);
    } else {
      // No range requested, send entire file
      res.writeHead(200, {
        'Content-Length': fileSize,
        'Content-Type': 'application/octet-stream',
        'Content-Disposition': `attachment; filename="${filename}"`,
      });

      fs.createReadStream(filepath).pipe(res);
    }
  });
});

app.listen(3000);
```

API Development Best Practices

RESTful API Design

```
const express = require('express');
const router = express.Router();

// GET /api/users - Get all users
router.get('/users', (req, res) => {
  // Pagination
  const page = parseInt(req.query.page) || 1;
  const limit = parseInt(req.query.limit) || 10;
  const skip = (page - 1) * limit;

  // Filtering
  const filter = {};
  if (req.query.role) {
    filter.role = req.query.role;
  }

  // Sorting
  const sort = {};
  if (req.query.sort) {
    const sortFields = req.query.sort.split(',');
    sortFields.forEach((field) => {
      if (field.startsWith('-')) {
        sort[field.substring(1)] = -1;
      } else {
        sort[field] = 1;
      }
    });
  }

  // Field selection
  const fields = req.query.fields ? req.query.fields.split(',').join(' ') : '';

  // Mock database call
  // In real app: const users = await
User.find(filter).skip(skip).limit(limit).sort(sort).select(fields);
  const users = [
    { id: 1, name: 'John Doe', email: 'john@example.com', role: 'user' },
    { id: 2, name: 'Jane Smith', email: 'jane@example.com', role: 'admin' },
  ];

  // Response with pagination info
  res.json({
    data: users,
    pagination: {
      total: 100, // Total count from database
      page,
      limit,
      pages: Math.ceil(100 / limit),
    },
  });
```

```
});

// GET /api/users/:id - Get user by ID
router.get('/users/:id', (req, res) => {
  const userId = req.params.id;

  // Mock database call
  // In real app: const user = await User.findById(userId);
  const user = {
    id: userId,
    name: 'John Doe',
    email: 'john@example.com',
    role: 'user',
  };

  if (!user) {
    return res.status(404).json({ error: 'User not found' });
  }

  res.json({
    data: user,
  });
});

// POST /api/users - Create new user
router.post('/users', (req, res) => {
  // Validate input
  const { name, email, password, role } = req.body;

  if (!name || !email || !password) {
    return res
      .status(400)
      .json({ error: 'Name, email and password are required' });
  }

  // Mock database call
  // In real app: const user = await User.create({ name, email, password, role });
  const user = { id: 3, name, email, role: role || 'user' };

  res.status(201).json({
    data: user,
  });
});

// PUT /api/users/:id - Update user (full update)
router.put('/users/:id', (req, res) => {
  const userId = req.params.id;
  const { name, email, role } = req.body;

  if (!name || !email) {
    return res.status(400).json({ error: 'Name and email are required' });
  }

  // Mock database call
```

```
  // In real app: const user = await User.findByIdAndUpdate(userId, { name, email,
role }, { new: true });
  const user = { id: userId, name, email, role };

  if (!user) {
    return res.status(404).json({ error: 'User not found' });
  }

  res.json({
    data: user,
  });
});

// PATCH /api/users/:id - Update user (partial update)
router.patch('/users/:id', (req, res) => {
  const userId = req.params.id;
  const updates = req.body;

  // Mock database call
  // In real app: const user = await User.findByIdAndUpdate(userId, updates, {
new: true });
  const user = { id: userId, name: 'Updated Name', ...updates };

  if (!user) {
    return res.status(404).json({ error: 'User not found' });
  }

  res.json({
    data: user,
  });
});

// DELETE /api/users/:id - Delete user
router.delete('/users/:id', (req, res) => {
  const userId = req.params.id;

  // Mock database call
  // In real app: const user = await User.findByIdAndDelete(userId);
  const user = { id: userId, name: 'John Doe' };

  if (!user) {
    return res.status(404).json({ error: 'User not found' });
  }

  res.status(204).send();
});

module.exports = router;
```

API Versioning

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

// Version 1 API
const v1Router = express.Router();

v1Router.get('/users', (req, res) => {
  res.json({
    version: 'v1',
    data: [
      { id: 1, name: 'John' },
      { id: 2, name: 'Jane' },
    ],
  });
});

// Version 2 API with additional fields
const v2Router = express.Router();

v2Router.get('/users', (req, res) => {
  res.json({
    version: 'v2',
    data: [
      { id: 1, name: 'John', email: 'john@example.com', role: 'user' },
      { id: 2, name: 'Jane', email: 'jane@example.com', role: 'admin' },
    ],
  });
});

// Mount routers at versioned paths
app.use('/api/v1', v1Router);
app.use('/api/v2', v2Router);

// Alternative versioning using Accept header
app.get('/api/users', (req, res) => {
  const version = req.headers['accept-version'] || '1.0.0';

  if (version.startsWith('1.')) {
    return res.json({
      version: '1.x',
      data: [
        { id: 1, name: 'John' },
        { id: 2, name: 'Jane' },
      ],
    });
  }

  if (version.startsWith('2.')) {
    return res.json({
      version: '2.x',
      data: [
        { id: 1, name: 'John', email: 'john@example.com' },
        { id: 2, name: 'Jane', email: 'jane@example.com' },
```

```
      ],
    });
  }

  res.status(400).json({ error: 'Unsupported API version' });
});

app.listen(3000);
```

API Documentation with Swagger

```
const express = require('express');
const swaggerJsDoc = require('swagger-jsdoc');
const swaggerUi = require('swagger-ui-express');
const app = express();

// Swagger configuration
const swaggerOptions = {
  swaggerDefinition: {
    openapi: '3.0.0',
    info: {
      title: 'User API',
      version: '1.0.0',
      description: 'User management API',
      contact: {
        name: 'API Support',
        email: 'support@example.com',
      },
    },
    servers: [
      {
        url: 'http://localhost:3000',
        description: 'Development server',
      },
    ],
  },
  apis: ['./routes/*.js'], // Path to the API routes files
};

const swaggerDocs = swaggerJsDoc(swaggerOptions);
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocs));

// Example route with Swagger annotations
/**
 * @swagger
 * /api/users:
 *   get:
 *     summary: Returns a list of users
 *     description: Retrieve a list of users from the database
 *     parameters:
 *       - in: query
```

```
 *         name: page
 *         schema:
 *           type: integer
 *           default: 1
 *         description: Page number
 *       - in: query
 *         name: limit
 *         schema:
 *           type: integer
 *           default: 10
 *         description: Number of items per page
 *     responses:
 *       200:
 *         description: A list of users
 *         content:
 *           application/json:
 *             schema:
 *               type: object
 *               properties:
 *                 data:
 *                   type: array
 *                   items:
 *                     type: object
 *                     properties:
 *                       id:
 *                         type: integer
 *                       name:
 *                         type: string
 *                       email:
 *                         type: string
 */
app.get('/api/users', (req, res) => {
  const users = [
    { id: 1, name: 'John Doe', email: 'john@example.com' },
    { id: 2, name: 'Jane Smith', email: 'jane@example.com' },
  ];

  res.json({ data: users });
});

/**
 * @swagger
 * /api/users/{id}:
 *   get:
 *     summary: Get a user by ID
 *     parameters:
 *       - in: path
 *         name: id
 *         required: true
 *         schema:
 *           type: integer
 *         description: User ID
 *     responses:
 *       200:
```

```
 *         description: User data
 *       404:
 *         description: User not found
 */
app.get('/api/users/:id', (req, res) => {
  const userId = parseInt(req.params.id);
  const user = { id: userId, name: 'John Doe', email: 'john@example.com' };

  if (!user) {
    return res.status(404).json({ error: 'User not found' });
  }

  res.json({ data: user });
});

app.listen(3000);
```

API Rate Limiting

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

// Basic rate limiter: max 100 requests per hour
const apiLimiter = rateLimit({
  windowMs: 60 * 60 * 1000, // 1 hour
  max: 100, // limit each IP to 100 requests per windowMs
  message: 'Too many requests from this IP, please try again after an hour',
  standardHeaders: true, // Return rate limit info in the `RateLimit-*` headers
  legacyHeaders: false, // Disable the `X-RateLimit-*` headers
});

// Apply to all API routes
app.use('/api/', apiLimiter);

// Different limits for different routes
const loginLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 5, // limit each IP to 5 login attempts per windowMs
  message: 'Too many login attempts, please try again after 15 minutes',
});

app.use('/api/login', loginLimiter);

// DDoS protection (require all requests to send a predefined token in header)
app.use('/api/sensitive', (req, res, next) => {
  const token = req.headers['api-security-token'];

  if (!token || token !== process.env.API_SECURITY_TOKEN) {
    return res.status(403).json({ error: 'Access denied' });
  }
```

```
  next();
});

app.listen(3000);
```

API Security Best Practices

1. **Use HTTPS**

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

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

// Create HTTPS server
https.createServer(options, app).listen(443, () => {
  console.log('HTTPS server running on port 443');
});

// Redirect HTTP to HTTPS
const http = require('http');
http
  .createServer((req, res) => {
    res.writeHead(301, {
      Location: `https://${req.headers.host}${req.url}`,
    });
    res.end();
  })
  .listen(80);
```

2. **Set security headers with Helmet**

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

// Use helmet to set various security headers
app.use(helmet());

// Or configure specific headers
app.use(
  helmet({
```

```
    contentSecurityPolicy: {
      directives: {
        defaultSrc: ["'self'"],
        scriptSrc: ["'self'", 'trusted-cdn.com'],
      },
    },
    xssFilter: true,
    noSniff: true,
    referrerPolicy: { policy: 'no-referrer' },
  })
);
```

3. **Validate and sanitize input**

```
const express = require('express');
const { body, validationResult } = require('express-validator');
const app = express();

app.use(express.json());

app.post(
  '/api/users',
  [
    // Validate input
    body('username')
      .isAlphanumeric()
      .withMessage('Username must be alphanumeric'),
    body('email').isEmail().withMessage('Must be a valid email'),
    body('password')
      .isLength({ min: 8 })
      .withMessage('Password must be at least 8 characters'),
  ],
  (req, res) => {
    // Check for validation errors
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    // Process valid input
    // ...

    res.status(201).json({ message: 'User created' });
  }
);
```

4. **API Authentication**

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

// API key authentication
function validateApiKey(req, res, next) {
  const apiKey = req.header('X-API-Key');

  if (!apiKey || !isValidApiKey(apiKey)) {
    return res.status(401).json({ error: 'Invalid API key' });
  }

  next();
}

// JWT authentication
function verifyToken(req, res, next) {
  const token = req.header('Authorization')?.replace('Bearer ', '');

  if (!token) {
    return res.status(401).json({ error: 'Access token required' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    res.status(403).json({ error: 'Invalid token' });
  }
}

// Protected routes
app.use('/api/public', validateApiKey);
app.use('/api/private', verifyToken);
```

5. **Prevent parameter pollution**

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

// Prevent parameter pollution
app.use(hpp());

// Or configure with whitelist
app.use(
  hpp({
    whitelist: ['filter', 'sort'], // Allow these parameters to be
duplicated
```

```
    })
  );
```

Testing Express Applications

Unit Testing with Jest

```
// user.controller.js
const User = require('../models/user.model');

exports.getUsers = async (req, res) => {
  try {
    const users = await User.find();
    res.status(200).json(users);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

exports.getUserById = async (req, res) => {
  try {
    const user = await User.findById(req.params.id);
    if (!user) {
      return res.status(404).json({ error: 'User not found' });
    }
    res.status(200).json(user);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

// user.controller.test.js
const { getUsers, getUserById } = require('../controllers/user.controller');
const User = require('../models/user.model');

// Mock the User model
jest.mock('../models/user.model');

describe('User Controller', () => {
  let req, res;

  beforeEach(() => {
    req = {
      params: {
        id: 'mock-user-id',
      },
    };

    res = {
      status: jest.fn().mockReturnThis(),
      json: jest.fn(),
```

```
  };
});

afterEach(() => {
  jest.clearAllMocks();
});

describe('getUsers', () => {
  test('should return all users', async () => {
    // Arrange
    const mockUsers = [{ name: 'User 1' }, { name: 'User 2' }];
    User.find.mockResolvedValue(mockUsers);

    // Act
    await getUsers(req, res);

    // Assert
    expect(User.find).toHaveBeenCalled();
    expect(res.status).toHaveBeenCalledWith(200);
    expect(res.json).toHaveBeenCalledWith(mockUsers);
  });

  test('should handle errors', async () => {
    // Arrange
    const errorMessage = 'Database error';
    User.find.mockRejectedValue(new Error(errorMessage));

    // Act
    await getUsers(req, res);

    // Assert
    expect(res.status).toHaveBeenCalledWith(500);
    expect(res.json).toHaveBeenCalledWith({ error: errorMessage });
  });
});

describe('getUserById', () => {
  test('should return a user if found', async () => {
    // Arrange
    const mockUser = { id: 'mock-user-id', name: 'User 1' };
    User.findById.mockResolvedValue(mockUser);

    // Act
    await getUserById(req, res);

    // Assert
    expect(User.findById).toHaveBeenCalledWith('mock-user-id');
    expect(res.status).toHaveBeenCalledWith(200);
    expect(res.json).toHaveBeenCalledWith(mockUser);
  });

  test('should return 404 if user not found', async () => {
    // Arrange
    User.findById.mockResolvedValue(null);
```

```
      // Act
      await getUserById(req, res);

      // Assert
      expect(res.status).toHaveBeenCalledWith(404);
      expect(res.json).toHaveBeenCalledWith({ error: 'User not found' });
    });
  });
});
```

Integration Testing with Supertest

```
// app.js
const express = require('express');
const userRoutes = require('./routes/user.routes');

const app = express();
app.use(express.json());
app.use('/api/users', userRoutes);

module.exports = app;

// server.js
const app = require('./app');
const PORT = process.env.PORT || 3000;

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});

// app.test.js
const request = require('supertest');
const mongoose = require('mongoose');
const app = require('../app');
const User = require('../models/user.model');

// Connect to test database before tests
beforeAll(async () => {
  const url = process.env.MONGO_TEST_URI || 'mongodb://localhost:27017/test_db';
  await mongoose.connect(url, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  });
});

// Clear database between tests
beforeEach(async () => {
  await User.deleteMany({});
});
```

```
// Disconnect after tests
afterAll(async () => {
  await mongoose.connection.close();
});

describe('User API', () => {
  describe('GET /api/users', () => {
    test('should return all users', async () => {
      // Arrange
      await User.create([
        { name: 'User 1', email: 'user1@example.com' },
        { name: 'User 2', email: 'user2@example.com' },
      ]);

      // Act
      const response = await request(app).get('/api/users');

      // Assert
      expect(response.status).toBe(200);
      expect(response.body.length).toBe(2);
      expect(response.body[0]).toHaveProperty('name', 'User 1');
      expect(response.body[1]).toHaveProperty('email', 'user2@example.com');
    });
  });

  describe('POST /api/users', () => {
    test('should create a new user', async () => {
      // Arrange
      const userData = {
        name: 'New User',
        email: 'newuser@example.com',
        password: 'password123',
      };

      // Act
      const response = await request(app)
        .post('/api/users')
        .send(userData)
        .set('Accept', 'application/json');

      // Assert
      expect(response.status).toBe(201);
      expect(response.body).toHaveProperty('name', userData.name);
      expect(response.body).toHaveProperty('email', userData.email);

      // Verify user was saved to database
      const savedUser = await User.findOne({ email: userData.email });
      expect(savedUser).not.toBeNull();
    });

    test('should return 400 for invalid data', async () => {
      // Arrange
      const invalidUserData = {
        name: 'Invalid User',
```

```
        // Missing email
        password: 'short', // Too short password
      };

      // Act
      const response = await request(app)
        .post('/api/users')
        .send(invalidUserData);

      // Assert
      expect(response.status).toBe(400);
      expect(response.body).toHaveProperty('errors');
    });
  });

  describe('GET /api/users/:id', () => {
    test('should return a user by ID', async () => {
      // Arrange
      const user = await User.create({
        name: 'Test User',
        email: 'testuser@example.com',
      });

      // Act
      const response = await request(app).get(`/api/users/${user._id}`);

      // Assert
      expect(response.status).toBe(200);
      expect(response.body).toHaveProperty('_id', user._id.toString());
      expect(response.body).toHaveProperty('name', 'Test User');
    });

    test('should return 404 for non-existent user', async () => {
      // Arrange
      const nonExistentId = mongoose.Types.ObjectId();

      // Act
      const response = await request(app).get(`/api/users/${nonExistentId}`);

      // Assert
      expect(response.status).toBe(404);
    });
  });
});
```

Testing Middleware

```
// auth.middleware.js
const jwt = require('jsonwebtoken');

function auth(req, res, next) {
```

```
  const token = req.header('x-auth-token');

  if (!token) {
    return res.status(401).json({ error: 'No token, authorization denied' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    res.status(401).json({ error: 'Token is not valid' });
  }
}

module.exports = auth;

// auth.middleware.test.js
const jwt = require('jsonwebtoken');
const auth = require('../middleware/auth.middleware');

// Mock jwt module
jest.mock('jsonwebtoken');

describe('Auth Middleware', () => {
  let req, res, next;

  beforeEach(() => {
    req = {
      header: jest.fn(),
    };

    res = {
      status: jest.fn().mockReturnThis(),
      json: jest.fn(),
    };

    next = jest.fn();
  });

  afterEach(() => {
    jest.clearAllMocks();
  });

  test('should call next() if token is valid', () => {
    // Arrange
    const token = 'valid-token';
    const decodedToken = { userId: '123', role: 'user' };

    req.header.mockReturnValue(token);
    jwt.verify.mockReturnValue(decodedToken);

    // Act
    auth(req, res, next);
```

```
    // Assert
    expect(req.header).toHaveBeenCalledWith('x-auth-token');
    expect(jwt.verify).toHaveBeenCalledWith(token, process.env.JWT_SECRET);
    expect(req.user).toEqual(decodedToken);
    expect(next).toHaveBeenCalled();
  });

  test('should return 401 if no token is provided', () => {
    // Arrange
    req.header.mockReturnValue(null);

    // Act
    auth(req, res, next);

    // Assert
    expect(res.status).toHaveBeenCalledWith(401);
    expect(res.json).toHaveBeenCalledWith({
      error: 'No token, authorization denied',
    });
    expect(next).not.toHaveBeenCalled();
  });

  test('should return 401 if token is invalid', () => {
    // Arrange
    const token = 'invalid-token';

    req.header.mockReturnValue(token);
    jwt.verify.mockImplementation(() => {
      throw new Error('Token is not valid');
    });

    // Act
    auth(req, res, next);

    // Assert
    expect(res.status).toHaveBeenCalledWith(401);
    expect(res.json).toHaveBeenCalledWith({ error: 'Token is not valid' });
    expect(next).not.toHaveBeenCalled();
  });
});
```

Code Coverage with Jest

```
// package.json
{
  "scripts": {
    "test": "jest",
    "test:coverage": "jest --coverage"
  },
  "jest": {
```

```
    "testEnvironment": "node",
    "coverageDirectory": "coverage",
    "collectCoverageFrom": [
      "**/*.js",
      "!**/node_modules/**",
      "!**/coverage/**",
      "!**/tests/**",
      "!**/config/**"
    ],
    "coverageThreshold": {
      "global": {
        "branches": 80,
        "functions": 80,
        "lines": 80,
        "statements": 80
      }
    }
  }
}
```

Testing Routes with Authentication

```
// routes/post.routes.js
const express = require('express');
const router = express.Router();
const auth = require('../middleware/auth');
const postController = require('../controllers/post.controller');

router.get('/', postController.getPosts);
router.get('/:id', postController.getPostById);
router.post('/', auth, postController.createPost);
router.put('/:id', auth, postController.updatePost);
router.delete('/:id', auth, postController.deletePost);

module.exports = router;

// routes/post.routes.test.js
const request = require('supertest');
const mongoose = require('mongoose');
const jwt = require('jsonwebtoken');
const app = require('../app');
const Post = require('../models/post.model');
const User = require('../models/user.model');

// Mock auth middleware
jest.mock('../middleware/auth', () => {
  return (req, res, next) => {
    if (req.headers['x-auth-token']) {
      req.user = { id: req.headers['user-id'] || 'default-user-id' };
      return next();
    }
```

```
      return res.status(401).json({ error: 'No token provided' });
    };
  });

  beforeAll(async () => {
    await mongoose.connect(process.env.MONGO_TEST_URI);
  });

  afterAll(async () => {
    await mongoose.connection.close();
  });

  beforeEach(async () => {
    await Post.deleteMany({});
    await User.deleteMany({});
  });

  describe('Post Routes', () => {
    let token, userId;

    beforeEach(async () => {
      // Create a test user
      const user = await User.create({
        name: 'Test User',
        email: 'test@example.com',
        password: 'password123',
      });

      userId = user._id;
      token = 'test-token'; // Mock token
    });

    describe('Protected routes', () => {
      test('should create a post when authenticated', async () => {
        // Arrange
        const postData = {
          title: 'Test Post',
          content: 'This is a test post',
        };

        // Act
        const response = await request(app)
          .post('/api/posts')
          .set('x-auth-token', token)
          .set('user-id', userId.toString())
          .send(postData);

        // Assert
        expect(response.status).toBe(201);
        expect(response.body).toHaveProperty('title', postData.title);
        expect(response.body).toHaveProperty('author', userId.toString());
      });

      test('should return 401 when not authenticated', async () => {
```

```
      // Arrange
      const postData = {
        title: 'Test Post',
        content: 'This is a test post',
      };

      // Act (no token provided)
      const response = await request(app).post('/api/posts').send(postData);

      // Assert
      expect(response.status).toBe(401);
    });
  });

  describe('Public routes', () => {
    test('should get all posts without authentication', async () => {
      // Arrange
      await Post.create([
        { title: 'Post 1', content: 'Content 1', author: userId },
        { title: 'Post 2', content: 'Content 2', author: userId },
      ]);

      // Act
      const response = await request(app).get('/api/posts');

      // Assert
      expect(response.status).toBe(200);
      expect(response.body.length).toBe(2);
    });
  });
});
```

MongoDB and Mongoose

MongoDB Concepts and Data Modeling

What is MongoDB?

MongoDB is a NoSQL document database that stores data in flexible, JSON-like documents. Unlike traditional relational databases with tables, rows, and columns, MongoDB uses collections and documents.

- **Document**: A record in MongoDB, similar to a row in a relational database, but can contain nested structures
- **Collection**: A group of documents, similar to a table in a relational database
- **Database**: A container for collections

MongoDB Document Structure

```
// Sample MongoDB document
{
```

```
  _id: ObjectId("5f8d3b9e5e8e7c4b1c6d2f7a"),
  name: "John Doe",
  email: "john@example.com",
  age: 30,
  address: {
    street: "123 Main St",
    city: "New York",
    state: "NY",
    zip: "10001"
  },
  hobbies: ["reading", "swimming", "cycling"],
  createdAt: ISODate("2020-10-19T12:34:56Z")
}
```

Key features of MongoDB documents:

1. **Dynamic schema**: Documents in the same collection can have different fields
2. **Nested structures**: Documents can contain arrays and nested objects
3. **No joins**: Data that is accessed together is typically stored together (embedding)

Data Modeling in MongoDB

MongoDB provides two main ways to structure relationships:

1. **Embedding** (Denormalization): Storing related data in the same document
2. **Referencing** (Normalization): Storing references (typically `_id`) to documents in other collections

Embedded Data Model Example:

```
// User document with embedded addresses
{
  _id: ObjectId("5f8d3b9e5e8e7c4b1c6d2f7a"),
  name: "John Doe",
  email: "john@example.com",
  addresses: [
    {
      type: "home",
      street: "123 Main St",
      city: "New York",
      state: "NY",
      zip: "10001"
    },
    {
      type: "work",
      street: "456 Corporate Ave",
      city: "New York",
      state: "NY",
      zip: "10002"
    }
  ]
}
```

Referenced Data Model Example:

```
// User document
{
  _id: ObjectId("5f8d3b9e5e8e7c4b1c6d2f7a"),
  name: "John Doe",
  email: "john@example.com",
  addresses: [
    ObjectId("5f8d3c1e5e8e7c4b1c6d2f7b"),
    ObjectId("5f8d3c1e5e8e7c4b1c6d2f7c")
  ]
}

// Address documents in a separate collection
{
  _id: ObjectId("5f8d3c1e5e8e7c4b1c6d2f7b"),
  user_id: ObjectId("5f8d3b9e5e8e7c4b1c6d2f7a"),
  type: "home",
  street: "123 Main St",
  city: "New York",
  state: "NY",
  zip: "10001"
}

{
  _id: ObjectId("5f8d3c1e5e8e7c4b1c6d2f7c"),
  user_id: ObjectId("5f8d3b9e5e8e7c4b1c6d2f7a"),
  type: "work",
  street: "456 Corporate Ave",
  city: "New York",
  state: "NY",
  zip: "10002"
}
```

When to Embed vs. Reference

Embed when:

- There's a one-to-one relationship
- There's a one-to-many relationship where the "many" objects always appear with the parent
- The embedded data is small and doesn't grow significantly
- The data is queried together most of the time

Reference when:

- There's a many-to-many relationship
- The embedded data would grow too large (MongoDB has a 16MB document size limit)
- The child data is frequently accessed on its own
- You need to model complex relationships

- You need to update the child data frequently without updating the parent

Common Data Modeling Patterns

1. **One-to-One**: Embed directly in the document

```
{
  _id: ObjectId("..."),
  name: "John Doe",
  profile: {
    bio: "Software Developer",
    avatar: "john.jpg",
    socialMedia: {
      twitter: "@johndoe",
      linkedin: "johndoe"
    }
  }
}
```

2. **One-to-Few**: Embed an array of documents

```
{
  _id: ObjectId("..."),
  name: "John Doe",
  phone_numbers: [
    { type: "home", number: "212-555-1234" },
    { type: "work", number: "646-555-5678" }
  ]
}
```

3. **One-to-Many**: Use references

```
// Author
{
  _id: ObjectId("author1"),
  name: "Jane Austen"
}

// Books
{
  _id: ObjectId("book1"),
  title: "Pride and Prejudice",
  author_id: ObjectId("author1")
}
{
  _id: ObjectId("book2"),
  title: "Sense and Sensibility",
```

```
  author_id: ObjectId("author1")
}
```

4. **Many-to-Many**: Use arrays of references on both sides

```
// Student
{
  _id: ObjectId("student1"),
  name: "Alice",
  course_ids: [ObjectId("course1"), ObjectId("course2")]
}

// Course
{
  _id: ObjectId("course1"),
  name: "Mathematics",
  student_ids: [ObjectId("student1"), ObjectId("student2")]
}
```

5. **Tree Structure**: Use parent references or path arrays

```
// Parent reference approach
{
  _id: ObjectId("category1"),
  name: "Electronics",
  parent_id: null
}
{
  _id: ObjectId("category2"),
  name: "Computers",
  parent_id: ObjectId("category1")
}

// Path array approach
{
  _id: ObjectId("category1"),
  name: "Electronics",
  path: []
}
{
  _id: ObjectId("category2"),
  name: "Computers",
  path: [ObjectId("category1")]
}
```

Setting up MongoDB

Local Installation

Installing MongoDB Community Edition (Ubuntu):

```
# Import MongoDB public GPG key
wget -qO - https://www.mongodb.org/static/pgp/server-6.0.asc | sudo apt-key add -

# Create list file for MongoDB
echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/6.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-6.0.list

# Reload local package database
sudo apt-get update

# Install MongoDB packages
sudo apt-get install -y mongodb-org

# Start MongoDB service
sudo systemctl start mongod

# Enable MongoDB to start on boot
sudo systemctl enable mongod

# Verify installation
mongod --version
```

Installing MongoDB Community Edition (macOS with Homebrew):

```
# Install MongoDB
brew tap mongodb/brew
brew install mongodb-community

# Start MongoDB service
brew services start mongodb-community
```

Installing MongoDB Community Edition (Windows):

1. Download the MongoDB Community Server MSI installer from the MongoDB website
2. Run the installer and follow the instructions
3. MongoDB is installed as a Windows service by default

MongoDB Atlas Setup

MongoDB Atlas is a cloud database service that provides a free tier for small projects.

1. **Create an account and cluster**:
    - Go to MongoDB Atlas
    - Sign up or log in
    - Create a new project

- Build a new cluster (free tier is sufficient for learning)
- Choose a cloud provider and region

2. **Configure network access**:
   - Go to the "Network Access" tab
   - Click "Add IP Address"
   - For development, you can add your current IP or allow access from anywhere (0.0.0.0/0)

3. **Create a database user**:
   - Go to the "Database Access" tab
   - Click "Add New Database User"
   - Enter username and password
   - Assign appropriate privileges (e.g., "Read and Write to Any Database")

4. **Connect to your cluster**:
   - Go to the "Clusters" tab
   - Click "Connect"
   - Choose "Connect your application"
   - Select Node.js as the driver
   - Copy the connection string

Connecting to MongoDB from Node.js

Using the MongoDB Native Driver:

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

// Connection URI
const uri = 'mongodb://localhost:27017/mydatabase'; // Local MongoDB
// OR
// const uri = 'mongodb+srv://<username>:
<password>@cluster0.mongodb.net/mydatabase?retryWrites=true&w=majority'; //
MongoDB Atlas

// Create a new MongoClient
const client = new MongoClient(uri, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

async function connectToDatabase() {
  try {
    // Connect to the MongoDB server
    await client.connect();
    console.log('Connected to MongoDB');

    // Get the database and collection
    const database = client.db('mydatabase');
```

```
    const collection = database.collection('users');

    // Perform operations
    // ...

    return { database, collection };
  } catch (err) {
    console.error('Error connecting to MongoDB:', err);
    throw err;
  }
}

// Usage
async function main() {
  try {
    const { collection } = await connectToDatabase();

    // Perform database operations
    const result = await collection.insertOne({
      name: 'John Doe',
      email: 'john@example.com',
    });

    console.log('Inserted document:', result.insertedId);

    const users = await collection.find({}).toArray();
    console.log('Users:', users);
  } catch (err) {
    console.error('Error:', err);
  } finally {
    // Close the connection when done
    await client.close();
    console.log('Disconnected from MongoDB');
  }
}

main();
```

CRUD Operations Using MongoDB Native Driver

Create (Insert) Operations

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://localhost:27017/mydatabase';
const client = new MongoClient(uri);

async function create() {
  try {
    await client.connect();
    const database = client.db('mydatabase');
    const collection = database.collection('users');
```

```
    // Insert a single document
    const singleResult = await collection.insertOne({
      name: 'John Doe',
      email: 'john@example.com',
      age: 30,
      createdAt: new Date(),
    });

    console.log(`Document inserted with ID: ${singleResult.insertedId}`);

    // Insert multiple documents
    const multipleResult = await collection.insertMany([
      {
        name: 'Jane Smith',
        email: 'jane@example.com',
        age: 25,
        createdAt: new Date(),
      },
      {
        name: 'Bob Johnson',
        email: 'bob@example.com',
        age: 35,
        createdAt: new Date(),
      },
    ]);

    console.log(`${multipleResult.insertedCount} documents inserted`);
    console.log(multipleResult.insertedIds);

    // Insert with ordered option (stops on first error)
    await collection.insertMany(
      [
        { _id: 1, name: 'Alice' },
        { _id: 2, name: 'Bob' },
      ],
      { ordered: true }
    );

    // Insert with unordered option (continues after errors)
    try {
      await collection.insertMany(
        [
          { _id: 1, name: 'Alice' }, // This will fail if _id 1 already exists
          { _id: 3, name: 'Charlie' }, // This will still be inserted
        ],
        { ordered: false }
      );
    } catch (err) {
      console.log('Some inserts failed, but others succeeded');
    }
  } finally {
    await client.close();
  }
```

```
}

create();
```

Read (Query) Operations

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://localhost:27017/mydatabase';
const client = new MongoClient(uri);

async function read() {
  try {
    await client.connect();
    const database = client.db('mydatabase');
    const collection = database.collection('users');

    // Find a single document
    const singleUser = await collection.findOne({ email: 'john@example.com' });
    console.log('Single user:', singleUser);

    // Find multiple documents
    const users = await collection.find({ age: { $gt: 25 } }).toArray();
    console.log('Users over 25:', users);

    // Count documents
    const count = await collection.countDocuments({ age: { $gt: 25 } });
    console.log('Count of users over 25:', count);

    // Find with projection (include only specified fields)
    const projection = await collection
      .find(
        { age: { $gt: 25 } },
        { projection: { name: 1, email: 1, _id: 0 } } // Include name and email,
exclude _id
      )
      .toArray();
    console.log('Projection result:', projection);

    // Find with sorting
    const sorted = await collection
      .find({})
      .sort({ age: -1 }) // Descending order by age
      .toArray();
    console.log('Sorted by age (descending):', sorted);

    // Find with limit and skip (pagination)
    const page = 1;
    const pageSize = 2;
    const skip = (page - 1) * pageSize;

    const paginatedResults = await collection
```

```
      .find({})
      .skip(skip)
      .limit(pageSize)
      .toArray();
    console.log(`Page ${page} results:`, paginatedResults);

    // Complex query with multiple conditions
    const complexQuery = await collection
      .find({
        $and: [
          { age: { $gte: 25 } },
          { age: { $lte: 40 } },
          {
            $or: [
              { name: /^J/ }, // Names starting with J
              { email: /example\.com$/ }, // Emails ending with example.com
            ],
          },
        ],
      })
      .toArray();
    console.log('Complex query results:', complexQuery);
  } finally {
    await client.close();
  }
}

read();
```

Update Operations

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://localhost:27017/mydatabase';
const client = new MongoClient(uri);

async function update() {
  try {
    await client.connect();
    const database = client.db('mydatabase');
    const collection = database.collection('users');

    // Update a single document
    const updateOneResult = await collection.updateOne(
      { email: 'john@example.com' }, // Filter
      { $set: { age: 31, updatedAt: new Date() } } // Update
    );

    console.log(`Matched ${updateOneResult.matchedCount} document(s)`);
    console.log(`Modified ${updateOneResult.modifiedCount} document(s)`);

    // Update multiple documents
```

```
const updateManyResult = await collection.updateMany(
  { age: { $lt: 30 } }, // Filter: age less than 30
  { $inc: { age: 1 } } // Increment age by 1
);

console.log(`Updated ${updateManyResult.modifiedCount} documents`);

// Replace a document
const replaceResult = await collection.replaceOne(
  { email: 'bob@example.com' },
  {
    name: 'Robert Johnson',
    email: 'robert@example.com',
    age: 36,
    createdAt: new Date(),
  }
);

console.log(`Replaced ${replaceResult.modifiedCount} document`);

// Upsert (update if exists, insert if not)
const upsertResult = await collection.updateOne(
  { email: 'alice@example.com' },
  {
    $set: {
      name: 'Alice Wilson',
      age: 28,
      updatedAt: new Date(),
    },
    $setOnInsert: {
      createdAt: new Date(),
    },
  },
  { upsert: true }
);

if (upsertResult.upsertedCount > 0) {
  console.log(
    `Inserted a new document with ID: ${upsertResult.upsertedId}`
  );
} else {
  console.log(`Updated an existing document`);
}

// Array updates
await collection.updateOne(
  { email: 'john@example.com' },
  {
    $push: { hobbies: 'cooking' }, // Add to array
  }
);

await collection.updateOne(
  { email: 'john@example.com' },
```

```
      {
        $addToSet: { hobbies: 'reading' }, // Add to array if not exists
      }
    );

    await collection.updateOne(
      { email: 'john@example.com' },
      {
        $pull: { hobbies: 'cooking' }, // Remove from array
      }
    );
  } finally {
    await client.close();
  }
}

update();
```

Delete Operations

```
const { MongoClient } = require('mongodb');
const uri = 'mongodb://localhost:27017/mydatabase';
const client = new MongoClient(uri);

async function remove() {
  try {
    await client.connect();
    const database = client.db('mydatabase');
    const collection = database.collection('users');

    // Delete a single document
    const deleteOneResult = await collection.deleteOne({
      email: 'jane@example.com',
    });
    console.log(`Deleted ${deleteOneResult.deletedCount} document`);

    // Delete multiple documents
    const deleteManyResult = await collection.deleteMany({ age: { $lt: 25 } });
    console.log(`Deleted ${deleteManyResult.deletedCount} documents`);

    // Delete all documents (use with caution!)
    // const deleteAllResult = await collection.deleteMany({});
    // console.log(`Deleted ${deleteAllResult.deletedCount} documents`);

    // Find one document and delete it
    const findAndDeleteResult = await collection.findOneAndDelete({
      email: 'alice@example.com',
    });
    console.log('Deleted document:', findAndDeleteResult.value);
  } finally {
    await client.close();
```

```
  }
}

remove();
```

Mongoose ODM (Object Data Modeling)

What is Mongoose?

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a higher-level, schema-based API over the native MongoDB driver, making it easier to:

- Define schemas for your data
- Validate data before saving
- Define relationships between data
- Create models that map to MongoDB collections
- Perform middleware functions (pre/post hooks)

Installing and Connecting with Mongoose

```
npm install mongoose
```

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

// Connection URI
const uri = 'mongodb://localhost:27017/mydatabase';
// OR for MongoDB Atlas
// const uri = 'mongodb+srv://<username>:
<password>@cluster0.mongodb.net/mydatabase?retryWrites=true&w=majority';

// Connect to MongoDB
mongoose
  .connect(uri, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
  })
  .then(() => console.log('Connected to MongoDB'))
  .catch((err) => console.error('Error connecting to MongoDB:', err));

// Connection events
mongoose.connection.on('connected', () => {
  console.log('Mongoose connected to database');
});

mongoose.connection.on('error', (err) => {
  console.error('Mongoose connection error:', err);
});
```

```
mongoose.connection.on('disconnected', () => {
  console.log('Mongoose disconnected');
});

// Close the Mongoose connection when Node process ends
process.on('SIGINT', async () => {
  await mongoose.connection.close();
  console.log('Mongoose connection closed through app termination');
  process.exit(0);
});
```

Defining Schemas and Models

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

// Define a schema
const userSchema = new Schema(
  {
    name: {
      type: String,
      required: [true, 'Name is required'],
      trim: true,
      minlength: [2, 'Name must be at least 2 characters'],
      maxlength: [50, 'Name cannot exceed 50 characters'],
    },
    email: {
      type: String,
      required: [true, 'Email is required'],
      unique: true,
      lowercase: true,
      trim: true,
      match: [/^\S+@\S+\.\S+$/, 'Please use a valid email address'],
    },
    password: {
      type: String,
      required: [true, 'Password is required'],
      minlength: [8, 'Password must be at least 8 characters'],
    },
    age: {
      type: Number,
      min: [18, 'Must be at least 18 years old'],
      max: [120, 'Age cannot exceed 120'],
    },
    role: {
      type: String,
      enum: {
        values: ['user', 'admin', 'editor'],
        message: '{VALUE} is not a valid role',
      },
      default: 'user',
```

```
    },
    isActive: {
      type: Boolean,
      default: true,
    },
    address: {
      street: String,
      city: String,
      state: String,
      zip: String,
      country: {
        type: String,
        default: 'USA',
      },
    },
    hobbies: [String],
    loginAttempts: {
      type: Number,
      default: 0,
    },
    meta: {
      type: Map,
      of: String,
    },
  },
  {
    timestamps: true, // Adds createdAt and updatedAt timestamps
    collection: 'users', // Optional: specify collection name (defaults to plural
of model name)
  }
);

// Add a virtual property (not stored in MongoDB)
userSchema.virtual('fullName').get(function () {
  return `${this.name.first} ${this.name.last}`;
});

// Static method (model method)
userSchema.statics.findByEmail = function (email) {
  return this.findOne({ email });
};

// Instance method
userSchema.methods.incrementLoginAttempts = function () {
  this.loginAttempts += 1;
  return this.save();
};

// Query helper
userSchema.query.activeOnly = function () {
  return this.where({ isActive: true });
};

// Middleware (pre hook)
```

```
userSchema.pre('save', function (next) {
  // Only hash the password if it's modified or new
  if (!this.isModified('password')) return next();

  // Hash password logic would go here
  // this.password = hashedPassword;
  console.log('Pre-save hook: Password would be hashed here');
  next();
});

// Middleware (post hook)
userSchema.post('save', function (doc, next) {
  console.log(`User ${doc._id} has been saved`);
  next();
});

// Create model from schema
const User = mongoose.model('User', userSchema);

module.exports = User;
```

Validation in Mongoose

```
const productSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, 'Product name is required'],
    trim: true,
  },
  price: {
    type: Number,
    required: [true, 'Price is required'],
    min: [0, 'Price cannot be negative'],
  },
  description: String,
  category: {
    type: String,
    enum: ['electronics', 'clothing', 'food', 'books', 'other'],
  },
  inStock: {
    type: Boolean,
    default: true,
  },
  quantity: {
    type: Number,
    validate: {
      validator: function (value) {
        return value >= 0;
      },
      message: 'Quantity cannot be negative',
    },
```

```
  },
  tags: {
    type: [String],
    validate: {
      validator: function (arr) {
        return arr && arr.length > 0;
      },
      message: 'At least one tag is required',
    },
  },
  sku: {
    type: String,
    validate: {
      validator: async function (value) {
        // Custom async validator
        const product = await this.constructor.findOne({ sku: value });
        // If this is a new product or the SKU hasn't changed, it's valid
        return !product || product._id.equals(this._id);
      },
      message: 'SKU must be unique',
    },
  },
  dimensions: {
    width: Number,
    height: Number,
    depth: Number,
  },
  reviews: [
    {
      user: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'User',
      },
      rating: {
        type: Number,
        required: true,
        min: 1,
        max: 5,
      },
      comment: String,
      date: {
        type: Date,
        default: Date.now,
      },
    },
  ],
});

// Custom schema method for validation
productSchema.methods.validateDimensions = function () {
  if (this.dimensions) {
    const { width, height, depth } = this.dimensions;
    return width > 0 && height > 0 && depth > 0;
  }
```

```
  return true;
};

const Product = mongoose.model('Product', productSchema);

// Using validation
async function createProduct() {
  try {
    const product = new Product({
      name: 'Smartphone',
      price: 599.99,
      category: 'electronics',
      quantity: 50,
      tags: ['tech', 'mobile'],
      sku: 'PHONE-12345',
      dimensions: {
        width: 7,
        height: 14,
        depth: 0.8,
      },
    });

    // Validate before saving
    const isValid = await product.validate();
    console.log('Validation passed');

    // Save if validation passed
    await product.save();
    console.log('Product saved:', product);
  } catch (err) {
    if (err.name === 'ValidationError') {
      // Extract and format validation errors
      const errors = Object.values(err.errors).map((e) => e.message);
      console.error('Validation errors:', errors);
    } else {
      console.error('Error saving product:', err);
    }
  }
}

createProduct();
```

CRUD Operations with Mongoose

```
// Creating documents
async function createUser() {
  try {
    // Create a single document
    const user = new User({
      name: 'John Doe',
      email: 'john@example.com',
```

```
      password: 'password123',
      age: 30,
      role: 'user',
      address: {
        street: '123 Main St',
        city: 'New York',
        state: 'NY',
        zip: '10001',
      },
      hobbies: ['reading', 'hiking'],
    });

    await user.save();
    console.log('User created:', user);

    // Alternative: Create using model
    const anotherUser = await User.create({
      name: 'Jane Smith',
      email: 'jane@example.com',
      password: 'password456',
      age: 25,
      role: 'editor',
    });
    console.log('Another user created:', anotherUser);

    // Create multiple documents
    const multipleUsers = await User.insertMany([
      {
        name: 'Bob Johnson',
        email: 'bob@example.com',
        password: 'password789',
        age: 35,
      },
      {
        name: 'Alice Williams',
        email: 'alice@example.com',
        password: 'passwordabc',
        age: 28,
      },
    ]);
    console.log(`${multipleUsers.length} users created`);
  } catch (err) {
    console.error('Error creating user:', err);
  }
}

// Reading documents
async function findUsers() {
  try {
    // Find a single document
    const user = await User.findById('60af896f56d6e52a8877bcf3');
    console.log('User by ID:', user);

    // Find one document by criteria
```

```
    const adminUser = await User.findOne({ role: 'admin' });
    console.log('Admin user:', adminUser);

    // Find multiple documents
    const youngUsers = await User.find({ age: { $lt: 30 } });
    console.log(`Found ${youngUsers.length} young users`);

    // Find with field selection
    const userNames = await User.find({}, 'name email -_id');
    console.log('User names and emails:', userNames);

    // Find with sorting
    const sortedUsers = await User.find()
      .sort({ age: -1, name: 1 }) // Sort by age (desc) and then name (asc)
      .exec();
    console.log('Sorted users:', sortedUsers);

    // Find with pagination
    const page = 1;
    const limit = 10;
    const skip = (page - 1) * limit;

    const paginatedUsers = await User.find().skip(skip).limit(limit).exec();
    console.log(`Page ${page} users:`, paginatedUsers);

    // Find with population (joins)
    const userWithPosts = await User.findById('60af896f56d6e52a8877bcf3')
      .populate('posts') // Assuming user has posts field referencing Post model
      .exec();
    console.log('User with posts:', userWithPosts);

    // Count documents
    const count = await User.countDocuments({ isActive: true });
    console.log(`${count} active users`);

    // Using query helpers
    const activeUsers = await User.find().activeOnly().exec();
    console.log(`${activeUsers.length} active users found with query helper`);

    // Using static methods
    const userByEmail = await User.findByEmail('john@example.com');
    console.log('User by email:', userByEmail);
  } catch (err) {
    console.error('Error finding users:', err);
  }
}

// Updating documents
async function updateUsers() {
  try {
    // Update a single document
    const updatedUser = await User.findByIdAndUpdate(
      '60af896f56d6e52a8877bcf3',
      {
```

```
        age: 31,
        'address.city': 'Boston',
      },
      { new: true } // Return the updated document
    );
    console.log('Updated user:', updatedUser);

    // Update multiple documents
    const result = await User.updateMany(
      { age: { $lt: 30 } },
      { $inc: { age: 1 } }
    );
    console.log(`Updated ${result.nModified} users`);

    // Update with validation
    const user = await User.findById('60af896f56d6e52a8877bcf3');
    user.email = 'john.doe@example.com';
    user.age = 32;
    await user.save(); // Runs validation
    console.log('User updated with validation:', user);

    // Update arrays
    await User.findByIdAndUpdate('60af896f56d6e52a8877bcf3', {
      $push: { hobbies: 'swimming' },
    });

    // Find one and update
    const replacedUser = await User.findOneAndReplace(
      { email: 'alice@example.com' },
      {
        name: 'Alice Brown',
        email: 'alice.brown@example.com',
        password: 'newpassword',
        age: 29,
        role: 'user',
      },
      { new: true }
    );
    console.log('Replaced user:', replacedUser);
  } catch (err) {
    console.error('Error updating users:', err);
  }
}

// Deleting documents
async function deleteUsers() {
  try {
    // Delete a single document
    const deletedUser = await User.findByIdAndDelete(
      '60af896f56d6e52a8877bcf3'
    );
    console.log('Deleted user:', deletedUser);

    // Delete many documents
```

```
    const result = await User.deleteMany({ isActive: false });
    console.log(`Deleted ${result.deletedCount} inactive users`);

    // Find one and delete
    const foundAndDeleted = await User.findOneAndDelete({
      email: 'john@example.com',
    });
    console.log('Found and deleted:', foundAndDeleted);
  } catch (err) {
    console.error('Error deleting users:', err);
  }
}
```

Relationships in Mongoose

Modeling Relationships by Reference (Normalized)

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

// Author schema
const authorSchema = new Schema(
  {
    name: {
      type: String,
      required: true,
    },
    bio: String,
    website: String,
  },
  { timestamps: true }
);

const Author = mongoose.model('Author', authorSchema);

// Book schema (references Author)
const bookSchema = new Schema(
  {
    title: {
      type: String,
      required: true,
    },
    summary: String,
    isbn: {
      type: String,
      unique: true,
    },
    author: {
      type: Schema.Types.ObjectId,
      ref: 'Author',
      required: true,
```

```
    },
    categories: [
      {
        type: Schema.Types.ObjectId,
        ref: 'Category',
      },
    ],
  },
  { timestamps: true }
);

const Book = mongoose.model('Book', bookSchema);

// Category schema (many-to-many with books)
const categorySchema = new Schema({
  name: {
    type: String,
    required: true,
    unique: true,
  },
  description: String,
  books: [
    {
      type: Schema.Types.ObjectId,
      ref: 'Book',
    },
  ],
});

const Category = mongoose.model('Category', categorySchema);

// Create an author and a book
async function createAuthorAndBook() {
  try {
    // Create author
    const author = await Author.create({
      name: 'J.K. Rowling',
      bio: 'British author best known for the Harry Potter series',
    });

    // Create book referencing the author
    const book = await Book.create({
      title: "Harry Potter and the Philosopher's Stone",
      summary: 'The first Harry Potter book',
      isbn: '9780747532743',
      author: author._id,
    });

    console.log('Author and book created');

    // Create categories
    const fantasy = await Category.create({
      name: 'Fantasy',
      description: 'Fantasy literature',
```

```
    });

    const youngAdult = await Category.create({
      name: 'Young Adult',
      description: 'Literature for young adults',
    });

    // Add categories to book
    book.categories.push(fantasy._id, youngAdult._id);
    await book.save();

    // Add book to categories
    fantasy.books.push(book._id);
    youngAdult.books.push(book._id);

    await fantasy.save();
    await youngAdult.save();

    console.log('Categories added to book and vice versa');
  } catch (err) {
    console.error('Error:', err);
  }
}

// Find books with author information
async function findBooksWithAuthors() {
  try {
    const books = await Book.find()
      .populate('author')
      .populate('categories')
      .exec();

    books.forEach((book) => {
      console.log(`Title: ${book.title}`);
      console.log(`Author: ${book.author.name}`);
      console.log('Categories:', book.categories.map((c) => c.name).join(', '));
      console.log('---');
    });
  } catch (err) {
    console.error('Error:', err);
  }
}

// Find authors with their books
async function findAuthorsWithBooks() {
  try {
    const authors = await Author.find();

    for (const author of authors) {
      const books = await Book.find({ author: author._id });
      console.log(`Author: ${author.name}`);
      console.log('Books:', books.map((b) => b.title).join(', '));
      console.log('---');
    }
```

```
    } catch (err) {
      console.error('Error:', err);
    }
  }

  // Find categories with their books
  async function findCategoriesWithBooks() {
    try {
      const categories = await Category.find().populate('books').exec();

      categories.forEach((category) => {
        console.log(`Category: ${category.name}`);
        console.log('Books:', category.books.map((b) => b.title).join(', '));
        console.log('---');
      });
    } catch (err) {
      console.error('Error:', err);
    }
  }
```

Modeling Relationships by Embedding (Denormalized)

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

// Comment subdocument schema (embedded in Post)
const commentSchema = new Schema({
  user: {
    type: String,
    required: true,
  },
  text: {
    type: String,
    required: true,
  },
  createdAt: {
    type: Date,
    default: Date.now,
  },
});

// Post schema with embedded comments
const postSchema = new Schema(
  {
    title: {
      type: String,
      required: true,
    },
    content: {
      type: String,
      required: true,
    },
```

```
    author: {
      name: String,
      email: String,
    },
    tags: [String],
    comments: [commentSchema],
    likes: {
      type: Number,
      default: 0,
    },
  },
  { timestamps: true }
);

const Post = mongoose.model('Post', postSchema);

// Create a post with embedded comments
async function createPostWithComments() {
  try {
    const post = await Post.create({
      title: 'Introduction to MongoDB',
      content: 'MongoDB is a NoSQL database...',
      author: {
        name: 'John Doe',
        email: 'john@example.com',
      },
      tags: ['mongodb', 'nosql', 'database'],
      comments: [
        {
          user: 'Jane Smith',
          text: 'Great introduction!',
        },
        {
          user: 'Bob Johnson',
          text: 'Thanks for sharing this information.',
        },
      ],
    });

    console.log('Post created with comments:', post);
  } catch (err) {
    console.error('Error:', err);
  }
}

// Add a comment to an existing post
async function addCommentToPost() {
  try {
    const post = await Post.findOne({ title: 'Introduction to MongoDB' });

    if (!post) {
      console.log('Post not found');
      return;
    }
```

```
    post.comments.push({
      user: 'Alice Williams',
      text: 'I learned a lot from this post.',
    });

    await post.save();
    console.log('Comment added to post');
  } catch (err) {
    console.error('Error:', err);
  }
}

// Find posts with comment filtering
async function findPostsWithCommentFiltering() {
  try {
    // Find posts that have comments from a specific user
    const posts = await Post.find({
      'comments.user': 'Jane Smith',
    });

    console.log(`Found ${posts.length} posts with comments by Jane Smith`);

    // Extract and display specific comments
    posts.forEach((post) => {
      const janeComments = post.comments.filter(
        (comment) => comment.user === 'Jane Smith'
      );

      console.log(`Post: ${post.title}`);
      console.log('Comments by Jane Smith:');
      janeComments.forEach((comment) => {
        console.log(`- ${comment.text}`);
      });
      console.log('---');
    });
  } catch (err) {
    console.error('Error:', err);
  }
}

// Update a specific comment
async function updateComment() {
  try {
    const post = await Post.findOne({ title: 'Introduction to MongoDB' });

    if (!post || post.comments.length === 0) {
      console.log('Post or comments not found');
      return;
    }

    // Find the comment to update (first one in this example)
    const commentId = post.comments[0]._id;
```

```
    // Update using $ positional operator
    await Post.updateOne(
      { _id: post._id, 'comments._id': commentId },
      { $set: { 'comments.$.text': 'Updated comment text!' } }
    );

    console.log('Comment updated');
  } catch (err) {
    console.error('Error:', err);
  }
}

// Remove a comment
async function removeComment() {
  try {
    const post = await Post.findOne({ title: 'Introduction to MongoDB' });

    if (!post || post.comments.length === 0) {
      console.log('Post or comments not found');
      return;
    }

    // Find the comment to remove (last one in this example)
    const commentId = post.comments[post.comments.length - 1]._id;

    // Remove using $pull operator
    await Post.updateOne(
      { _id: post._id },
      { $pull: { comments: { _id: commentId } } }
    );

    console.log('Comment removed');
  } catch (err) {
    console.error('Error:', err);
  }
}
```

Mongoose Middleware (Hooks)

```
const mongoose = require('mongoose');
const bcrypt = require('bcrypt');
const Schema = mongoose.Schema;

const userSchema = new Schema(
  {
    username: {
      type: String,
      required: true,
      unique: true,
      trim: true,
    },
```

```
    email: {
      type: String,
      required: true,
      unique: true,
      lowercase: true,
    },
    password: {
      type: String,
      required: true,
      minlength: 8,
    },
    isActive: {
      type: Boolean,
      default: true,
    },
    lastLogin: Date,
    loginHistory: [
      {
        date: Date,
        ipAddress: String,
      },
    ],
  },
  { timestamps: true }
);

// Document middleware

// Pre-save hook (before saving a document)
userSchema.pre('save', async function (next) {
  // Only hash the password if it's modified or new
  if (!this.isModified('password')) return next();

  try {
    // Generate a salt and hash the password
    const salt = await bcrypt.genSalt(10);
    this.password = await bcrypt.hash(this.password, salt);
    next();
  } catch (err) {
    next(err);
  }
});

// Post-save hook (after saving a document)
userSchema.post('save', function (doc, next) {
  console.log(`User ${doc.username} saved successfully`);
  next();
});

// Pre-remove hook (before removing a document)
userSchema.pre('remove', function (next) {
  console.log(`User ${this.username} is about to be removed`);
  // Clean up related data in other collections if needed
  // Example: remove user's posts, comments, etc.
```

```
  next();
});

// Pre-validate hook (before validation)
userSchema.pre('validate', function (next) {
  console.log(`Validating user ${this.username}`);

  // Set default values or transform data if needed
  if (this.username) {
    this.username = this.username.trim();
  }

  next();
});

// Query middleware

// Pre-find hook (before executing a find query)
userSchema.pre('find', function () {
  // 'this' refers to the query, not the document
  console.log('Executing find query');

  // Add a default condition to exclude inactive users
  this.where({ isActive: true });
});

// Post-find hook (after executing a find query)
userSchema.post('find', function (docs) {
  console.log(`Found ${docs.length} documents`);
});

// Pre-findOne hook (before executing a findOne query)
userSchema.pre('findOne', function () {
  console.log('Executing findOne query');

  // Add default condition
  this.where({ isActive: true });
});

// Apply to multiple operations
userSchema.pre(['updateOne', 'updateMany'], function () {
  console.log(`Executing ${this.op} operation`);

  // Add updatedAt timestamp
  this.updateOne({}, { $set: { updatedAt: new Date() } });
});

// Aggregate middleware
userSchema.pre('aggregate', function () {
  console.log('Executing aggregate pipeline');

  // Add a match stage to the beginning of the pipeline
  this.pipeline().unshift({ $match: { isActive: true } });
});
```

```
// Add instance method to compare passwords
userSchema.methods.comparePassword = async function (candidatePassword) {
  return bcrypt.compare(candidatePassword, this.password);
};

// Add instance method to record login
userSchema.methods.recordLogin = function (ipAddress) {
  this.lastLogin = new Date();
  this.loginHistory.push({
    date: new Date(),
    ipAddress: ipAddress || 'unknown',
  });
  return this.save();
};

const User = mongoose.model('User', userSchema);

// Usage example
async function createAndLoginUser() {
  try {
    // Create user (password will be hashed by pre-save hook)
    const user = await User.create({
      username: 'testuser',
      email: 'test@example.com',
      password: 'password123',
    });

    console.log('User created:', user);

    // Test login
    const isMatch = await user.comparePassword('password123');
    console.log('Password match:', isMatch);

    if (isMatch) {
      // Record login
      await user.recordLogin('127.0.0.1');
      console.log('Login recorded');
    }

    // Find users (pre-find hook will add isActive: true filter)
    const users = await User.find();
    console.log(`Found ${users.length} active users`);
  } catch (err) {
    console.error('Error:', err);
  }
}

createAndLoginUser();
```

MongoDB Indexing and Performance Optimization

Creating and Managing Indexes

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

// Product schema with various indexes
const productSchema = new Schema({
  name: {
    type: String,
    required: true,
  },
  sku: {
    type: String,
    required: true,
    unique: true, // Creates a unique index
  },
  price: {
    type: Number,
    required: true,
    index: true, // Creates a simple index
  },
  category: {
    type: String,
    index: true, // Creates a simple index
  },
  tags: {
    type: [String],
    index: true, // Creates an index on the array
  },
  manufacturer: {
    name: String,
    country: String,
  },
  createdAt: {
    type: Date,
    default: Date.now,
  },
});

// Compound index
productSchema.index({ category: 1, price: -1 }); // 1 for ascending, -1 for
descending

// Compound index with options
productSchema.index(
  { manufacturer: 1, 'manufacturer.country': 1 },
  { name: 'manufacturer_index' } // Custom name for the index
);

// Text index for full-text search
productSchema.index(
  { name: 'text', description: 'text' },
```

```
  {
    weights: {
      name: 10, // Higher weight for name matches
      description: 5,
    },
    name: 'product_text_index',
  }
);

// TTL (Time-To-Live) index
productSchema.index(
  { createdAt: 1 },
  { expireAfterSeconds: 30 * 24 * 60 * 60 } // Delete after 30 days
);

// Partial index
productSchema.index(
  { price: 1 },
  {
    partialFilterExpression: { price: { $gt: 100 } }, // Only index products with
price > 100
    name: 'expensive_products',
  }
);

// Create model
const Product = mongoose.model('Product', productSchema);

// Create indexes
async function createIndexes() {
  try {
    await Product.createIndexes();
    console.log('Indexes created');

    // List all indexes
    const indexes = await Product.collection.indexes();
    console.log('Indexes:', indexes);
  } catch (err) {
    console.error('Error creating indexes:', err);
  }
}

// Drop an index
async function dropIndex() {
  try {
    await Product.collection.dropIndex('expensive_products');
    console.log('Index dropped');
  } catch (err) {
    console.error('Error dropping index:', err);
  }
}

// Drop all indexes (except _id)
async function dropAllIndexes() {
```

```
  try {
    await Product.collection.dropIndexes();
    console.log('All indexes dropped');
  } catch (err) {
    console.error('Error dropping indexes:', err);
  }
}
```

Query Performance and Optimization

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

const orderSchema = new Schema({
  orderNumber: {
    type: String,
    required: true,
    unique: true,
    index: true,
  },
  customer: {
    type: Schema.Types.ObjectId,
    ref: 'Customer',
    required: true,
    index: true,
  },
  products: [
    {
      product: {
        type: Schema.Types.ObjectId,
        ref: 'Product',
      },
      quantity: Number,
      price: Number,
    },
  ],
  totalAmount: {
    type: Number,
    required: true,
    index: true,
  },
  status: {
    type: String,
    enum: ['pending', 'processing', 'shipped', 'delivered', 'cancelled'],
    default: 'pending',
    index: true,
  },
  shippingAddress: {
    street: String,
    city: String,
    state: String,
```

```
    zip: String,
    country: String,
  },
  createdAt: {
    type: Date,
    default: Date.now,
    index: true,
  },
});

// Compound indexes for common queries
orderSchema.index({ customer: 1, createdAt: -1 });
orderSchema.index({ status: 1, createdAt: -1 });

const Order = mongoose.model('Order', orderSchema);

// Using explain() to analyze query performance
async function analyzeQuery() {
  try {
    // Simple find query with explain
    const explanation = await Order.find({ status: 'processing' })
      .sort({ createdAt: -1 })
      .limit(10)
      .explain('executionStats');

    console.log('Query explanation:');
    console.log(
      'Execution time (ms):',
      explanation.executionStats.executionTimeMillis
    );
    console.log(
      'Documents examined:',
      explanation.executionStats.totalDocsExamined
    );
    console.log('Documents returned:', explanation.executionStats.nReturned);
    console.log('Query plan:', explanation.queryPlanner.winningPlan);

    // Check if we're using an index
    if (explanation.queryPlanner.winningPlan.inputStage.stage === 'IXSCAN') {
      console.log(
        'Using index:',
        explanation.queryPlanner.winningPlan.inputStage.indexName
      );
    } else {
      console.log('Not using an index (COLLSCAN)');
    }
  } catch (err) {
    console.error('Error analyzing query:', err);
  }
}

// Optimized vs non-optimized queries
async function compareQueries() {
  try {
```

```
    console.time('Unoptimized query');
    // Not using projection (fetches all fields)
    const unoptimizedResult = await Order.find({ status: 'shipped' }).sort({
      createdAt: -1,
    });
    console.timeEnd('Unoptimized query');
    console.log(
      `Unoptimized query returned ${unoptimizedResult.length} documents`
    );

    console.time('Optimized query');
    // Using projection (fetches only needed fields)
    const optimizedResult = await Order.find({ status: 'shipped' })
      .select('orderNumber totalAmount createdAt')
      .sort({ createdAt: -1 });
    console.timeEnd('Optimized query');
    console.log(`Optimized query returned ${optimizedResult.length} documents`);

    // Comparison with lean() - returns plain JavaScript objects
    console.time('With lean()');
    const leanResult = await Order.find({ status: 'shipped' })
      .select('orderNumber totalAmount createdAt')
      .sort({ createdAt: -1 })
      .lean();
    console.timeEnd('With lean()');
    console.log(`Lean query returned ${leanResult.length} documents`);
  } catch (err) {
    console.error('Error comparing queries:', err);
  }
}

// Optimization for large result sets: pagination vs cursors
async function handleLargeResults() {
  try {
    // Pagination (skip/limit)
    console.time('Pagination');
    const page = 2;
    const limit = 100;
    const skip = (page - 1) * limit;

    const paginatedResults = await Order.find()
      .skip(skip)
      .limit(limit)
      .select('orderNumber totalAmount')
      .lean();
    console.timeEnd('Pagination');
    console.log(
      `Retrieved ${paginatedResults.length} documents (page ${page})`
    );

    // Cursor-based pagination (more efficient for large datasets)
    console.time('Cursor-based');
    // Get the last _id from the previous page
    const lastId = '6148b59c9c7142001c123456'; // Example ID from page 1
```

```
    const cursorResults = await Order.find({ _id: { $gt: lastId } })
      .limit(limit)
      .select('orderNumber totalAmount')
      .sort({ _id: 1 })
      .lean();
    console.timeEnd('Cursor-based');
    console.log(
      `Retrieved ${cursorResults.length} documents (after ID ${lastId})`
    );
  } catch (err) {
    console.error('Error handling large results:', err);
  }
}

// Aggregation performance optimization
async function optimizedAggregation() {
  try {
    // Basic aggregation (slower)
    console.time('Basic aggregation');
    const basicResults = await Order.aggregate([
      { $match: { status: 'delivered' } },
      {
        $group: {
          _id: '$customer',
          orderCount: { $sum: 1 },
          totalSpent: { $sum: '$totalAmount' },
        },
      },
      { $sort: { totalSpent: -1 } },
      { $limit: 10 },
    ]);
    console.timeEnd('Basic aggregation');
    console.log(`Basic aggregation returned ${basicResults.length} results`);

    // Optimized aggregation (using indexes)
    console.time('Optimized aggregation');
    const optimizedResults = await Order.aggregate([
      { $match: { status: 'delivered' } }, // Uses index on status
      {
        $project: {
          customer: 1,
          totalAmount: 1,
        },
      },
      {
        $group: {
          _id: '$customer',
          orderCount: { $sum: 1 },
          totalSpent: { $sum: '$totalAmount' },
        },
      },
      { $sort: { totalSpent: -1 } },
      { $limit: 10 },
```

```
    ]).option({ allowDiskUse: true }); // For large datasets
    console.timeEnd('Optimized aggregation');
    console.log(
      `Optimized aggregation returned ${optimizedResults.length} results`
    );
  } catch (err) {
    console.error('Error in aggregation:', err);
  }
}
```

Indexing Best Practices

1. **Only Index What You Need**
   - Each index takes up space and affects write performance
   - Create indexes based on your query patterns
   - Monitor and remove unused indexes
2. **Understand Your Query Patterns**
   - Analyze your most common and performance-critical queries
   - Create compound indexes for queries that filter on multiple fields
   - Ensure the order of fields in compound indexes matches your query patterns
3. **Consider Index Size**
   - Smaller indexes fit better in memory and improve performance
   - Avoid indexing large fields (like text descriptions or arrays with many elements)
   - Use partial indexes for filtered queries
4. **Balance Read and Write Performance**
   - Indexes speed up reads but slow down writes
   - For write-heavy collections, minimize the number of indexes
   - For read-heavy collections, optimize indexes for query patterns
5. **Use Covered Queries When Possible**
   - A covered query is satisfied entirely by index keys (no need to fetch documents)
   - Include all fields needed in the result in your index
   - Use projection to return only needed fields
6. **Optimize for Sort Operations**
   - Sorting without an index is expensive (in-memory sort or disk-based sort)
   - Create indexes that support both your filter conditions and sort fields
   - The order of fields in the index should match your most common sort order
7. **Monitor Index Usage**
   - Use explain() to see which indexes are used by your queries

- Check for COLLSCAN operations, which indicate full collection scans
- Review the executionStats to find slow queries

```
// Example of implementing these best practices
const userSchema = new Schema({
  username: String,
  email: String,
  active: Boolean,
  lastLogin: Date,
  role: String,
  profile: {
    firstName: String,
    lastName: String,
    age: Number,
    location: String,
  },
});

// Good practice 1: Only index fields used for filtering
userSchema.index({ username: 1 });
userSchema.index({ email: 1 });

// Good practice 2: Create compound indexes for common query patterns
userSchema.index({ role: 1, active: 1 });

// Good practice 3: Use partial indexes for filtered queries
userSchema.index(
  { lastLogin: 1 },
  { partialFilterExpression: { active: true } }
);

// Good practice 4: Create indexes for sorting
userSchema.index({ 'profile.location': 1, lastLogin: -1 });

// Bad practice: Too many indexes
// userSchema.index({ 'profile.firstName': 1 }); // Rarely used in queries
// userSchema.index({ 'profile.lastName': 1 });  // Rarely used in queries
// userSchema.index({ 'profile.age': 1 });       // Rarely used in queries

// Test index effectiveness
async function testIndexEffectiveness() {
  try {
    // Query that should use the compound index
    const result1 = await User.find({
      role: 'admin',
      active: true,
    }).explain('executionStats');

    console.log(
      'Index used:',
      result1.queryPlanner.winningPlan.inputStage.indexName
    );
```

```
    console.log(
      'Execution time (ms):',
      result1.executionStats.executionTimeMillis
    );

    // Query that should use the partial index
    const result2 = await User.find({
      active: true,
      lastLogin: { $gte: new Date('2023-01-01') },
    }).explain('executionStats');

    console.log(
      'Index used:',
      result2.queryPlanner.winningPlan.inputStage.indexName
    );
    console.log(
      'Execution time (ms):',
      result2.executionStats.executionTimeMillis
    );

    // Example of a covered query
    const result3 = await User.find(
      { role: 'admin', active: true },
      { role: 1, active: 1, _id: 0 } // Only return fields in the index
    ).explain('executionStats');

    console.log(
      'Is covered query:',
      result3.queryPlanner.winningPlan.coveredQuery ? 'Yes' : 'No'
    );
    console.log(
      'Execution time (ms):',
      result3.executionStats.executionTimeMillis
    );
  } catch (err) {
    console.error('Error testing indexes:', err);
  }
}
```

MongoDB Aggregation Framework

Basic Aggregation Pipeline

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

// Sample schema for sales data
const saleSchema = new Schema({
  product: String,
  quantity: Number,
  price: Number,
```

```
  date: Date,
  customer: {
    name: String,
    location: String,
    type: String, // 'retail' or 'wholesale'
  },
  store: {
    id: String,
    location: String,
    region: String,
  },
});

const Sale = mongoose.model('Sale', saleSchema);

// Basic aggregation: Counting documents
async function countDocuments() {
  try {
    const result = await Sale.aggregate([{ $count: 'totalSales' }]);

    console.log('Total sales:', result[0].totalSales);
  } catch (err) {
    console.error('Error counting documents:', err);
  }
}

// Basic aggregation: Filtering with $match
async function filterSales() {
  try {
    const result = await Sale.aggregate([
      { $match: { 'customer.type': 'retail' } },
      { $count: 'retailSales' },
    ]);

    console.log('Retail sales:', result[0].retailSales);
  } catch (err) {
    console.error('Error filtering sales:', err);
  }
}

// Basic aggregation: Group and sum
async function groupAndSum() {
  try {
    const result = await Sale.aggregate([
      {
        $group: {
          _id: '$product',
          totalSold: { $sum: '$quantity' },
          totalRevenue: { $sum: { $multiply: ['$price', '$quantity'] } },
          averagePrice: { $avg: '$price' },
          minPrice: { $min: '$price' },
          maxPrice: { $max: '$price' },
          count: { $sum: 1 },
        },
```

```
      },
    ]);

    console.log('Sales by product:', result);
  } catch (err) {
    console.error('Error grouping sales:', err);
  }
}

// Basic aggregation: Sort results
async function sortResults() {
  try {
    const result = await Sale.aggregate([
      {
        $group: {
          _id: '$product',
          totalRevenue: { $sum: { $multiply: ['$price', '$quantity'] } },
        },
      },
      { $sort: { totalRevenue: -1 } },
      { $limit: 5 },
    ]);

    console.log('Top 5 products by revenue:', result);
  } catch (err) {
    console.error('Error sorting results:', err);
  }
}

// Basic aggregation: Project fields
async function projectFields() {
  try {
    const result = await Sale.aggregate([
      { $match: { 'store.region': 'West' } },
      {
        $project: {
          product: 1,
          revenue: { $multiply: ['$price', '$quantity'] },
          month: { $month: '$date' },
          year: { $year: '$date' },
          customerType: '$customer.type',
          _id: 0,
        },
      },
    ]);

    console.log('Projected sales data:', result.slice(0, 5)); // Show first 5
results
  } catch (err) {
    console.error('Error projecting fields:', err);
  }
}
```

Advanced Aggregation Operations

```
// Advanced aggregation: Multiple stages
async function multiStageAggregation() {
  try {
    const result = await Sale.aggregate([
      // Stage 1: Filter to only include sales from 2023
      {
        $match: {
          date: {
            $gte: new Date('2023-01-01'),
            $lt: new Date('2024-01-01'),
          },
        },
      },

      // Stage 2: Add a calculated field for revenue
      {
        $addFields: {
          revenue: { $multiply: ['$price', '$quantity'] },
        },
      },

      // Stage 3: Group by region and month
      {
        $group: {
          _id: {
            region: '$store.region',
            month: { $month: '$date' },
          },
          totalRevenue: { $sum: '$revenue' },
          averageOrderValue: { $avg: '$revenue' },
          orderCount: { $sum: 1 },
        },
      },

      // Stage 4: Sort by region and month
      {
        $sort: {
          '_id.region': 1,
          '_id.month': 1,
        },
      },

      // Stage 5: Reshape the output
      {
        $project: {
          region: '$_id.region',
          month: '$_id.month',
          totalRevenue: 1,
          averageOrderValue: 1,
          orderCount: 1,
```

```
            _id: 0,
          },
        },
      ]);

      console.log('Monthly revenue by region:', result);
    } catch (err) {
      console.error('Error in multi-stage aggregation:', err);
    }
  }

  // Advanced aggregation: $lookup (joins)
  async function lookupAggregation() {
    try {
      // Assuming we have a Product model with details about each product
      const result = await Sale.aggregate([
        { $match: { 'customer.type': 'wholesale' } },

        // Join with the products collection
        {
          $lookup: {
            from: 'products', // The collection to join with
            localField: 'product', // Field from the sales collection
            foreignField: 'name', // Field from the products collection
            as: 'productDetails', // Output array field
          },
        },

        // Unwind the joined array (each sale will have one product)
        { $unwind: '$productDetails' },

        // Group by product category (from the joined product details)
        {
          $group: {
            _id: '$productDetails.category',
            totalSales: { $sum: { $multiply: ['$price', '$quantity'] } },
            productsSold: { $sum: '$quantity' },
          },
        },

        // Sort by total sales
        { $sort: { totalSales: -1 } },
      ]);

      console.log('Wholesale sales by product category:', result);
    } catch (err) {
      console.error('Error in lookup aggregation:', err);
    }
  }

  // Advanced aggregation: Using $facet for multiple aggregations in one query
  async function facetAggregation() {
    try {
      const result = await Sale.aggregate([
```

```
    {
      $match: {
        date: {
          $gte: new Date('2023-01-01'),
          $lt: new Date('2024-01-01'),
        },
      },
    },

    {
      $facet: {
        // Facet 1: Sales by region
        salesByRegion: [
          {
            $group: {
              _id: '$store.region',
              totalSales: { $sum: { $multiply: ['$price', '$quantity'] } },
            },
          },
          { $sort: { totalSales: -1 } },
        ],

        // Facet 2: Top 5 products
        topProducts: [
          {
            $group: {
              _id: '$product',
              totalSold: { $sum: '$quantity' },
            },
          },
          { $sort: { totalSold: -1 } },
          { $limit: 5 },
        ],

        // Facet 3: Monthly trend
        monthlyTrend: [
          {
            $group: {
              _id: { $month: '$date' },
              revenue: { $sum: { $multiply: ['$price', '$quantity'] } },
            },
          },
          { $sort: { _id: 1 } },
        ],

        // Facet 4: Customer type breakdown
        customerTypes: [
          {
            $group: {
              _id: '$customer.type',
              count: { $sum: 1 },
              revenue: { $sum: { $multiply: ['$price', '$quantity'] } },
            },
          },
        ],
```

```
            ],
          },
        },
      ]);

      // The result contains all four aggregations
      console.log('Sales by region:', result[0].salesByRegion);
      console.log('Top products:', result[0].topProducts);
      console.log('Monthly trend:', result[0].monthlyTrend);
      console.log('Customer types:', result[0].customerTypes);
    } catch (err) {
      console.error('Error in facet aggregation:', err);
    }
  }

  // Advanced aggregation: Using $bucket for histogram
  async function bucketAggregation() {
    try {
      const result = await Sale.aggregate([
        // Create price range buckets
        {
          $bucket: {
            groupBy: '$price',
            boundaries: [0, 10, 20, 50, 100, 500],
            default: 'Above 500',
            output: {
              count: { $sum: 1 },
              averageQuantity: { $avg: '$quantity' },
              products: { $addToSet: '$product' },
            },
          },
        },
      ]);

      console.log('Sales by price range:', result);
    } catch (err) {
      console.error('Error in bucket aggregation:', err);
    }
  }

  // Advanced aggregation: Time series analysis
  async function timeSeriesAggregation() {
    try {
      const result = await Sale.aggregate([
        // Match sales for the past year
        {
          $match: {
            date: {
              $gte: new Date(
                new Date().setFullYear(new Date().getFullYear() - 1)
              ),
            },
          },
        },
```

```
    // Project to extract date parts
    {
      $project: {
        product: 1,
        revenue: { $multiply: ['$price', '$quantity'] },
        year: { $year: '$date' },
        month: { $month: '$date' },
        day: { $dayOfMonth: '$date' },
        dayOfWeek: { $dayOfWeek: '$date' },
        week: { $week: '$date' },
      },
    },

    // Group by week
    {
      $group: {
        _id: {
          year: '$year',
          week: '$week',
        },
        totalRevenue: { $sum: '$revenue' },
        orderCount: { $sum: 1 },
        averageOrderValue: { $avg: '$revenue' },
      },
    },

    // Sort by year and week
    {
      $sort: {
        '_id.year': 1,
        '_id.week': 1,
      },
    },
  ]);

  console.log('Weekly sales trend:', result);

  // Daily sales pattern by day of week
  const dayOfWeekPattern = await Sale.aggregate([
    {
      $project: {
        dayOfWeek: { $dayOfWeek: '$date' }, // 1 for Sunday, 2 for Monday, etc.
        revenue: { $multiply: ['$price', '$quantity'] },
      },
    },

    {
      $group: {
        _id: '$dayOfWeek',
        totalRevenue: { $sum: '$revenue' },
        averageRevenue: { $avg: '$revenue' },
        count: { $sum: 1 },
      },
```

```
      },

      { $sort: { _id: 1 } },
    ]);

    console.log('Sales by day of week:', dayOfWeekPattern);
  } catch (err) {
    console.error('Error in time series aggregation:', err);
  }
}

// Advanced aggregation: Geographic analysis
async function geoAggregation() {
  try {
    // Assuming stores have location field with GeoJSON format
    const result = await Sale.aggregate([
      // Enrich with store details
      {
        $lookup: {
          from: 'stores',
          localField: 'store.id',
          foreignField: 'storeId',
          as: 'storeDetails',
        },
      },

      { $unwind: '$storeDetails' },

      // Filter for stores with location data
      {
        $match: {
          'storeDetails.location': { $exists: true },
        },
      },

      // Group by region
      {
        $group: {
          _id: '$store.region',
          totalSales: { $sum: { $multiply: ['$price', '$quantity'] } },
          storeCount: { $addToSet: '$store.id' },
          // Calculate the geographic center of the region
          locations: { $push: '$storeDetails.location.coordinates' },
        },
      },

      // Calculate the size of the storeCount array
      {
        $addFields: {
          storeCount: { $size: '$storeCount' },
        },
      },

      { $sort: { totalSales: -1 } },
```

```
    ]);

    console.log('Sales by geographic region:', result);
  } catch (err) {
    console.error('Error in geo aggregation:', err);
  }
}

// Advanced aggregation: Using $graphLookup for hierarchical data
async function graphLookupAggregation() {
  try {
    // Assuming we have a categories collection with parent/child relationships
    const result = await Sale.aggregate([
      // Group sales by product
      {
        $group: {
          _id: '$product',
          totalSales: { $sum: { $multiply: ['$price', '$quantity'] } },
        },
      },

      // Join with the products collection
      {
        $lookup: {
          from: 'products',
          localField: '_id',
          foreignField: 'name',
          as: 'productDetails',
        },
      },

      { $unwind: '$productDetails' },

      // Recursive lookup to get all parent categories
      {
        $graphLookup: {
          from: 'categories',
          startWith: '$productDetails.categoryId',
          connectFromField: 'categoryId',
          connectToField: 'parentCategoryId',
          as: 'categoryHierarchy',
          depthField: 'level',
        },
      },

      // Group by top-level category
      { $unwind: '$categoryHierarchy' },
      { $match: { 'categoryHierarchy.parentCategoryId': null } },

      {
        $group: {
          _id: '$categoryHierarchy.name',
          totalRevenue: { $sum: '$totalSales' },
        },
```

```
      },

      { $sort: { totalRevenue: -1 } },
    ]);

    console.log('Sales by top-level category:', result);
  } catch (err) {
    console.error('Error in graph lookup aggregation:', err);
  }
}

// Custom aggregation for sales funnel analysis
async function salesFunnelAggregation() {
  try {
    // Assuming we have a sales pipeline with stages: lead, quote, negotiation,
closed
    const result = await Sale.aggregate([
      // Match only sales from the current quarter
      {
        $match: {
          date: {
            $gte: new Date(
              new Date().setMonth(Math.floor(new Date().getMonth() / 3) * 3)
            ),
            $lt: new Date(
              new Date().setMonth(Math.floor(new Date().getMonth() / 3) * 3 + 3)
            ),
          },
        },
      },

      // Group by stage
      {
        $group: {
          _id: '$stage',
          count: { $sum: 1 },
          value: { $sum: { $multiply: ['$price', '$quantity'] } },
        },
      },

      // Add conversion rates (would normally use $lookup with previous stage
data)
      {
        $addFields: {
          conversionRate: {
            $cond: {
              if: { $eq: ['$_id', 'lead'] },
              then: 1,
              else: {
                $cond: {
                  if: { $eq: ['$_id', 'quote'] },
                  then: 0.75, // Example conversion rate from lead to quote
                  else: {
                    $cond: {
```

```
                        if: { $eq: ['$_id', 'negotiation'] },
                        then: 0.5, // Example conversion rate from quote to
negotiation
                        else: 0.3, // Example conversion rate from negotiation to
closed
                      },
                    },
                  },
                },
              },
            },
          },
        },

        // Sort by funnel stage order
        {
          $sort: {
            stage: 1,
          },
        },
      ]);

      console.log('Sales funnel analysis:', result);
    } catch (err) {
      console.error('Error in sales funnel aggregation:', err);
    }
  }
```

Performance Considerations for Aggregation

```
  // Performance tips for aggregation pipelines
  async function optimizedAggregation() {
    try {
      // Bad practice: Moving $match stage after $group
      console.time('Unoptimized pipeline');
      const unoptimizedResult = await Sale.aggregate([
        // Group all documents first (processes entire collection)
        {
          $group: {
            _id: '$store.region',
            totalSales: { $sum: { $multiply: ['$price', '$quantity'] } },
          },
        },

        // Then filter the results (too late, already processed everything)
        {
          $match: {
            _id: 'West',
          },
        },
      ]);
```

```
console.timeEnd('Unoptimized pipeline');

// Good practice: Place $match stages early
console.time('Optimized pipeline');
const optimizedResult = await Sale.aggregate([
  // Filter documents first (reduces documents processed by later stages)
  {
    $match: {
      'store.region': 'West',
    },
  },

  // Then group the filtered documents
  {
    $group: {
      _id: '$store.region',
      totalSales: { $sum: { $multiply: ['$price', '$quantity'] } },
    },
  },
]);
console.timeEnd('Optimized pipeline');

// Good practice: Use $project or $addFields to include only necessary fields
console.time('Project fields');
const projectedResult = await Sale.aggregate([
  {
    $match: {
      'store.region': 'West',
    },
  },

  // Only keep fields needed for later stages
  {
    $project: {
      price: 1,
      quantity: 1,
    },
  },

  {
    $group: {
      _id: null,
      totalSales: { $sum: { $multiply: ['$price', '$quantity'] } },
    },
  },
]);
console.timeEnd('Project fields');

// Good practice: Use indexes to support your aggregation pipeline
// Create an index to support the common query pattern
await Sale.collection.createIndex({ 'store.region': 1, date: 1 });

console.time('Index-supported pipeline');
const indexedResult = await Sale.aggregate([
```

```
        // This $match can use the compound index
        {
          $match: {
            'store.region': 'West',
            date: { $gte: new Date('2023-01-01') },
          },
        },

        {
          $group: {
            _id: { $month: '$date' },
            totalSales: { $sum: { $multiply: ['$price', '$quantity'] } },
          },
        },
      ]);
      console.timeEnd('Index-supported pipeline');

      // Use explain() to analyze aggregation performance
      const explanation = await Sale.aggregate([
        { $match: { 'store.region': 'West' } },
        {
          $group: {
            _id: '$product',
            totalSales: { $sum: { $multiply: ['$price', '$quantity'] } },
          },
        },
      ]).explain('executionStats');

      console.log('Execution stats:', {
        executionTimeMillis: explanation.executionStats.executionTimeMillis,
        totalDocsExamined: explanation.executionStats.totalDocsExamined,
        usedIndex:
          explanation.queryPlanner.winningPlan.inputStage.indexName ||
          'No index used',
      });
    } catch (err) {
      console.error('Error in optimized aggregation:', err);
    }
  }

  // Allow disk use for large aggregations
  async function largeAggregation() {
    try {
      const result = await Sale.aggregate([
        // Complex aggregation that may exceed memory limits
        {
          $group: {
            _id: {
              product: '$product',
              customer: '$customer.name',
              store: '$store.id',
            },
            sales: { $sum: { $multiply: ['$price', '$quantity'] } },
          },
```

```
      },

      {
        $group: {
          _id: '$_id.product',
          customerCount: { $sum: 1 },
          totalSales: { $sum: '$sales' },
        },
      },

      { $sort: { totalSales: -1 } },
    ]).option({ allowDiskUse: true }); // Allow using disk for large operations

    console.log('Large aggregation result count:', result.length);
  } catch (err) {
    console.error('Error in large aggregation:', err);
  }
}

// Using cursor for aggregation with large result sets
async function aggregationWithCursor() {
  try {
    // Create an aggregation cursor instead of executing it directly
    const cursor = Sale.aggregate([
      { $match: { 'customer.type': 'retail' } },
      {
        $project: {
          product: 1,
          revenue: { $multiply: ['$price', '$quantity'] },
          date: 1,
        },
      },
    ]).cursor({ batchSize: 100 }); // Process in batches of 100

    let count = 0;
    let totalRevenue = 0;

    // Process documents one at a time
    await cursor.eachAsync((doc) => {
      count++;
      totalRevenue += doc.revenue;

      // Process in batches to avoid memory issues
      if (count % 1000 === 0) {
        console.log(
          `Processed ${count} documents. Current revenue: ${totalRevenue}`
        );
      }
    });

    console.log(
      `Aggregation complete. Processed ${count} documents with total revenue
${totalRevenue}`
    );
```

```
    } catch (err) {
      console.error('Error in cursor aggregation:', err);
    }
  }
```

Transaction Management in MongoDB

MongoDB supports multi-document transactions starting from version 4.0 for replica sets and 4.2 for sharded clusters.

Basic Transaction Structure

```
const mongoose = require('mongoose');
const { ObjectId } = require('mongodb');

// Sample models
const Account = mongoose.model(
  'Account',
  new mongoose.Schema({
    name: String,
    balance: Number,
  })
);

const Transaction = mongoose.model(
  'Transaction',
  new mongoose.Schema({
    fromAccount: { type: mongoose.Schema.Types.ObjectId, ref: 'Account' },
    toAccount: { type: mongoose.Schema.Types.ObjectId, ref: 'Account' },
    amount: Number,
    date: { type: Date, default: Date.now },
  })
);

// Basic transaction using MongoDB session
async function transferFunds(fromAccountId, toAccountId, amount) {
  // Start a session
  const session = await mongoose.startSession();

  try {
    // Start a transaction
    session.startTransaction();

    // Get account documents (within the transaction)
    const fromAccount = await Account.findById(fromAccountId).session(session);
    const toAccount = await Account.findById(toAccountId).session(session);

    if (!fromAccount || !toAccount) {
      throw new Error('One or both accounts not found');
    }
```

```
      if (fromAccount.balance < amount) {
        throw new Error('Insufficient funds');
      }

      // Update account balances
      await Account.updateOne(
        { _id: fromAccountId },
        { $inc: { balance: -amount } }
      ).session(session);

      await Account.updateOne(
        { _id: toAccountId },
        { $inc: { balance: amount } }
      ).session(session);

      // Record the transaction
      await Transaction.create(
        [
          {
            fromAccount: fromAccountId,
            toAccount: toAccountId,
            amount: amount,
            date: new Date(),
          },
        ],
        { session }
      );

      // Commit the transaction
      await session.commitTransaction();
      console.log('Transaction committed successfully');
    } catch (error) {
      // Abort transaction on error
      await session.abortTransaction();
      console.error('Transaction aborted:', error);
      throw error;
    } finally {
      // End session
      session.endSession();
    }
  }

  // Usage
  async function performTransfer() {
    try {
      // Create some test accounts first
      const accountA = await Account.create({ name: 'Account A', balance: 1000 });
      const accountB = await Account.create({ name: 'Account B', balance: 500 });

      console.log('Initial account states:');
      console.log('Account A:', accountA);
      console.log('Account B:', accountB);

      // Perform a transfer
```

```
    await transferFunds(accountA._id, accountB._id, 300);

    // Check final balances
    const updatedA = await Account.findById(accountA._id);
    const updatedB = await Account.findById(accountB._id);

    console.log('Final account states:');
    console.log('Account A:', updatedA);
    console.log('Account B:', updatedB);

    // Get transaction history
    const transactions = await Transaction.find({
      $or: [{ fromAccount: accountA._id }, { toAccount: accountA._id }],
    }).populate('fromAccount toAccount');

    console.log('Transaction history:', transactions);
  } catch (err) {
    console.error('Error performing transfer:', err);
  }
}
```

Advanced Transaction Patterns

```
// Complex transaction with error simulation
async function orderProcessingTransaction(userId, productId, quantity) {
  const session = await mongoose.startSession();

  try {
    session.startTransaction();

    // Check user exists and has sufficient credit
    const user = await User.findById(userId).session(session);
    if (!user) {
      throw new Error('User not found');
    }

    // Check product exists and has sufficient inventory
    const product = await Product.findById(productId).session(session);
    if (!product) {
      throw new Error('Product not found');
    }

    if (product.inventory < quantity) {
      throw new Error('Insufficient inventory');
    }

    const totalCost = product.price * quantity;

    if (user.credit < totalCost) {
      throw new Error('Insufficient credit');
    }
```

```
    // Update product inventory
    await Product.updateOne(
      { _id: productId },
      { $inc: { inventory: -quantity } }
    ).session(session);

    // Update user credit
    await User.updateOne(
      { _id: userId },
      { $inc: { credit: -totalCost } }
    ).session(session);

    // Create order
    const order = await Order.create(
      [
        {
          user: userId,
          product: productId,
          quantity: quantity,
          totalCost: totalCost,
          status: 'completed',
          date: new Date(),
        },
      ],
      { session }
    );

    // Simulate a random error to test transaction abort
    if (Math.random() < 0.3) {
      // 30% chance of error
      throw new Error('Random error occurred');
    }

    // Commit the transaction
    await session.commitTransaction();
    console.log('Order processed successfully:', order[0]._id);
    return order[0];
  } catch (error) {
    // Abort transaction on error
    await session.abortTransaction();
    console.error('Order processing failed:', error);
    throw error;
  } finally {
    session.endSession();
  }
}

// Retry logic for transactions
async function retryTransaction(operationFn, maxRetries = 3) {
  let retryCount = 0;

  while (retryCount < maxRetries) {
    try {
```

```
      return await operationFn();
    } catch (error) {
      // Check if error is a transient error that warrants a retry
      if (
        error.name === 'MongoError' &&
        (error.code === 112 || // Write conflict
          error.code === 251) // Transaction aborted
      ) {
        retryCount++;
        console.log(`Retrying transaction (${retryCount}/${maxRetries})`);

        // Exponential backoff
        const delay = Math.pow(2, retryCount) * 100;
        await new Promise((resolve) => setTimeout(resolve, delay));
      } else {
        // Non-transient error, don't retry
        throw error;
      }
    }
  }

  throw new Error(`Transaction failed after ${maxRetries} retries`);
}

// Usage with retry logic
async function processOrderWithRetry(userId, productId, quantity) {
  try {
    const result = await retryTransaction(() => {
      return orderProcessingTransaction(userId, productId, quantity);
    });

    console.log('Order processed with retry logic:', result);
    return result;
  } catch (err) {
    console.error('Order processing failed after retries:', err);
    throw err;
  }
}

// Distributed transaction pattern (two-phase commit simulation)
async function twoPhaseCommit(operations) {
  // First phase: prepare - validate and lock resources
  const transactionId = new ObjectId();

  try {
    // Create transaction record in 'preparing' state
    await TransactionLog.create({
      _id: transactionId,
      state: 'preparing',
      operations: operations,
      startTime: new Date(),
    });

    // Prepare each operation
```

```
      for (const op of operations) {
        // Lock resources
        await ResourceLock.create({
          resourceId: op.resourceId,
          transactionId: transactionId,
          operation: op.type,
          lockedAt: new Date(),
        });

        // Validate operation
        await validateOperation(op);
      }

      // Update transaction state to 'prepared'
      await TransactionLog.updateOne(
        { _id: transactionId },
        { $set: { state: 'prepared' } }
      );

      // Second phase: commit - apply changes
      for (const op of operations) {
        await applyOperation(op);
      }

      // Update transaction state to 'committed'
      await TransactionLog.updateOne(
        { _id: transactionId },
        { $set: { state: 'committed', completedAt: new Date() } }
      );

      // Release locks
      await ResourceLock.deleteMany({ transactionId });

      return { success: true, transactionId };
    } catch (error) {
      // Roll back and release locks on error
      await TransactionLog.updateOne(
        { _id: transactionId },
        { $set: { state: 'aborted', error: error.message } }
      );

      // Release locks
      await ResourceLock.deleteMany({ transactionId });

      console.error('Two-phase commit failed:', error);
      throw error;
    }
  }
```

Transaction Best Practices

1. **Keep Transactions Short**

- Long-running transactions block other operations and increase the chance of conflicts
- Minimize the number of operations in a transaction
- Avoid operations that may take a long time, like complex queries or network calls

2. **Handle Transient Errors with Retries**

- Transactions can fail due to temporary conditions (network issues, write conflicts)
- Implement retry logic with exponential backoff
- Only retry for transient errors; don't retry for logical errors

3. **Use Session Correctly**

- Always pass the session to all operations in the transaction
- End the session after the transaction completes or aborts
- Consider using a separate session for each transaction

4. **Be Aware of Transaction Limitations**

- Transactions have a 60-second maximum runtime
- There's a limit to the size of the operations in a transaction
- Operations that affect multiple shards may have performance implications

5. **Error Handling**

- Always use try/catch/finally blocks
- Ensure session.abortTransaction() is called on errors
- Ensure session.endSession() is always called

6. **Monitor Transaction Performance**

- Long-running transactions can impact overall database performance
- Use MongoDB's monitoring tools to track transaction metrics
- Log transaction execution times and success/failure rates

```
// Example implementing these best practices
async function robustTransaction(operationFn) {
  const session = await mongoose.startSession();
  let result;

  try {
    // Start timing the transaction
    const startTime = Date.now();

    // Start transaction
    session.startTransaction();

    // Execute the operation function with the session
    result = await operationFn(session);

    // Commit the transaction
    await session.commitTransaction();
```

```
      // Log successful transaction
      const duration = Date.now() - startTime;
      console.log(`Transaction completed successfully in ${duration}ms`);

      return result;
    } catch (error) {
      // Check if transaction is active before aborting
      if (session.inTransaction()) {
        await session.abortTransaction();
      }

      // Classify and log the error
      if (
        error.name === 'MongoError' &&
        (error.code === 112 || error.code === 251)
      ) {
        console.error('Transaction failed due to transient error:', error);
      } else {
        console.error('Transaction failed due to logical error:', error);
      }

      // Re-throw the error for higher-level handling
      throw error;
    } finally {
      // Always end the session
      session.endSession();
    }
  }

  // Usage
  async function transferFundsRobust(fromAccountId, toAccountId, amount) {
    try {
      return await robustTransaction(async (session) => {
        const fromAccount = await Account.findById(fromAccountId).session(
          session
        );
        const toAccount = await Account.findById(toAccountId).session(session);

        if (!fromAccount || !toAccount) {
          throw new Error('One or both accounts not found');
        }

        if (fromAccount.balance < amount) {
          throw new Error('Insufficient funds');
        }

        await Account.updateOne(
          { _id: fromAccountId },
          { $inc: { balance: -amount } }
        ).session(session);

        await Account.updateOne(
          { _id: toAccountId },
          { $inc: { balance: amount } }
```

```
      ).session(session);

      const transaction = await Transaction.create(
        [
          {
            fromAccount: fromAccountId,
            toAccount: toAccountId,
            amount: amount,
            date: new Date(),
          },
        ],
        { session }
      );

      return transaction[0];
    });
  } catch (error) {
    console.error('Transfer funds operation failed:', error);
    throw error;
  }
}
```

MongoDB Security Considerations

Authentication and Authorization

```
// Connecting with authentication
const mongoose = require('mongoose');

// Connection with authentication
async function connectWithAuth() {
  try {
    await mongoose.connect(
      'mongodb://username:password@localhost:27017/mydatabase',
      {
        useNewUrlParser: true,
        useUnifiedTopology: true,
        authSource: 'admin', // Database where user credentials are stored
      }
    );

    console.log('Connected to MongoDB with authentication');
  } catch (err) {
    console.error('MongoDB connection error:', err);
  }
}

// Connection with SSL/TLS
async function connectWithSSL() {
  try {
    await mongoose.connect('mongodb://localhost:27017/mydatabase', {
```

```
      useNewUrlParser: true,
      useUnifiedTopology: true,
      ssl: true,
      sslCA: '/path/to/ca.pem', // Path to CA certificate
      sslKey: '/path/to/client.key', // Path to client key
      sslCert: '/path/to/client.crt', // Path to client certificate
    });

    console.log('Connected to MongoDB with SSL/TLS');
  } catch (err) {
    console.error('MongoDB SSL connection error:', err);
  }
}

// MongoDB Atlas connection with authentication
async function connectToAtlas() {
  try {
    await mongoose.connect(
      'mongodb+srv://username:password@cluster0.mongodb.net/mydatabase?
retryWrites=true&w=majority',
      {
        useNewUrlParser: true,
        useUnifiedTopology: true,
      }
    );

    console.log('Connected to MongoDB Atlas');
  } catch (err) {
    console.error('MongoDB Atlas connection error:', err);
  }
}
```

Creating MongoDB Users with Specific Roles

Here's how to create users in MongoDB with appropriate roles:

```
// Using MongoDB shell commands (mongosh)

// Create admin user
db.createUser({
  user: 'adminUser',
  pwd: 'securePassword',
  roles: [{ role: 'userAdminAnyDatabase', db: 'admin' }],
});

// Create application user with read/write access to specific database
db.createUser({
  user: 'appUser',
  pwd: 'appPassword',
  roles: [{ role: 'readWrite', db: 'mydatabase' }],
});
```

```
// Create read-only user
db.createUser({
  user: 'readOnlyUser',
  pwd: 'readPassword',
  roles: [{ role: 'read', db: 'mydatabase' }],
});

// Create user with custom role
db.createRole({
  role: 'customRole',
  privileges: [
    {
      resource: { db: 'mydatabase', collection: 'products' },
      actions: ['find', 'update'],
    },
    {
      resource: { db: 'mydatabase', collection: 'categories' },
      actions: ['find'],
    },
  ],
  roles: [],
});

db.createUser({
  user: 'customUser',
  pwd: 'customPassword',
  roles: [{ role: 'customRole', db: 'mydatabase' }],
});
```

Data Encryption

MongoDB provides several ways to encrypt data:

1. **Client-Side Field Level Encryption**:

```
const mongoose = require('mongoose');
const { ClientEncryption } = require('mongodb-client-encryption');
const { MongoClient } = require('mongodb');

async function setupEncryption() {
  // Generate a local key
  const localMasterKey = crypto.randomBytes(96);

  // Key vault namespace
  const keyVaultNamespace = 'encryption.__keyVault';

  // KMS providers configuration
  const kmsProviders = {
    local: {
      key: localMasterKey,
```

```
    },
  };

  // Create the client encryption object
  const clientEncryption = new ClientEncryption(mongoClient, {
    keyVaultNamespace,
    kmsProviders,
  });

  // Create a data encryption key
  const dataKeyId = await clientEncryption.createDataKey('local');
  console.log('Created data key ID:', dataKeyId.toString('hex'));

  return { dataKeyId, clientEncryption };
}

// Schema with encrypted fields
async function createEncryptedSchema() {
  const { dataKeyId, clientEncryption } = await setupEncryption();

  const userSchema = new mongoose.Schema({
    username: String,
    email: String,
    // Encrypted fields
    ssn: {
      type: String,
      get: async function () {
        // Decrypt when retrieving
        if (this._ssn) {
          try {
            return await clientEncryption.decrypt(this._ssn);
          } catch (err) {
            console.error('Decryption error:', err);
            return this._ssn; // Return encrypted value if decryption fails
          }
        }
        return undefined;
      },
      set: async function (value) {
        // Encrypt when setting
        if (value) {
          try {
            this._ssn = await clientEncryption.encrypt(value, {
              algorithm: 'AEAD_AES_256_CBC_HMAC_SHA_512-Deterministic',
              keyId: dataKeyId,
            });
          } catch (err) {
            console.error('Encryption error:', err);
            this._ssn = value; // Store unencrypted as fallback
          }
        }
        return value;
      },
    },
```

```
  });

  return mongoose.model('User', userSchema);
}
```

2. **Server-Side Field Level Encryption** (managed by MongoDB)

```
// Using MongoDB Enterprise with server-side encryption
// This is configured at the MongoDB server level
```

3. **Encrypted Storage Engine** (MongoDB Enterprise)

```
// Configure MongoDB Enterprise server with encryption at rest
// mongod --enableEncryption --encryptionKeyFile <path-to-key-file>
```

Network Security Best Practices

1. **Use TLS/SSL for Connections**:
   - Always enable TLS/SSL for MongoDB connections
   - Verify certificates to prevent man-in-the-middle attacks
2. **Firewall Configuration**:
   - Restrict MongoDB port access (default is 27017)
   - Only allow connections from application servers
   - Use VPC peering or private networks in cloud environments
3. **IP Binding**:
   - Bind MongoDB to specific IP addresses, not 0.0.0.0
   - Configuration: `bindIp: 127.0.0.1,192.168.1.100`
4. **VPN or SSH Tunneling**:
   - For remote connections, use VPN or SSH tunneling
   - Example SSH tunnel:

```
ssh -L 27017:localhost:27017 user@mongodb-server
```

Application-Level Security

```
// Input validation to prevent NoSQL injection
function safeQuery(userInput) {
```

```
  // Validate input is a string
  if (typeof userInput !== 'string') {
    throw new Error('Invalid input type');
  }

  // Check for suspicious patterns
  const suspiciousPatterns = [
    /\$/, // MongoDB operators
    /\{.*\}/, // JSON objects
    /\.\$/, // Dot notation with operators
  ];

  for (const pattern of suspiciousPatterns) {
    if (pattern.test(userInput)) {
      throw new Error('Potentially unsafe input detected');
    }
  }

  return userInput;
}

// Using input validation with MongoDB queries
async function findUserSafely(username) {
  try {
    const safeUsername = safeQuery(username);
    return await User.findOne({ username: safeUsername });
  } catch (err) {
    console.error('Security error:', err);
    throw new Error('Invalid search parameter');
  }
}

// Using mongoose schema validation
const userSchema = new mongoose.Schema({
  username: {
    type: String,
    required: true,
    validate: {
      validator: function (v) {
        return /^[a-zA-Z0-9_]{3,30}$/.test(v);
      },
      message: (props) => `${props.value} is not a valid username!`,
    },
  },
  email: {
    type: String,
    required: true,
    validate: {
      validator: function (v) {
        return /^\S+@\S+\.\S+$/.test(v);
      },
      message: (props) => `${props.value} is not a valid email!`,
    },
  },
```

```
});

// Using MongoDB filter sanitization
function sanitizeFilter(filter) {
  // Deep copy to avoid modifying the original
  const sanitized = JSON.parse(JSON.stringify(filter));

  // Remove $ operators from keys that start with $
  // (except for valid query operators)
  const validOperators = [
    '$eq',
    '$gt',
    '$gte',
    '$lt',
    '$lte',
    '$in',
    '$nin',
    '$ne',
    '$exists',
  ];

  function sanitizeObject(obj) {
    for (const key in obj) {
      if (key.startsWith('$') && !validOperators.includes(key)) {
        delete obj[key];
      } else if (typeof obj[key] === 'object' && obj[key] !== null) {
        sanitizeObject(obj[key]);
      }
    }
  }

  sanitizeObject(sanitized);
  return sanitized;
}

// Using sanitized filters
async function findDocumentsSafely(filter) {
  const sanitizedFilter = sanitizeFilter(filter);
  return await Collection.find(sanitizedFilter);
}
```

MongoDB Production Deployment Best Practices

Replica Sets for High Availability

```
// Connecting to a MongoDB replica set
const mongoose = require('mongoose');

async function connectToReplicaSet() {
  try {
    await mongoose.connect(
```

```
      'mongodb://server1:27017,server2:27017,server3:27017/mydatabase',
      {
        useNewUrlParser: true,
        useUnifiedTopology: true,
        replicaSet: 'rs0',
        readPreference: 'secondaryPreferred', // Read from secondaries when
possible
        w: 'majority', // Wait for writes to be acknowledged by majority
        retryWrites: true,
      }
    );

    console.log('Connected to MongoDB replica set');
  } catch (err) {
    console.error('Failed to connect to replica set:', err);
  }
}
```

Sharding for Horizontal Scaling

MongoDB sharding distributes data across multiple machines to support deployments with very large data sets and high throughput operations.

```
// Connecting to a sharded cluster
async function connectToShardedCluster() {
  try {
    await mongoose.connect('mongodb://mongos1:27017,mongos2:27017/mydatabase', {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });

    console.log('Connected to MongoDB sharded cluster');
  } catch (err) {
    console.error('Failed to connect to sharded cluster:', err);
  }
}

// Defining a sharded collection in MongoDB shell
/*
use mydatabase
sh.enableSharding("mydatabase")

// Shard using hashed sharding (evenly distributes data)
db.products.createIndex({ _id: "hashed" })
sh.shardCollection("mydatabase.products", { _id: "hashed" })

// Alternatively, shard using ranged sharding (for targeted queries)
db.users.createIndex({ country: 1, city: 1 })
sh.shardCollection("mydatabase.users", { country: 1, city: 1 })
*/
```

Backup and Restore

```
# MongoDB backup using mongodump
mongodump --uri="mongodb://username:password@localhost:27017/mydatabase" --
out=/backup/location

# MongoDB restore using mongorestore
mongorestore --uri="mongodb://username:password@localhost:27017" --drop
/backup/location

# Point-in-time recovery with oplog
mongodump --uri="mongodb://username:password@localhost:27017/mydatabase" --oplog -
-out=/backup/location
mongorestore --uri="mongodb://username:password@localhost:27017" --oplogReplay
/backup/location
```

Monitoring and Maintenance

```
// Using mongoose to get database stats
async function getDBStats() {
  try {
    const stats = await mongoose.connection.db.stats();
    console.log('Database stats:', {
      collections: stats.collections,
      views: stats.views,
      objects: stats.objects,
      avgObjSize: stats.avgObjSize,
      dataSize: `${(stats.dataSize / 1024 / 1024).toFixed(2)} MB`,
      storageSize: `${(stats.storageSize / 1024 / 1024).toFixed(2)} MB`,
      indexes: stats.indexes,
      indexSize: `${(stats.indexSize / 1024 / 1024).toFixed(2)} MB`,
    });
  } catch (err) {
    console.error('Error getting database stats:', err);
  }
}

// Get collection stats
async function getCollectionStats(collectionName) {
  try {
    const stats = await mongoose.connection.db
      .collection(collectionName)
      .stats();
    console.log(`${collectionName} stats:`, {
      count: stats.count,
      size: `${(stats.size / 1024 / 1024).toFixed(2)} MB`,
      avgObjSize: stats.avgObjSize,
      storageSize: `${(stats.storageSize / 1024 / 1024).toFixed(2)} MB`,
      indexed: stats.nindexes,
      indexSize: `${(stats.totalIndexSize / 1024 / 1024).toFixed(2)} MB`,
```

```
    });
  } catch (err) {
    console.error(`Error getting ${collectionName} stats:`, err);
  }
}

// MongoDB server status (requires admin privileges)
async function getServerStatus() {
  try {
    const admin = mongoose.connection.db.admin();
    const status = await admin.serverStatus();

    console.log('Server status:', {
      version: status.version,
      uptime: `${(status.uptime / 3600).toFixed(2)} hours`,
      connections: status.connections.current,
      network: {
        bytesIn: `${(status.network.bytesIn / 1024 / 1024).toFixed(2)} MB`,
        bytesOut: `${(status.network.bytesOut / 1024 / 1024).toFixed(2)} MB`,
        requests: status.network.numRequests,
      },
      memory: {
        resident: `${(status.mem.resident / 1024).toFixed(2)} MB`,
        virtual: `${(status.mem.virtual / 1024).toFixed(2)} MB`,
      },
    });
  } catch (err) {
    console.error('Error getting server status:', err);
  }
}

// Collection maintenance - compact and reIndex
async function maintainCollection(collectionName) {
  try {
    console.log(`Starting maintenance for ${collectionName}`);

    // Get initial stats
    await getCollectionStats(collectionName);

    // Compact the collection (reclaims space)
    await mongoose.connection.db.command({
      compact: collectionName,
      force: true,
    });
    console.log(`Compacted ${collectionName}`);

    // Rebuild indexes
    await mongoose.connection.db.collection(collectionName).reIndex();
    console.log(`Rebuilt indexes for ${collectionName}`);

    // Get final stats
    await getCollectionStats(collectionName);
  } catch (err) {
    console.error(`Error maintaining ${collectionName}:`, err);
```

```
  }
}
```

Performance Tuning Tips

1. **Connection Pooling**:

```
// Configure connection pool size based on workload
mongoose.connect('mongodb://localhost:27017/mydatabase', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  maxPoolSize: 50, // Default is 5, increase for high-concurrency apps
  minPoolSize: 5, // Maintain at least 5 connections
  maxIdleTimeMS: 30000, // Close idle connections after 30 seconds
});
```

2. **Index Optimization**:

```
// Check index usage
async function checkIndexUsage() {
  try {
    const indexStats = await mongoose.connection.db
      .collection('users')
      .aggregate([
        { $indexStats: {} },
        {
          $project: {
            name: 1,
            accesses: 1,
            operations: '$accesses.ops',
            since: '$accesses.since',
          },
        },
      ])
      .toArray();

    console.log('Index usage stats:', indexStats);

    // Identify unused indexes
    const unusedIndexes = indexStats.filter((stat) => stat.operations === 0);
    console.log('Unused indexes:', unusedIndexes);
  } catch (err) {
    console.error('Error checking index usage:', err);
  }
}

// Remove unused indexes
async function removeUnusedIndexes(collectionName, indexNames) {
  try {
```

```
    for (const indexName of indexNames) {
      await mongoose.connection.db
        .collection(collectionName)
        .dropIndex(indexName);
      console.log(`Dropped unused index ${indexName} from ${collectionName}`);
    }
  } catch (err) {
    console.error('Error removing unused indexes:', err);
  }
}
```

3. **Query Optimization**:

```
// Use explain to analyze query performance
async function analyzeQuery(collection, query, options = {}) {
  try {
    const explanation = await mongoose.connection.db
      .collection(collection)
      .find(query, options)
      .explain('executionStats');

    console.log('Query explanation:', {
      executionTimeMillis: explanation.executionStats.executionTimeMillis,
      totalDocsExamined: explanation.executionStats.totalDocsExamined,
      totalKeysExamined: explanation.executionStats.totalKeysExamined,
      nReturned: explanation.executionStats.nReturned,
      indexesUsed:
        explanation.queryPlanner.winningPlan.inputStage.indexName ||
        'No index used',
    });

    // Analyze index efficiency
    const ratio =
      explanation.executionStats.nReturned /
      (explanation.executionStats.totalDocsExamined || 1);

    console.log('Query efficiency:', {
      returnRatio: ratio.toFixed(2),
      efficient: ratio > 0.5, // Rule of thumb: at least 50% of examined docs
should be returned
      recommendation:
        ratio < 0.5 ? 'Consider a more specific index' : 'Query is efficient',
    });
  } catch (err) {
    console.error('Error analyzing query:', err);
  }
}
```

4. **Write Concern Configuration**:

```
// Configure write concern based on requirements
const options = {
  // For critical data where durability is important
  critical: {
    writeConcern: {
      w: 'majority', // Wait for majority of replica set nodes
      j: true, // Wait for journal commit
      wtimeout: 5000, // Timeout after 5 seconds
    },
  },

  // For high-throughput scenarios where speed is more important
  highThroughput: {
    writeConcern: {
      w: 1, // Wait for primary only
      j: false, // Don't wait for journal commit
    },
  },
};

// Usage example
async function createUserWithWriteConcern(userData, concern = 'critical') {
  try {
    const user = new User(userData);
    await user.save(options[concern]);
    console.log('User created with write concern:', concern);
  } catch (err) {
    console.error('Error creating user:', err);
  }
}
```

5. **Document Design Optimization**:

```
// Avoid excessively large documents
// MongoDB has a 16MB document size limit

// Consider when to denormalize/normalize
/*
Normalized (separate collections):
- When data is frequently updated
- When most queries only need a subset of the data
- For many-to-many relationships

Denormalized (embedded documents):
- When data is mostly read
- When data is queried together
- For one-to-many relationships with limited "many" side
*/

// Example of appropriate denormalization
const productSchema = new mongoose.Schema({
```

```
  name: String,
  price: Number,
  // Embed category info that rarely changes
  category: {
    name: String,
    slug: String,
  },
  // Don't embed reviews (they change often and there can be many)
  // Instead, use a reference
  reviews: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Review',
    },
  ],
});
```

Template Engines with Express

Overview of Template Engines

Template engines enable you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client.

Popular Template Engines

1. **EJS (Embedded JavaScript)**
   - Uses JavaScript within HTML with `<% %>` tags
   - Simple syntax, familiar to JavaScript developers
2. **Pug (formerly Jade)**
   - Minimalist syntax with significant indentation
   - No closing tags, shorter code, cleaner look
3. **Handlebars**
   - Logicless templates with `{{}}` syntax
   - Focused on keeping logic out of views

Setting Up Template Engines with Express

Setting Up EJS

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

```
// Set EJS as the view engine
app.set('view engine', 'ejs');

// Set the directory where your views are located
app.set('views', path.join(__dirname, 'views'));

// Optional: Custom configuration
app.set('view cache', process.env.NODE_ENV === 'production'); // Enable caching in
production
app.locals.title = 'My EJS App'; // Global variables available in all views

// Route that renders a view
app.get('/', (req, res) => {
  // Render the index view with data
  res.render('index', {
    title: 'Home Page',
    message: 'Welcome to EJS!',
    users: ['John', 'Jane', 'Bob'],
  });
});

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

Example EJS template (views/index.ejs):

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
  </head>
  <body>
    <h1><%= title %></h1>
    <p><%= message %></p>

    <h2>Users</h2>
    <ul>
      <% users.forEach(user => { %>
      <li><%= user %></li>
      <% }); %>
    </ul>
  </body>
</html>
```

Setting Up Pug

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

```
const app = express();

// Set Pug as the view engine
app.set('view engine', 'pug');
app.set('views', path.join(__dirname, 'views'));

app.get('/', (req, res) => {
  res.render('index', {
    title: 'Home Page',
    message: 'Welcome to Pug!',
    users: ['John', 'Jane', 'Bob'],
  });
});

app.listen(3000);
```

Example Pug template (views/index.pug):

```
doctype html
html
  head
    title= title
  body
    h1= title
    p= message

    h2 Users
    ul
      each user in users
        li= user
```

Setting Up Handlebars

```
const express = require('express');
const path = require('path');
const exphbs = require('express-handlebars');
const app = express();

// Set up Handlebars
const hbs = exphbs.create({
  extname: '.hbs', // Use .hbs extension instead of .handlebars
  defaultLayout: 'main', // Default layout
  layoutsDir: path.join(__dirname, 'views/layouts'), // Layouts directory
  partialsDir: path.join(__dirname, 'views/partials'), // Partials directory
});

// Register handlebars as the view engine
app.engine('.hbs', hbs.engine);
app.set('view engine', '.hbs');
```

```
app.set('views', path.join(__dirname, 'views'));

app.get('/', (req, res) => {
  res.render('home', {
    title: 'Home Page',
    message: 'Welcome to Handlebars!',
    users: ['John', 'Jane', 'Bob'],
  });
});

app.listen(3000);
```

Example Handlebars templates:

```
<!-- views/layouts/main.hbs -->
<!DOCTYPE html>
<html>
<head>
  <title>{{title}}</title>
</head>
<body>
  {{> header}}

  <div class="container">
    {{{body}}}
  </div>

  {{> footer}}
</body>
</html>

<!-- views/partials/header.hbs -->
<header>
  <h1>My Handlebars App</h1>
  <nav>
    <a href="/">Home</a>
    <a href="/about">About</a>
  </nav>
</header>

<!-- views/partials/footer.hbs -->
<footer>
  <p>&copy; 2023 My App</p>
</footer>

<!-- views/home.hbs -->
<h2>{{title}}</h2>
<p>{{message}}</p>

<h3>Users</h3>
<ul>
  {{#each users}}
```

```
    <li>{{this}}</li>
  {{/each}}
</ul>
```

Template Engine Features and Usage

Creating Layouts, Partials, and Includes

EJS Partials and Includes:

```
// File structure
// views/
//   ├── partials/
//   │   ├── header.ejs
//   │   └── footer.ejs
//   └── index.ejs

// app.js
app.get('/', (req, res) => {
  res.render('index', {
    title: 'Home Page',
    user: { name: 'John Doe' },
  });
});
```

```
<!-- views/partials/header.ejs -->
<!DOCTYPE html>
<html>
  <head>
    <title><%= title %></title>
    <link rel="stylesheet" href="/css/style.css" />
  </head>
  <body>
    <header>
      <h1><%= title %></h1>
      <% if (user) { %>
      <p>Welcome, <%= user.name %></p>
      <% } %>
      <nav>
        <ul>
          <li><a href="/">Home</a></li>
          <li><a href="/about">About</a></li>
          <li><a href="/contact">Contact</a></li>
        </ul>
      </nav>
    </header>

    <!-- views/partials/footer.ejs -->
    <footer>
```

```
      <p>&copy; <%= new Date().getFullYear() %> My App</p>
    </footer>
  </body>
</html>

<!-- views/index.ejs -->
<%- include('partials/header') %>

<main>
  <h2>Welcome to the Home Page</h2>
  <p>This is the main content of the page.</p>
</main>

<%- include('partials/footer') %>
```

Pug Layouts and Includes:

```
// File structure
// views/
//   ├── layout.pug
//   ├── includes/
//   │   ├── header.pug
//   │   └── footer.pug
//   └── index.pug

// app.js
app.get('/', (req, res) => {
  res.render('index', {
    title: 'Home Page',
    user: { name: 'John Doe' },
  });
});
```

```
// views/layout.pug
doctype html
html
  head
    title= title
    link(rel="stylesheet", href="/css/style.css")
  body
    include includes/header.pug

    block content

    include includes/footer.pug

// views/includes/header.pug
header
  h1= title
```

```
  if user
    p Welcome, #{user.name}
  nav
    ul
      li: a(href="/") Home
      li: a(href="/about") About
      li: a(href="/contact") Contact

// views/includes/footer.pug
footer
  p &copy; #{new Date().getFullYear()} My App

// views/index.pug
extends layout

block content
  main
    h2 Welcome to the Home Page
    p This is the main content of the page.
```

Handlebars Layouts and Partials:

```
// File structure
// views/
//   ├── layouts/
//   │   └── main.hbs
//   ├── partials/
//   │   ├── header.hbs
//   │   └── footer.hbs
//   └── home.hbs

// app.js
const hbs = exphbs.create({
  defaultLayout: 'main',
  layoutsDir: path.join(__dirname, 'views/layouts'),
  partialsDir: path.join(__dirname, 'views/partials'),
});

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

```
<!-- views/layouts/main.hbs -->
<!DOCTYPE html>
<html>
<head>
  <title>{{title}}</title>
  <link rel="stylesheet" href="/css/style.css">
</head>
<body>
```

```
  {{> header}}

  <main>
    {{{body}}}
  </main>

  {{> footer}}
</body>
</html>

<!-- views/partials/header.hbs -->
<header>
  <h1>{{title}}</h1>
  {{#if user}}
    <p>Welcome, {{user.name}}</p>
  {{/if}}
  <nav>
    <ul>
      <li><a href="/">Home</a></li>
      <li><a href="/about">About</a></li>
      <li><a href="/contact">Contact</a></li>
    </ul>
  </nav>
</header>

<!-- views/partials/footer.hbs -->
<footer>
  <p>&copy; {{currentYear}} My App</p>
</footer>

<!-- views/home.hbs -->
<h2>Welcome to the Home Page</h2>
<p>This is the main content of the page.</p>
```

Passing Data to Templates

Basic Data Passing:

```
app.get('/product/:id', async (req, res) => {
  try {
    const product = await Product.findById(req.params.id);
    if (!product) {
      return res.status(404).render('error', {
        message: 'Product not found',
      });
    }

    res.render('product', {
      title: product.name,
      product: product,
      relatedProducts: await Product.find({ category: product.category }).limit(
```

```
        4
      ),
      isAdmin: req.user && req.user.role === 'admin',
      formatPrice: (price) => `$${price.toFixed(2)}`,
    });
  } catch (err) {
    res.status(500).render('error', {
      message: 'Server error',
      error: process.env.NODE_ENV === 'development' ? err : {},
    });
  }
});
```

Using Locals for Global Variables:

```
// Set application-wide variables available in all templates
app.locals.siteName = 'My E-commerce Site';
app.locals.currentYear = new Date().getFullYear();
app.locals.supportEmail = 'support@example.com';

// Set request-specific variables using middleware
app.use((req, res, next) => {
  res.locals.user = req.user || null;
  res.locals.isAuthenticated = req.isAuthenticated();
  res.locals.messages = req.flash(); // If using connect-flash
  next();
});
```

Conditional Rendering and Loops

EJS:

```
<!-- Conditionals -->
<% if (user && user.isAdmin) { %>
<div class="admin-panel">
  <h3>Admin Panel</h3>
  <a href="/admin/dashboard">Dashboard</a>
</div>
<% } else if (user) { %>
<p>You don't have admin privileges</p>
<% } else { %>
<p>Please <a href="/login">login</a> to access more features</p>
<% } %>

<!-- Loops -->
<h2>Product List</h2>
<% if (products && products.length > 0) { %>
<ul class="product-list">
  <% products.forEach(function(product) { %>
```

```
    <li>
      <h3><%= product.name %></h3>
      <p><%= product.description %></p>
      <span class="price"><%= formatPrice(product.price) %></span>
      <% if (product.inStock) { %>
      <button class="add-to-cart" data-id="<%= product._id %>">
        Add to Cart
      </button>
      <% } else { %>
      <span class="out-of-stock">Out of Stock</span>
      <% } %>
    </li>
    <% }); %>
</ul>
<% } else { %>
<p>No products found</p>
<% } %>

<!-- Switch-case equivalent -->
<% switch(user.role) { case 'admin': %>
<span class="badge admin">Admin</span>
<% break; case 'moderator': %>
<span class="badge mod">Moderator</span>
<% break; default: %>
<span class="badge user">User</span>
<% } %>
```

Pug:

```
// Conditionals
if user && user.isAdmin
  .admin-panel
    h3 Admin Panel
    a(href="/admin/dashboard") Dashboard
else if user
  p You don't have admin privileges
else
  p Please #[a(href="/login") login] to access more features

// Loops
h2 Product List
if products && products.length > 0
  ul.product-list
    each product in products
      li
        h3= product.name
        p= product.description
        span.price= formatPrice(product.price)
        if product.inStock
          button.add-to-cart(data-id=product._id) Add to Cart
        else
          span.out-of-stock Out of Stock
```

```
else
  p No products found

// Switch-case equivalent
case user.role
  when 'admin'
    span.badge.admin Admin
  when 'moderator'
    span.badge.mod Moderator
  default
    span.badge.user User
```

Handlebars:

```
<!-- Conditionals -->
{{#if user}}
  {{#if user.isAdmin}}
    <div class='admin-panel'>
      <h3>Admin Panel</h3>
      <a href='/admin/dashboard'>Dashboard</a>
    </div>
  {{else}}
    <p>You don't have admin privileges</p>
  {{/if}}
{{else}}
  <p>Please <a href='/login'>login</a> to access more features</p>
{{/if}}

<!-- Loops -->
<h2>Product List</h2>
{{#if products.length}}
  <ul class='product-list'>
    {{#each products}}
      <li>
        <h3>{{name}}</h3>
        <p>{{description}}</p>
        <span class='price'>{{formatPrice price}}</span>
        {{#if inStock}}
          <button class='add-to-cart' data-id='{{_id}}'>Add to Cart</button>
        {{else}}
          <span class='out-of-stock'>Out of Stock</span>
        {{/if}}
      </li>
    {{/each}}
  </ul>
{{else}}
  <p>No products found</p>
{{/if}}

<!-- Switch-case equivalent (using custom helper) -->
{{#switch user.role}}
  {{#case 'admin'}}
```

```
      <span class='badge admin'>Admin</span>
    {{/case}}
    {{#case 'moderator'}}
      <span class='badge mod'>Moderator</span>
    {{/case}}
    {{#default}}
      <span class='badge user'>User</span>
    {{/default}}
  {{/switch}}
```

Working with Forms

EJS Form Example:

```
<form action="/users/register" method="POST">
  <div class="form-group">
    <label for="name">Name</label>
    <input
      type="text"
      id="name"
      name="name"
      value="<%= typeof user != 'undefined' ? user.name : '' %>"
      required
    >
  </div>

  <div class="form-group">
    <label for="email">Email</label>
    <input
      type="email"
      id="email"
      name="email"
      value="<%= typeof user != 'undefined' ? user.email : '' %>"
      required
    >
  </div>

  <div class="form-group">
    <label for="password">Password</label>
    <input
      type="password"
      id="password"
      name="password"
      required
    >
  </div>

  <div class="form-group">
    <label for="confirmPassword">Confirm Password</label>
    <input
      type="password"
```

```
        id="confirmPassword"
        name="confirmPassword"
        required
      >
    </div>

    <div class="form-group">
      <label>Interests</label>
      <div class="checkbox-group">
        <label>
          <input
            type="checkbox"
            name="interests"
            value="technology"
            <%= typeof interests != 'undefined' && interests.includes('technology')
? 'checked' : '' %>
          > Technology
        </label>
        <label>
          <input
            type="checkbox"
            name="interests"
            value="sports"
            <%= typeof interests != 'undefined' && interests.includes('sports') ?
'checked' : '' %>
          > Sports
        </label>
        <label>
          <input
            type="checkbox"
            name="interests"
            value="food"
            <%= typeof interests != 'undefined' && interests.includes('food') ?
'checked' : '' %>
          > Food
        </label>
      </div>
    </div>

    <div class="form-group">
      <label for="country">Country</label>
      <select id="country" name="country">
        <option value="">Select a country</option>
        <% ['USA', 'Canada', 'UK', 'Australia'].forEach(function(country) { %>
          <option
            value="<%= country %>"
            <%= typeof user != 'undefined' && user.country === country ? 'selected'
: '' %>
          ><%= country %></option>
        <% }) %>
      </select>
    </div>

    <% if (typeof errors != 'undefined') { %>
```

```
    <div class="alert alert-danger">
      <ul>
        <% errors.forEach(function(error) { %>
          <li><%= error.msg %></li>
        <% }) %>
      </ul>
    </div>
  <% } %>

  <button type="submit" class="btn btn-primary">Register</button>
</form>
```

Handling Form Submission in Express:

```
const express = require('express');
const { check, validationResult } = require('express-validator');
const router = express.Router();

// Display registration form
router.get('/register', (req, res) => {
  res.render('register', {
    title: 'Register Account',
  });
});

// Process registration form
router.post(
  '/register',
  [
    // Validation
    check('name', 'Name is required').not().isEmpty(),
    check('email', 'Please include a valid email').isEmail(),
    check(
      'password',
      'Please enter a password with 6 or more characters'
    ).isLength({ min: 6 }),
    check('confirmPassword').custom((value, { req }) => {
      if (value !== req.body.password) {
        throw new Error('Password confirmation does not match password');
      }
      return true;
    }),
  ],
  async (req, res) => {
    // Check for validation errors
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
      // If there are errors, re-render the form with error messages
      return res.render('register', {
        title: 'Register Account',
        errors: errors.array(),
```

```
          user: req.body,
          interests: req.body.interests || [],
        });
      }

      try {
        // Process valid form data
        const { name, email, password, country, interests } = req.body;

        // Check if user already exists
        const existingUser = await User.findOne({ email });

        if (existingUser) {
          return res.render('register', {
            title: 'Register Account',
            errors: [{ msg: 'User already exists' }],
            user: req.body,
            interests: req.body.interests || [],
          });
        }

        // Create new user
        const user = new User({
          name,
          email,
          password, // Should be hashed before saving
          country,
          interests: Array.isArray(interests)
            ? interests
            : [interests].filter(Boolean),
        });

        await user.save();

        // Redirect to login page
        req.flash('success', 'Registration successful! You can now log in.');
        res.redirect('/users/login');
      } catch (err) {
        console.error(err);
        res.render('register', {
          title: 'Register Account',
          errors: [{ msg: 'Server error' }],
          user: req.body,
          interests: req.body.interests || [],
        });
      }
    }
  );

  module.exports = router;
```

Custom Helpers and Functions in Templates

Custom Helpers in EJS

```
// app.js
app.locals.formatDate = function (date) {
  return new Date(date).toLocaleDateString();
};

app.locals.truncate = function (str, len) {
  if (str.length > len && str.length > 0) {
    let newStr = str.substring(0, len);
    newStr = newStr.substring(
      0,
      Math.min(newStr.length, newStr.lastIndexOf(' '))
    );
    return `${newStr}...`;
  }
  return str;
};

app.locals.stripTags = function (input) {
  return input.replace(/<(?:.|\n)*?>/gm, '');
};

app.locals.editIcon = function (storyUser, loggedUser, storyId) {
  if (storyUser._id.toString() === loggedUser._id.toString()) {
    return `<a href="/stories/edit/${storyId}" class="btn-floating"><i class="fas
fa-edit"></i></a>`;
  }
  return '';
};
```

Using the helpers in an EJS template:

```
<div class="card">
  <div class="card-content">
    <h3><%= story.title %></h3>
    <p><%= truncate(stripTags(story.body), 150) %></p>
    <div class="meta">
      <small>Posted on: <%= formatDate(story.createdAt) %></small>
    </div>
  </div>
  <div class="card-action"><%- editIcon(story.user, user, story._id) %></div>
</div>
```

Custom Helpers in Handlebars

```
// app.js
const hbs = exphbs.create({
  helpers: {
    formatDate: function (date, format) {
      return moment(date).format(format);
    },
    truncate: function (str, len) {
      if (str.length > len && str.length > 0) {
        let newStr = str.substring(0, len);
        newStr = newStr.substring(
          0,
          Math.min(newStr.length, newStr.lastIndexOf(' '))
        );
        return new Handlebars.SafeString(`${newStr}...`);
      }
      return str;
    },
    stripTags: function (input) {
      return input.replace(/<(?:.|\n)*?>/gm, '');
    },
    editIcon: function (storyUser, loggedUser, storyId, floating = true) {
      if (storyUser._id.toString() === loggedUser._id.toString()) {
        if (floating) {
          return new Handlebars.SafeString(
            `<a href="/stories/edit/${storyId}" class="btn-floating"><i class="fas
fa-edit"></i></a>`
          );
        } else {
          return new Handlebars.SafeString(
            `<a href="/stories/edit/${storyId}"><i class="fas fa-edit"></i></a>`
          );
        }
      }
      return '';
    },
    select: function (selected, options) {
      return options
        .fn(this)
        .replace(
          new RegExp(' value="' + selected + '"'),
          '$& selected="selected"'
        );
    },
    ifEquals: function (arg1, arg2, options) {
      return arg1 === arg2 ? options.fn(this) : options.inverse(this);
    },
    each_upto: function (ary, max, options) {
      if (!ary || ary.length === 0) return options.inverse(this);

      const result = [];
      for (let i = 0; i < Math.min(ary.length, max); ++i) {
        result.push(options.fn(ary[i]));
      }
```

```
      return result.join('');
    },
  },
  defaultLayout: 'main',
  layoutsDir: path.join(__dirname, 'views/layouts'),
  partialsDir: path.join(__dirname, 'views/partials'),
});
```

Using the helpers in a Handlebars template:

```
<div class="card">
  <div class="card-content">
    <h3>{{story.title}}</h3>
    <p>{{truncate (stripTags story.body) 150}}</p>
    <div class="meta">
      <small>Posted on: {{formatDate story.createdAt 'MMMM Do YYYY, h:mm:ss a'}}
</small>
    </div>
  </div>
  <div class="card-action">
    {{{editIcon story.user ../user story._id}}}
  </div>
</div>

<form action="/stories/edit/{{story._id}}" method="POST">
  <div class="form-group">
    <label for="status">Status</label>
    <select id="status" name="status">
      {{#select story.status}}
        <option value="public">Public</option>
        <option value="private">Private</option>
        <option value="unpublished">Unpublished</option>
      {{/select}}
    </select>
  </div>
</form>

{{#ifEquals user._id story.user._id}}
  <a href="/stories/edit/{{story._id}}" class="btn">Edit</a>
{{else}}
  <p>You cannot edit this story</p>
{{/ifEquals}}

<h3>Recent Stories</h3>
<div class="story-list">
  {{#each_upto stories 5}}
    <div class="story-item">
      <h4>{{title}}</h4>
      <p>{{truncate (stripTags body) 100}}</p>
    </div>
  {{else}}
    <p>No stories found</p>
```

```
  {{/each_upto}}
</div>
```

Comparison of Template Engines

Feature Comparison

Feature	EJS	Pug	Handlebars
Syntax	HTML with `<% %>` tags	Indentation-based, no closing tags	HTML-like with `{{}}` tags
Learning Curve	Easy (similar to HTML)	Steeper (different from HTML)	Easy (similar to HTML)
Logic in Templates	Full JavaScript	Full JavaScript	Limited to helpers
Layouts	Through includes	Built-in extends/block	Built-in layouts
Partials/Includes	`<%- include() %>`	`include file`	`{{> partial}}`
Conditionals	`<% if() { %>`	`if condition`	`{{#if}}`
Loops	`<% for() { %>`	`each item in items`	`{{#each}}`
Performance	Fast	Very fast	Fast
Output Escaping	`<%= %>` (escaped), `<%- %>` (unescaped)	Auto-escaped by default	Auto-escaped by default
Debugging	Detailed errors	Detailed errors	Less detailed errors

Choosing the Right Template Engine

1. **Choose EJS if:**
   - You prefer to stay close to HTML
   - You need to use complex JavaScript logic in templates
   - Your team is already familiar with JavaScript
   - You want a minimal learning curve
2. **Choose Pug if:**
   - You prefer concise syntax
   - You value faster typing (no closing tags)
   - You like white-space significant code
   - You're building a large application with many templates
   - Performance is a critical concern
3. **Choose Handlebars if:**
   - You want to enforce separation between logic and presentation

    - You need to share templates between server and client
    - You prefer a more restrictive template syntax
    - You need strict HTML/logic separation for a large team

When to Use Template Engines vs. Client-Side Frameworks

Use Template Engines When:

- Building traditional web applications with multiple pages
- SEO is a primary concern (server-side rendering)
- You need a simpler tech stack without a complex build process
- Building applications with limited client-side interactivity
- Working on projects where initial page load speed is critical
- You need to support clients with JavaScript disabled

Use Client-Side Frameworks (React, Vue, Angular) When:

- Building single-page applications (SPAs)
- Creating highly interactive user interfaces
- Real-time updates are required
- Complex state management is needed
- You need a component-based architecture
- Working with complex forms and validations
- Building progressive web applications (PWAs)

Hybrid Approaches:

- Server-side rendering with client-side hydration
- Using template engines for initial render, then enhancing with JavaScript
- Micro-frontends with different tech stacks

Full Stack Integration

Project Architecture Examples

Traditional MVC Architecture

```
project/
├── config/              # Configuration files
│   ├── database.js      # Database configuration
│   ├── passport.js      # Authentication configuration
│   └── config.js        # General app configuration
├── controllers/         # Route handlers
│   ├── authController.js
│   ├── userController.js
│   └── productController.js
├── middleware/          # Custom middleware
│   ├── auth.js          # Authentication middleware
│   ├── error.js         # Error handling middleware
│   └── upload.js        # File upload middleware
```

```
├── models/              # Mongoose models
│   ├── User.js
│   ├── Product.js
│   └── Order.js
├── public/              # Static assets
│   ├── css/
│   ├── js/
│   └── images/
├── routes/              # Route definitions
│   ├── auth.js
│   ├── users.js
│   └── products.js
├── utils/               # Helper functions
│   ├── validators.js
│   ├── formatters.js
│   └── logger.js
├── views/               # Template files
│   ├── layouts/
│   ├── partials/
│   └── pages/
├── .env                 # Environment variables
├── .gitignore
├── app.js               # Application entry point
├── package.json
└── README.md
```

Sample code for MVC architecture:

```
// models/User.js - The Model
const mongoose = require('mongoose');
const bcrypt = require('bcrypt');

const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true,
    unique: true,
    lowercase: true
  },
  password: {
    type: String,
    required: true,
    minlength: 6
  },
  // More fields...
}, { timestamps: true });

// Pre-save hook to hash password
```

```
userSchema.pre('save', async function(next) {
  if (this.isModified('password')) {
    this.password = await bcrypt.hash(this.password, 10);
  }
  next();
});

// Method to compare passwords
userSchema.methods.comparePassword = async function(candidatePassword) {
  return bcrypt.compare(candidatePassword, this.password);
};

module.exports = mongoose.model('User', userSchema);

// controllers/userController.js - The Controller
const User = require('../models/User');

// Get user profile
exports.getProfile = async (req, res) => {
  try {
    const user = await User.findById(req.user.id).select('-password');
    res.render('profile', { user });
  } catch (err) {
    console.error(err);
    res.status(500).render('error', { message: 'Server error' });
  }
};

// Update user profile
exports.updateProfile = async (req, res) => {
  try {
    const { name, email } = req.body;

    // Update user
    const user = await User.findByIdAndUpdate(
      req.user.id,
      { $set: { name, email } },
      { new: true }
    ).select('-password');

    res.render('profile', {
      user,
      message: 'Profile updated successfully'
    });
  } catch (err) {
    console.error(err);
    res.status(500).render('error', { message: 'Server error' });
  }
};

// routes/users.js - Routes
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');
```

```
const auth = require('../middleware/auth');

// Get user profile - protected route
router.get('/profile', auth, userController.getProfile);

// Update user profile - protected route
router.post('/profile', auth, userController.updateProfile);

module.exports = router;

// views/profile.ejs - The View
<%- include('../partials/header') %>

<div class="container">
  <h1>User Profile</h1>

  <% if (typeof message !== 'undefined') { %>
    <div class="alert alert-success">
      <%= message %>
    </div>
  <% } %>

  <div class="profile-info">
    <p><strong>Name:</strong> <%= user.name %></p>
    <p><strong>Email:</strong> <%= user.email %></p>
    <p><strong>Member since:</strong> <%= new
Date(user.createdAt).toLocaleDateString() %></p>
  </div>

  <form action="/users/profile" method="POST">
    <div class="form-group">
      <label for="name">Name</label>
      <input type="text" id="name" name="name" value="<%= user.name %>" required>
    </div>

    <div class="form-group">
      <label for="email">Email</label>
      <input type="email" id="email" name="email" value="<%= user.email %>"
required>
    </div>

    <button type="submit" class="btn btn-primary">Update Profile</button>
  </form>
</div>

<%- include('../partials/footer') %>
```

Layered Architecture

```
project/
├── src/
```

```
│   ├── config/               # Configuration
│   │   └── index.js          # Central config
│   ├── data/                 # Data layer
│   │   ├── models/           # Mongoose models
│   │   ├── repositories/     # Data access
│   │   └── db.js             # Database connection
│   ├── domain/               # Business logic
│   │   ├── services/         # Business services
│   │   ├── validators/       # Domain validators
│   │   └── events/           # Domain events
│   ├── presentation/         # Presentation layer
│   │   ├── controllers/      # API controllers
│   │   ├── routes/           # Route definitions
│   │   ├── views/            # Templates
│   │   └── middleware/       # HTTP middleware
│   ├── utils/                # Shared utilities
│   │   ├── logger.js
│   │   └── helpers.js
│   └── app.js                # App entry point
├── public/                   # Static assets
├── tests/                    # Tests
│   ├── unit/
│   ├── integration/
│   └── e2e/
├── .env
├── package.json
└── README.md
```

Sample code for Layered Architecture:

```
// data/models/User.js - Data Layer (Model)
const mongoose = require('mongoose');
const bcrypt = require('bcrypt');

const userSchema = new mongoose.Schema(
  {
    name: String,
    email: String,
    password: String,
  },
  { timestamps: true }
);

// Hash password
userSchema.pre('save', async function (next) {
  // ... password hashing logic
  next();
});

module.exports = mongoose.model('User', userSchema);

// data/repositories/userRepository.js - Data Layer (Repository)
```

```
const User = require('../models/User');

class UserRepository {
  async findById(id) {
    return User.findById(id).select('-password');
  }

  async findByEmail(email) {
    return User.findOne({ email });
  }

  async create(userData) {
    const user = new User(userData);
    return user.save();
  }

  async update(id, userData) {
    return User.findByIdAndUpdate(id, userData, { new: true }).select(
      '-password'
    );
  }
}

module.exports = new UserRepository();

// domain/services/userService.js - Domain Layer (Business Logic)
const userRepository = require('../../data/repositories/userRepository');
const bcrypt = require('bcrypt');
const { ValidationError } = require('../../utils/errors');

class UserService {
  async getUserProfile(userId) {
    return userRepository.findById(userId);
  }

  async updateUserProfile(userId, userData) {
    // Validate user input
    if (!userData.name || !userData.email) {
      throw new ValidationError('Name and email are required');
    }

    // Check for existing email
    const existingUser = await userRepository.findByEmail(userData.email);
    if (existingUser && existingUser._id.toString() !== userId) {
      throw new ValidationError('Email already in use');
    }

    // Update user
    return userRepository.update(userId, userData);
  }

  async changePassword(userId, currentPassword, newPassword) {
    // Get user with password
    const user = await User.findById(userId);
```

```
    // Verify current password
    const isMatch = await bcrypt.compare(currentPassword, user.password);
    if (!isMatch) {
      throw new ValidationError('Current password is incorrect');
    }

    // Update password
    user.password = newPassword;
    await user.save();

    return { success: true };
  }
}

module.exports = new UserService();

// presentation/controllers/userController.js - Presentation Layer
const userService = require('../../domain/services/userService');

class UserController {
  async getProfile(req, res, next) {
    try {
      const user = await userService.getUserProfile(req.user.id);
      res.render('profile', { user });
    } catch (err) {
      next(err);
    }
  }

  async updateProfile(req, res, next) {
    try {
      const { name, email } = req.body;
      const user = await userService.updateUserProfile(req.user.id, {
        name,
        email,
      });

      res.render('profile', {
        user,
        message: 'Profile updated successfully',
      });
    } catch (err) {
      next(err);
    }
  }

  async changePassword(req, res, next) {
    try {
      const { currentPassword, newPassword, confirmPassword } = req.body;

      // Confirm passwords match
      if (newPassword !== confirmPassword) {
        return res.render('change-password', {
```

```
          error: 'New passwords do not match',
        });
      }

      await userService.changePassword(
        req.user.id,
        currentPassword,
        newPassword
      );

      res.render('change-password', {
        message: 'Password changed successfully',
      });
    } catch (err) {
      next(err);
    }
  }
}

module.exports = new UserController();

// presentation/routes/users.js - Presentation Layer (Routes)
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');
const auth = require('../middleware/auth');

router.get('/profile', auth, userController.getProfile.bind(userController));
router.post(
  '/profile',
  auth,
  userController.updateProfile.bind(userController)
);
router.get('/change-password', auth, (req, res) =>
  res.render('change-password')
);
router.post(
  '/change-password',
  auth,
  userController.changePassword.bind(userController)
);

module.exports = router;
```

API-Based Architecture

```
project/
├── backend/                    # Node.js API backend
│   ├── src/
│   │   ├── config/             # Configuration
│   │   ├── controllers/        # API controllers
```

```
│   │   ├── middleware/        # Middleware
│   │   ├── models/            # Mongoose models
│   │   ├── routes/            # API routes
│   │   ├── services/          # Business logic
│   │   ├── utils/             # Utility functions
│   │   └── app.js             # Express app setup
│   ├── tests/                 # Backend tests
│   ├── .env
│   └── package.json
├── frontend/                  # Frontend application
│   ├── public/                # Static assets
│   ├── src/
│   │   ├── assets/            # Images, styles, etc.
│   │   ├── components/        # React components
│   │   ├── context/           # React context
│   │   ├── hooks/             # Custom hooks
│   │   ├── pages/             # Page components
│   │   ├── services/          # API services
│   │   ├── utils/             # Utility functions
│   │   ├── App.js             # Main React component
│   │   └── index.js           # Entry point
│   ├── .env
│   └── package.json
├── .gitignore
└── README.md
```

Sample code for API-Based Architecture:

```
// backend/src/controllers/userController.js
const User = require('../models/User');
const { validationResult } = require('express-validator');

// @route   GET /api/users/me
// @desc    Get current user profile
// @access  Private
exports.getCurrentUser = async (req, res) => {
  try {
    const user = await User.findById(req.user.id).select('-password');
    res.json(user);
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: 'Server error' });
  }
};

// @route   PUT /api/users/me
// @desc    Update user profile
// @access  Private
exports.updateProfile = async (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
```

```
  }

  const { name, email } = req.body;

  try {
    let user = await User.findById(req.user.id);

    // Check if email is already in use by another user
    if (email !== user.email) {
      const existingUser = await User.findOne({ email });
      if (existingUser) {
        return res.status(400).json({ message: 'Email already in use' });
      }
    }

    // Update user
    user = await User.findByIdAndUpdate(
      req.user.id,
      { $set: { name, email } },
      { new: true }
    ).select('-password');

    res.json(user);
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: 'Server error' });
  }
};

// backend/src/routes/userRoutes.js
const express = require('express');
const router = express.Router();
const userController = require('../controllers/userController');
const auth = require('../middleware/auth');
const { check } = require('express-validator');

// Get current user
router.get('/me', auth, userController.getCurrentUser);

// Update user profile
router.put(
  '/me',
  [
    auth,
    [
      check('name', 'Name is required').not().isEmpty(),
      check('email', 'Please include a valid email').isEmail(),
    ],
  ],
  userController.updateProfile
);

module.exports = router;
```

```
// frontend/src/services/userService.js
import axios from 'axios';

const API_URL = '/api/users';

// Get token from local storage
const getToken = () => localStorage.getItem('token');

// Set auth token in headers
const setAuthHeader = () => {
  const token = getToken();
  if (token) {
    axios.defaults.headers.common['x-auth-token'] = token;
  } else {
    delete axios.defaults.headers.common['x-auth-token'];
  }
};

// Get current user profile
export const getCurrentUser = async () => {
  setAuthHeader();
  const response = await axios.get(`${API_URL}/me`);
  return response.data;
};

// Update user profile
export const updateProfile = async (userData) => {
  setAuthHeader();
  const response = await axios.put(`${API_URL}/me`, userData);
  return response.data;
};

// frontend/src/pages/ProfilePage.js (continued)
import React, { useState, useEffect } from 'react';
import { getCurrentUser, updateProfile } from '../services/userService';

const ProfilePage = () => {
  const [user, setUser] = useState(null);
  const [formData, setFormData] = useState({
    name: '',
    email: '',
  });
  const [message, setMessage] = useState('');
  const [error, setError] = useState('');
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const fetchUser = async () => {
      try {
        const userData = await getCurrentUser();
        setUser(userData);
        setFormData({
          name: userData.name,
          email: userData.email,
```

```
        });
        setLoading(false);
      } catch (err) {
        setError('Failed to load user profile');
        setLoading(false);
      }
    };

    fetchUser();
  }, []);

  const handleChange = (e) => {
    setFormData({
      ...formData,
      [e.target.name]: e.target.value,
    });
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    setMessage('');
    setError('');

    try {
      const updatedUser = await updateProfile(formData);
      setUser(updatedUser);
      setMessage('Profile updated successfully');
    } catch (err) {
      setError(err.response?.data?.message || 'Error updating profile');
    }
  };

  if (loading) {
    return <div>Loading...</div>;
  }

  return (
    <div className="profile-container">
      <h1>User Profile</h1>

      {message && <div className="alert alert-success">{message}</div>}
      {error && <div className="alert alert-danger">{error}</div>}

      <div className="profile-info">
        <p>
          <strong>Member since:</strong>{' '}
          {new Date(user.createdAt).toLocaleDateString()}
        </p>
      </div>

      <form onSubmit={handleSubmit}>
        <div className="form-group">
          <label htmlFor="name">Name</label>
          <input
```

```
            type="text"
            id="name"
            name="name"
            value={formData.name}
            onChange={handleChange}
            required
          />
        </div>

        <div className="form-group">
          <label htmlFor="email">Email</label>
          <input
            type="email"
            id="email"
            name="email"
            value={formData.email}
            onChange={handleChange}
            required
          />
        </div>

        <button type="submit" className="btn btn-primary">
          Update Profile
        </button>
      </form>
    </div>
  );
};

export default ProfilePage;
```

Authentication Flows

Traditional Server-Side Authentication with Sessions

```
// backend/src/controllers/authController.js
const User = require('../models/User');
const bcrypt = require('bcrypt');

// Login controller
exports.login = async (req, res) => {
  const { email, password } = req.body;

  try {
    // Check if user exists
    const user = await User.findOne({ email });
    if (!user) {
      return res.status(401).render('login', {
        error: 'Invalid credentials',
      });
    }
```

```
    // Check password
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) {
      return res.status(401).render('login', {
        error: 'Invalid credentials',
      });
    }

    // Set user in session
    req.session.user = {
      id: user._id,
      name: user.name,
      email: user.email,
    };

    // Redirect to dashboard
    res.redirect('/dashboard');
  } catch (err) {
    console.error(err);
    res.status(500).render('login', {
      error: 'Server error',
    });
  }
};

// Logout controller
exports.logout = (req, res) => {
  req.session.destroy((err) => {
    if (err) {
      return res.status(500).render('error', {
        message: 'Error logging out',
      });
    }
    res.clearCookie('connect.sid');
    res.redirect('/login');
  });
};

// app.js
const express = require('express');
const session = require('express-session');
const MongoStore = require('connect-mongo');
const app = express();

// Session configuration
app.use(
  session({
    secret: process.env.SESSION_SECRET || 'your-secret-key',
    resave: false,
    saveUninitialized: false,
    store: MongoStore.create({
      mongoUrl: process.env.MONGODB_URI,
      collectionName: 'sessions',
```

```
    }),
    cookie: {
      maxAge: 1000 * 60 * 60 * 24, // 1 day
      httpOnly: true,
      secure: process.env.NODE_ENV === 'production', // Use secure cookies in
production
      sameSite: 'strict',
    },
  })
);

// Authentication middleware
function isAuthenticated(req, res, next) {
  if (req.session.user) {
    // Make user data available to all templates
    res.locals.user = req.session.user;
    return next();
  }
  res.redirect('/login');
}

// Routes
app.get('/dashboard', isAuthenticated, (req, res) => {
  res.render('dashboard');
});
```

JWT Authentication for API-Based Applications

```
// backend/src/controllers/authController.js
const User = require('../models/User');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const { validationResult } = require('express-validator');

// @route   POST /api/auth/register
// @desc    Register a user
// @access  Public
exports.register = async (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  const { name, email, password } = req.body;

  try {
    // Check if user already exists
    let user = await User.findOne({ email });
    if (user) {
      return res.status(400).json({ message: 'User already exists' });
    }
```

```
    // Create new user
    user = new User({
      name,
      email,
      password,
    });

    // Password is hashed in the pre-save middleware

    await user.save();

    // Create JWT payload
    const payload = {
      user: {
        id: user.id,
      },
    };

    // Sign token
    jwt.sign(
      payload,
      process.env.JWT_SECRET,
      { expiresIn: '24h' },
      (err, token) => {
        if (err) throw err;
        res.json({ token });
      }
    );
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: 'Server error' });
  }
};

// @route   POST /api/auth/login
// @desc    Authenticate user & get token
// @access  Public
exports.login = async (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  const { email, password } = req.body;

  try {
    // Check if user exists
    let user = await User.findOne({ email });
    if (!user) {
      return res.status(401).json({ message: 'Invalid credentials' });
    }

    // Check password
```

```
      const isMatch = await bcrypt.compare(password, user.password);
      if (!isMatch) {
        return res.status(401).json({ message: 'Invalid credentials' });
      }

      // Create JWT payload
      const payload = {
        user: {
          id: user.id,
        },
      };

      // Sign token
      jwt.sign(
        payload,
        process.env.JWT_SECRET,
        { expiresIn: '24h' },
        (err, token) => {
          if (err) throw err;
          res.json({ token });
        }
      );
    } catch (err) {
      console.error(err);
      res.status(500).json({ message: 'Server error' });
    }
  };

  // @route   GET /api/auth/me
  // @desc    Get current user
  // @access  Private
  exports.getCurrentUser = async (req, res) => {
    try {
      const user = await User.findById(req.user.id).select('-password');
      res.json(user);
    } catch (err) {
      console.error(err);
      res.status(500).json({ message: 'Server error' });
    }
  };

  // backend/src/middleware/auth.js
  const jwt = require('jsonwebtoken');

  module.exports = (req, res, next) => {
    // Get token from header
    const token = req.header('x-auth-token');

    // Check if no token
    if (!token) {
      return res.status(401).json({ message: 'No token, authorization denied' });
    }

    try {
```

```
    // Verify token
    const decoded = jwt.verify(token, process.env.JWT_SECRET);

    // Add user from payload
    req.user = decoded.user;
    next();
  } catch (err) {
    res.status(401).json({ message: 'Token is not valid' });
  }
};

// frontend/src/context/AuthContext.js
import React, { createContext, useReducer, useEffect } from 'react';
import axios from 'axios';
import authReducer from './authReducer';
import setAuthToken from '../utils/setAuthToken';

// Initial state
const initialState = {
  token: localStorage.getItem('token'),
  isAuthenticated: null,
  loading: true,
  user: null,
  error: null,
};

// Create context
export const AuthContext = createContext(initialState);

// Provider component
export const AuthProvider = ({ children }) => {
  const [state, dispatch] = useReducer(authReducer, initialState);

  // Load user
  useEffect(() => {
    const loadUser = async () => {
      if (localStorage.token) {
        setAuthToken(localStorage.token);
      }

      try {
        const res = await axios.get('/api/auth/me');

        dispatch({
          type: 'USER_LOADED',
          payload: res.data,
        });
      } catch (err) {
        dispatch({ type: 'AUTH_ERROR' });
      }
    };

    loadUser();
  }, []);
```

```
// Register user
const register = async (formData) => {
  const config = {
    headers: {
      'Content-Type': 'application/json',
    },
  };

  try {
    const res = await axios.post('/api/auth/register', formData, config);

    dispatch({
      type: 'REGISTER_SUCCESS',
      payload: res.data,
    });

    loadUser();
  } catch (err) {
    dispatch({
      type: 'REGISTER_FAIL',
      payload: err.response.data.message,
    });
  }
};

// Login user
const login = async (formData) => {
  const config = {
    headers: {
      'Content-Type': 'application/json',
    },
  };

  try {
    const res = await axios.post('/api/auth/login', formData, config);

    dispatch({
      type: 'LOGIN_SUCCESS',
      payload: res.data,
    });

    loadUser();
  } catch (err) {
    dispatch({
      type: 'LOGIN_FAIL',
      payload: err.response.data.message,
    });
  }
};

// Logout
const logout = () => dispatch({ type: 'LOGOUT' });
```

```
  return (
    <AuthContext.Provider
      value={{
        token: state.token,
        isAuthenticated: state.isAuthenticated,
        loading: state.loading,
        user: state.user,
        error: state.error,
        register,
        login,
        logout,
      }}
    >
      {children}
    </AuthContext.Provider>
  );
};

// frontend/src/components/auth/Login.js
import React, { useState, useContext, useEffect } from 'react';
import { AuthContext } from '../../context/AuthContext';
import { useNavigate } from 'react-router-dom';

const Login = () => {
  const [formData, setFormData] = useState({
    email: '',
    password: '',
  });
  const { email, password } = formData;
  const { login, isAuthenticated, error } = useContext(AuthContext);
  const navigate = useNavigate();

  useEffect(() => {
    // Redirect if authenticated
    if (isAuthenticated) {
      navigate('/dashboard');
    }
  }, [isAuthenticated, navigate]);

  const onChange = (e) => {
    setFormData({ ...formData, [e.target.name]: e.target.value });
  };

  const onSubmit = (e) => {
    e.preventDefault();
    login({ email, password });
  };

  return (
    <div className="login-container">
      <h1>Login</h1>
      {error && <div className="alert alert-danger">{error}</div>}
      <form onSubmit={onSubmit}>
        <div className="form-group">
```

```
            <label htmlFor="email">Email</label>
            <input
              type="email"
              id="email"
              name="email"
              value={email}
              onChange={onChange}
              required
            />
          </div>
          <div className="form-group">
            <label htmlFor="password">Password</label>
            <input
              type="password"
              id="password"
              name="password"
              value={password}
              onChange={onChange}
              required
            />
          </div>
          <button type="submit" className="btn btn-primary">
            Login
          </button>
        </form>
      </div>
    );
};

export default Login;
```

OAuth Authentication with Passport.js

```
// backend/src/config/passport.js
const passport = require('passport');
const GoogleStrategy = require('passport-google-oauth20').Strategy;
const User = require('../models/User');

passport.serializeUser((user, done) => {
  done(null, user.id);
});

passport.deserializeUser(async (id, done) => {
  try {
    const user = await User.findById(id);
    done(null, user);
  } catch (err) {
    done(err, null);
  }
});
```

```
passport.use(
  new GoogleStrategy(
    {
      clientID: process.env.GOOGLE_CLIENT_ID,
      clientSecret: process.env.GOOGLE_CLIENT_SECRET,
      callbackURL: '/auth/google/callback',
      proxy: true,
    },
    async (accessToken, refreshToken, profile, done) => {
      try {
        // Check if user already exists
        let user = await User.findOne({ 'google.id': profile.id });

        if (user) {
          return done(null, user);
        }

        // Create new user
        user = new User({
          name: profile.displayName,
          email: profile.emails[0].value,
          google: {
            id: profile.id,
            email: profile.emails[0].value,
            name: profile.displayName,
          },
          avatar: profile.photos[0].value,
        });

        await user.save();
        done(null, user);
      } catch (err) {
        done(err, null);
      }
    }
  )
);

// backend/src/routes/authRoutes.js
const express = require('express');
const passport = require('passport');
const router = express.Router();

// @route   GET /auth/google
// @desc    Auth with Google
// @access  Public
router.get(
  '/google',
  passport.authenticate('google', { scope: ['profile', 'email'] })
);

// @route   GET /auth/google/callback
// @desc    Google auth callback
// @access  Public
```

```
router.get(
  '/google/callback',
  passport.authenticate('google', { failureRedirect: '/login' }),
  (req, res) => {
    res.redirect('/dashboard');
  }
);

// @route   GET /auth/logout
// @desc    Logout user
// @access  Private
router.get('/logout', (req, res) => {
  req.logout();
  res.redirect('/');
});

module.exports = router;

// app.js
const express = require('express');
const passport = require('passport');
const session = require('express-session');
const MongoStore = require('connect-mongo');
const app = express();

// Passport config
require('./config/passport');

// Session middleware
app.use(
  session({
    secret: process.env.SESSION_SECRET,
    resave: false,
    saveUninitialized: false,
    store: MongoStore.create({
      mongoUrl: process.env.MONGODB_URI,
      collectionName: 'sessions',
    }),
  })
);

// Passport middleware
app.use(passport.initialize());
app.use(passport.session());

// Set global variables
app.use((req, res, next) => {
  res.locals.user = req.user || null;
  next();
});

// Routes
app.use('/auth', require('./routes/authRoutes'));
```

Form Handling and Validation

Server-Side Validation with Express Validator

```
// backend/src/routes/userRoutes.js
const express = require('express');
const router = express.Router();
const { check, validationResult } = require('express-validator');
const userController = require('../controllers/userController');
const auth = require('../middleware/auth');

// @route   POST /api/users
// @desc    Create a new user
// @access  Public
router.post(
  '/',
  [
    // Validation rules
    check('name', 'Name is required').not().isEmpty(),
    check('email', 'Please include a valid email').isEmail(),
    check(
      'password',
      'Please enter a password with 6 or more characters'
    ).isLength({ min: 6 }),
    check('confirmPassword').custom((value, { req }) => {
      if (value !== req.body.password) {
        throw new Error('Password confirmation does not match password');
      }
      return true;
    }),
  ],
  async (req, res) => {
    // Check for validation errors
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    // Continue with controller logic
    userController.createUser(req, res);
  }
);

// backend/src/controllers/userController.js
const User = require('../models/User');

exports.createUser = async (req, res) => {
  const { name, email, password } = req.body;

  try {
    // Check if user already exists
```

```
    let user = await User.findOne({ email });
    if (user) {
      return res.status(400).json({ message: 'User already exists' });
    }

    // Create new user
    user = new User({
      name,
      email,
      password,
    });

    await user.save();
    res.status(201).json({ message: 'User created successfully' });
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: 'Server error' });
  }
};
```

Client-Side Form Validation with React Hook Form

```
// frontend/src/components/RegisterForm.js
import React, { useState } from 'react';
import { useForm } from 'react-hook-form';
import { yupResolver } from '@hookform/resolvers/yup';
import * as yup from 'yup';
import axios from 'axios';

// Validation schema
const schema = yup.object().shape({
  name: yup.string().required('Name is required'),
  email: yup
    .string()
    .email('Invalid email format')
    .required('Email is required'),
  password: yup
    .string()
    .min(6, 'Password must be at least 6 characters')
    .required('Password is required'),
  confirmPassword: yup
    .string()
    .oneOf([yup.ref('password'), null], 'Passwords must match')
    .required('Confirm password is required'),
});

const RegisterForm = () => {
  const [isSubmitting, setIsSubmitting] = useState(false);
  const [serverError, setServerError] = useState('');
  const [success, setSuccess] = useState('');
```

```
  const {
    register,
    handleSubmit,
    formState: { errors },
    reset,
  } = useForm({
    resolver: yupResolver(schema),
  });

  const onSubmit = async (data) => {
    setIsSubmitting(true);
    setServerError('');
    setSuccess('');

    try {
      await axios.post('/api/users', data);
      setSuccess('Registration successful! You can now log in.');
      reset(); // Reset form
    } catch (err) {
      setServerError(err.response?.data?.message || 'Error registering user');
    } finally {
      setIsSubmitting(false);
    }
  };

  return (
    <div className="register-form">
      <h2>Create an Account</h2>

      {serverError && <div className="alert alert-danger">{serverError}</div>}
      {success && <div className="alert alert-success">{success}</div>}

      <form onSubmit={handleSubmit(onSubmit)}>
        <div className="form-group">
          <label htmlFor="name">Name</label>
          <input
            type="text"
            id="name"
            className={`form-control ${errors.name ? 'is-invalid' : ''}`}
            {...register('name')}
          />
          {errors.name && (
            <div className="invalid-feedback">{errors.name.message}</div>
          )}
        </div>

        <div className="form-group">
          <label htmlFor="email">Email</label>
          <input
            type="email"
            id="email"
            className={`form-control ${errors.email ? 'is-invalid' : ''}`}
            {...register('email')}
          />
```

```
            {errors.email && (
              <div className="invalid-feedback">{errors.email.message}</div>
            )}
          </div>

          <div className="form-group">
            <label htmlFor="password">Password</label>
            <input
              type="password"
              id="password"
              className={`form-control ${errors.password ? 'is-invalid' : ''}`}
              {...register('password')}
            />
            {errors.password && (
              <div className="invalid-feedback">{errors.password.message}</div>
            )}
          </div>

          <div className="form-group">
            <label htmlFor="confirmPassword">Confirm Password</label>
            <input
              type="password"
              id="confirmPassword"
              className={`form-control ${
                errors.confirmPassword ? 'is-invalid' : ''
              }`}
              {...register('confirmPassword')}
            />
            {errors.confirmPassword && (
              <div className="invalid-feedback">
                {errors.confirmPassword.message}
              </div>
            )}
          </div>

          <button
            type="submit"
            className="btn btn-primary"
            disabled={isSubmitting}
          >
            {isSubmitting ? 'Registering...' : 'Register'}
          </button>
        </form>
      </div>
    );
  };

  export default RegisterForm;
```

File Uploads and Management

Server-Side File Upload with Multer

```
// backend/src/middleware/upload.js
const multer = require('multer');
const path = require('path');
const fs = require('fs');

// Ensure upload directory exists
const uploadDir = path.join(__dirname, '../../uploads');
if (!fs.existsSync(uploadDir)) {
  fs.mkdirSync(uploadDir, { recursive: true });
}

// Configure storage
const storage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, uploadDir);
  },
  filename: (req, file, cb) => {
    // Generate unique filename
    const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1e9);
    const ext = path.extname(file.originalname);
    cb(null, `${file.fieldname}-${uniqueSuffix}${ext}`);
  },
});

// File filter
const fileFilter = (req, file, cb) => {
  // Accept images and PDFs
  const allowedMimes = [
    'image/jpeg',
    'image/png',
    'image/gif',
    'application/pdf',
  ];

  if (allowedMimes.includes(file.mimetype)) {
    cb(null, true);
  } else {
    cb(
      new Error(
        'Invalid file type. Only JPEG, PNG, GIF and PDF files are allowed.'
      ),
      false
    );
  }
};

// Configure multer
const upload = multer({
  storage: storage,
  limits: {
    fileSize: 5 * 1024 * 1024, // 5MB
  },
```

```
  fileFilter: fileFilter,
});

module.exports = {
  single: upload.single('file'),
  multiple: upload.array('files', 5), // Max 5 files
  fields: upload.fields([
    { name: 'avatar', maxCount: 1 },
    { name: 'documents', maxCount: 3 },
  ]),
};

// backend/src/controllers/uploadController.js
const fs = require('fs');
const path = require('path');

exports.uploadSingle = (req, res) => {
  if (!req.file) {
    return res.status(400).json({ message: 'No file uploaded' });
  }

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

exports.uploadMultiple = (req, res) => {
  if (!req.files || req.files.length === 0) {
    return res.status(400).json({ message: 'No files uploaded' });
  }

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

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

exports.uploadFields = (req, res) => {
  if (!req.files) {
```

```
    return res.status(400).json({ message: 'No files uploaded' });
  }

  const response = {
    message: 'Files uploaded successfully',
    avatar: req.files.avatar ? req.files.avatar[0] : null,
    documents: req.files.documents || [],
  };

  res.json(response);
};

exports.deleteFile = (req, res) => {
  const { filename } = req.params;
  const filePath = path.join(__dirname, '../../uploads', filename);

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

  // Delete file
  fs.unlink(filePath, (err) => {
    if (err) {
      console.error(err);
      return res.status(500).json({ message: 'Error deleting file' });
    }

    res.json({ message: 'File deleted successfully' });
  });
};

// backend/src/routes/uploadRoutes.js
const express = require('express');
const router = express.Router();
const uploadController = require('../controllers/uploadController');
const upload = require('../middleware/upload');
const auth = require('../middleware/auth');

// @route   POST /api/upload/single
// @desc    Upload a single file
// @access  Private
router.post('/single', [auth, upload.single], uploadController.uploadSingle);

// @route   POST /api/upload/multiple
// @desc    Upload multiple files
// @access  Private
router.post(
  '/multiple',
  [auth, upload.multiple],
  uploadController.uploadMultiple
);

// @route   POST /api/upload/fields
```

```
// @desc    Upload files to specific fields
// @access  Private
router.post('/fields', [auth, upload.fields], uploadController.uploadFields);

// @route   DELETE /api/upload/:filename
// @desc    Delete a file
// @access  Private
router.delete('/:filename', auth, uploadController.deleteFile);

module.exports = router;

// app.js
app.use('/uploads', express.static(path.join(__dirname, '../uploads')));
```

Client-Side File Upload with React

```
// frontend/src/components/FileUpload.js
import React, { useState } from 'react';
import axios from 'axios';

const FileUpload = () => {
  const [file, setFile] = useState('');
  const [filename, setFilename] = useState('Choose File');
  const [uploadedFile, setUploadedFile] = useState(null);
  const [message, setMessage] = useState('');
  const [uploadPercentage, setUploadPercentage] = useState(0);
  const [error, setError] = useState('');

  const onChange = (e) => {
    setFile(e.target.files[0]);
    setFilename(e.target.files[0]?.name || 'Choose File');
  };

  const onSubmit = async (e) => {
    e.preventDefault();

    if (!file) {
      setError('Please select a file');
      return;
    }

    const formData = new FormData();
    formData.append('file', file);

    try {
      const res = await axios.post('/api/upload/single', formData, {
        headers: {
          'Content-Type': 'multipart/form-data',
          'x-auth-token': localStorage.getItem('token'),
        },
        onUploadProgress: (progressEvent) => {
```

```
            setUploadPercentage(
              parseInt(
                Math.round((progressEvent.loaded * 100) / progressEvent.total)
              )
            );
          },
        });

        // Clear percentage after upload completes
        setTimeout(() => setUploadPercentage(0), 3000);

        const { file: uploadedFile, message } = res.data;
        setUploadedFile(uploadedFile);
        setMessage(message);
        setError('');
      } catch (err) {
        setError(err.response?.data?.message || 'Error uploading file');
        setUploadPercentage(0);
      }
    };

    return (
      <div className="file-upload-container">
        <h2>File Upload</h2>

        {message && <div className="alert alert-success">{message}</div>}
        {error && <div className="alert alert-danger">{error}</div>}

        <form onSubmit={onSubmit}>
          <div className="custom-file mb-4">
            <input
              type="file"
              className="custom-file-input"
              id="fileInput"
              onChange={onChange}
            />
            <label className="custom-file-label" htmlFor="fileInput">
              {filename}
            </label>
          </div>

          {uploadPercentage > 0 && (
            <div className="progress mb-4">
              <div
                className="progress-bar"
                role="progressbar"
                style={{ width: `${uploadPercentage}%` }}
                aria-valuenow={uploadPercentage}
                aria-valuemin="0"
                aria-valuemax="100"
              >
                {uploadPercentage}%
              </div>
            </div>
```

```
          )}

          <button type="submit" className="btn btn-primary">
            Upload
          </button>
        </form>

        {uploadedFile && (
          <div className="mt-4">
            <h3>Uploaded File</h3>
            <div className="card">
              <div className="card-body">
                <h5 className="card-title">{uploadedFile.originalname}</h5>
                <p className="card-text">
                  Size: {(uploadedFile.size / 1024).toFixed(2)} KB
                </p>
                {uploadedFile.mimetype.startsWith('image/') ? (
                  <img
                    src={`/uploads/${uploadedFile.filename}`}
                    alt="Uploaded"
                    className="img-fluid"
                  />
                ) : (
                  <a
                    href={`/uploads/${uploadedFile.filename}`}
                    className="btn btn-secondary"
                    target="_blank"
                    rel="noopener noreferrer"
                  >
                    View File
                  </a>
                )}
              </div>
            </div>
          </div>
        )}
      </div>
    );
};

export default FileUpload;
```

File Upload to Cloud Storage (AWS S3)

```
// backend/src/middleware/s3upload.js
const multer = require('multer');
const {
  S3Client,
  PutObjectCommand,
  DeleteObjectCommand,
} = require('@aws-sdk/client-s3');
```

```
const { v4: uuidv4 } = require('uuid');
const path = require('path');

// Configure S3 client
const s3Client = new S3Client({
  region: process.env.AWS_REGION,
  credentials: {
    accessKeyId: process.env.AWS_ACCESS_KEY_ID,
    secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
  },
});

// Configure storage
const multerStorage = multer.memoryStorage();

// File filter
const fileFilter = (req, file, cb) => {
  const allowedMimes = [
    'image/jpeg',
    'image/png',
    'image/gif',
    'application/pdf',
  ];

  if (allowedMimes.includes(file.mimetype)) {
    cb(null, true);
  } else {
    cb(
      new Error(
        'Invalid file type. Only JPEG, PNG, GIF and PDF files are allowed.'
      ),
      false
    );
  }
};

// Configure multer
const upload = multer({
  storage: multerStorage,
  limits: {
    fileSize: 5 * 1024 * 1024, // 5MB
  },
  fileFilter: fileFilter,
});

// Upload file to S3
const uploadToS3 = async (file, folder = '') => {
  const fileExt = path.extname(file.originalname);
  const fileName = `${folder}/${uuidv4()}${fileExt}`;

  const params = {
    Bucket: process.env.AWS_S3_BUCKET_NAME,
    Key: fileName,
    Body: file.buffer,
```

```
    ContentType: file.mimetype,
    ACL: 'public-read', // Make file publicly accessible
  };

  await s3Client.send(new PutObjectCommand(params));

  return {
    key: fileName,
    url:
`https://${process.env.AWS_S3_BUCKET_NAME}.s3.${process.env.AWS_REGION}.amazonaws.
com/${fileName}`,
    originalname: file.originalname,
    mimetype: file.mimetype,
    size: file.size,
  };
};

// Delete file from S3
const deleteFromS3 = async (key) => {
  const params = {
    Bucket: process.env.AWS_S3_BUCKET_NAME,
    Key: key,
  };

  await s3Client.send(new DeleteObjectCommand(params));
};

module.exports = {
  upload,
  uploadToS3,
  deleteFromS3,
};

// backend/src/controllers/s3UploadController.js
const { uploadToS3, deleteFromS3 } = require('../middleware/s3upload');

exports.uploadSingle = async (req, res) => {
  if (!req.file) {
    return res.status(400).json({ message: 'No file uploaded' });
  }

  try {
    const folder = req.user ? `users/${req.user.id}` : 'uploads';
    const uploadedFile = await uploadToS3(req.file, folder);

    res.json({
      message: 'File uploaded successfully',
      file: uploadedFile,
    });
  } catch (err) {
    console.error('S3 upload error:', err);
    res.status(500).json({ message: 'Error uploading to S3' });
  }
};
```

```
exports.uploadMultiple = async (req, res) => {
  if (!req.files || req.files.length === 0) {
    return res.status(400).json({ message: 'No files uploaded' });
  }

  try {
    const folder = req.user ? `users/${req.user.id}` : 'uploads';
    const uploadPromises = req.files.map((file) => uploadToS3(file, folder));
    const uploadedFiles = await Promise.all(uploadPromises);

    res.json({
      message: `${uploadedFiles.length} files uploaded successfully`,
      files: uploadedFiles,
    });
  } catch (err) {
    console.error('S3 upload error:', err);
    res.status(500).json({ message: 'Error uploading to S3' });
  }
};

exports.deleteFile = async (req, res) => {
  const { key } = req.params;

  try {
    await deleteFromS3(key);
    res.json({ message: 'File deleted successfully' });
  } catch (err) {
    console.error('S3 delete error:', err);
    res.status(500).json({ message: 'Error deleting from S3' });
  }
};

// backend/src/routes/s3UploadRoutes.js
const express = require('express');
const router = express.Router();
const s3UploadController = require('../controllers/s3UploadController');
const { upload } = require('../middleware/s3upload');
const auth = require('../middleware/auth');

// @route   POST /api/s3/upload/single
// @desc    Upload a single file to S3
// @access  Private
router.post(
  '/upload/single',
  [auth, upload.single('file')],
  s3UploadController.uploadSingle
);

// @route   POST /api/s3/upload/multiple
// @desc    Upload multiple files to S3
// @access  Private
router.post(
  '/upload/multiple',
```

```
  [auth, upload.array('files', 5)],
  s3UploadController.uploadMultiple
);

// @route   DELETE /api/s3/delete/:key
// @desc    Delete a file from S3
// @access  Private
router.delete('/delete/:key', auth, s3UploadController.deleteFile);

module.exports = router;
```

Error Handling Across the Stack

Centralized Error Handling in Express

```
// backend/src/utils/AppError.js
class AppError extends Error {
  constructor(message, statusCode) {
    super(message);
    this.statusCode = statusCode;
    this.status = `${statusCode}`.startsWith('4') ? 'fail' : 'error';
    this.isOperational = true; // This is a known operational error, not a bug

    Error.captureStackTrace(this, this.constructor);
  }
}

module.exports = AppError;

// backend/src/middleware/errorHandler.js
const AppError = require('../utils/AppError');

// Development error handler (detailed error info)
const sendErrorDev = (err, req, res) => {
  // API error response
  if (req.originalUrl.startsWith('/api')) {
    return res.status(err.statusCode).json({
      status: err.status,
      error: err,
      message: err.message,
      stack: err.stack,
    });
  }

  // Rendered website error response
  console.error('ERROR 💥', err);
  return res.status(err.statusCode).render('error', {
    title: 'Error',
    msg: err.message,
    error: err,
    stack: err.stack,
```

```
  });
};

// Production error handler (limited error info)
const sendErrorProd = (err, req, res) => {
  // API error response
  if (req.originalUrl.startsWith('/api')) {
    // Operational, trusted error: send message to client
    if (err.isOperational) {
      return res.status(err.statusCode).json({
        status: err.status,
        message: err.message,
      });
    }

    // Programming or other unknown error: don't leak error details
    console.error('ERROR 💥', err);
    return res.status(500).json({
      status: 'error',
      message: 'Something went wrong',
    });
  }

  // Rendered website error response
  if (err.isOperational) {
    return res.status(err.statusCode).render('error', {
      title: 'Error',
      msg: err.message,
    });
  }

  // Programming or other unknown error: don't leak error details
  console.error('ERROR 💥', err);
  return res.status(err.statusCode).render('error', {
    title: 'Error',
    msg: 'Please try again later.',
  });
};

// Mongoose validation error handler
const handleValidationErrorDB = (err) => {
  const errors = Object.values(err.errors).map((el) => el.message);
  const message = `Invalid input data. ${errors.join('. ')}`;
  return new AppError(message, 400);
};

// Mongoose duplicate key error handler
const handleDuplicateFieldsDB = (err) => {
  const value = err.errmsg.match(/(["'])(\\?.)*?\1/)[0];
  const message = `Duplicate field value: ${value}. Please use another value!`;
  return new AppError(message, 400);
};

// Mongoose casting error handler
```

```
const handleCastErrorDB = (err) => {
  const message = `Invalid ${err.path}: ${err.value}`;
  return new AppError(message, 400);
};

// JWT error handlers
const handleJWTError = () =>
  new AppError('Invalid token. Please log in again.', 401);
const handleJWTExpiredError = () =>
  new AppError('Your token has expired. Please log in again.', 401);

// Main error handler middleware
module.exports = (err, req, res, next) => {
  err.statusCode = err.statusCode || 500;
  err.status = err.status || 'error';

  if (process.env.NODE_ENV === 'development') {
    sendErrorDev(err, req, res);
  } else if (process.env.NODE_ENV === 'production') {
    let error = { ...err };
    error.message = err.message;

    // Handle specific error types
    if (error.name === 'CastError') error = handleCastErrorDB(error);
    if (error.code === 11000) error = handleDuplicateFieldsDB(error);
    if (error.name === 'ValidationError')
      error = handleValidationErrorDB(error);
    if (error.name === 'JsonWebTokenError') error = handleJWTError();
    if (error.name === 'TokenExpiredError') error = handleJWTExpiredError();

    sendErrorProd(error, req, res);
  }
};

// Using the AppError in controllers
const AppError = require('../utils/AppError');

exports.getProduct = async (req, res, next) => {
  try {
    const product = await Product.findById(req.params.id);

    if (!product) {
      return next(new AppError('No product found with that ID', 404));
    }

    res.json({
      status: 'success',
      data: {
        product,
      },
    });
  } catch (err) {
    next(err);
  }
```

```
};

// app.js
const express = require('express');
const globalErrorHandler = require('./middleware/errorHandler');
const app = express();

// Routes
app.use('/api/users', require('./routes/userRoutes'));
// More routes...

// Handle undefined routes
app.all('*', (req, res, next) => {
  next(new AppError(`Can't find ${req.originalUrl} on this server!`, 404));
});

// Global error handler - must be last middleware
app.use(globalErrorHandler);
```

Client-Side Error Handling in React

```
// frontend/src/utils/errorHandler.js
// Helper functions for handling API errors
export const extractErrorMessage = (error) => {
  // Network error (no response)
  if (!error.response) {
    return 'Network error. Please check your connection.';
  }

  // Backend error with message
  if (error.response.data && error.response.data.message) {
    return error.response.data.message;
  }

  // Backend error with multiple validation errors
  if (error.response.data && error.response.data.errors &&
Array.isArray(error.response.data.errors)) {
    return error.response.data.errors.map(err => err.msg || err.message).join('.
');
  }

  // Default error message based on status code
  switch (error.response.status) {
    case 400:
      return 'Bad request. Please check your input.';
    case 401:
      return 'Unauthorized. Please log in again.';
    case 403:
      return 'Forbidden. You do not have permission to access this resource.';
    case 404:
      return 'Not found. The requested resource does not exist.';
```

```
      case 500:
        return 'Server error. Please try again later.';
      default:
        return `Error: ${error.response.status}`;
    }
  };

  // Check if error is an auth error (requiring login)
  export const isAuthError = (error) => {
    return error.response && (error.response.status === 401 || error.response.status
  === 403);
  };

  // frontend/src/components/ErrorBoundary.js
  import React, { Component } from 'react';

  class ErrorBoundary extends Component {
    constructor(props) {
      super(props);
      this.state = { hasError: false, error: null, errorInfo: null };
    }

    static getDerivedStateFromError(error) {
      // Update state so the next render will show the fallback UI
      return { hasError: true };
    }

    componentDidCatch(error, errorInfo) {
      // You can log the error to an error reporting service
      console.error('Error caught by ErrorBoundary:', error, errorInfo);
      this.setState({ error, errorInfo });

      // Optional: log to error tracking service
      // logErrorToService(error, errorInfo);
    }

    render() {
      if (this.state.hasError) {
        // You can render any custom fallback UI
        return this.props.fallback ? (
          this.props.fallback(this.state.error, this.state.errorInfo)
        ) : (
          <div className="error-boundary">
            <h2>Something went wrong.</h2>
            <details style={{ whiteSpace: 'pre-wrap' }}>
              {this.state.error && this.state.error.toString()}
              <br />
              {this.state.errorInfo && this.state.errorInfo.componentStack}
            </details>
            {this.props.resetAction && (
              <button onClick={this.props.resetAction}>
                Try again
              </button>
            )}
```

```
          </div>
        );
      }

      return this.props.children;
    }
  }

  export default ErrorBoundary;

  // frontend/src/context/ErrorContext.js
  import React, { createContext, useState, useContext } from 'react';

  const ErrorContext = createContext();

  export function useError() {
    return useContext(ErrorContext);
  }

  export function ErrorProvider({ children }) {
    const [error, setError] = useState(null);

    const showError = (message) => {
      setError(message);

      // Auto-dismiss error after 5 seconds
      setTimeout(() => {
        setError(null);
      }, 5000);
    };

    const clearError = () => {
      setError(null);
    };

    // Global error handler for async functions
    const handleAsyncError = async (asyncFunction, errorMessage = 'An error
  occurred') => {
      try {
        return await asyncFunction();
      } catch (err) {
        const message = err.response?.data?.message || errorMessage;
        showError(message);
        console.error(err);
        throw err; // Re-throw so caller can handle if needed
      }
    };

    return (
      <ErrorContext.Provider value={{ error, showError, clearError, handleAsyncError
  }}>
        {children}
        {error && (
          <div className="global-error-alert">
```

```
            <div className="alert alert-danger">
              <span>{error}</span>
              <button className="close-btn" onClick={clearError}>&times;</button>
            </div>
          </div>
        )}
      </ErrorContext.Provider>
    );
  }

  // frontend/src/App.js
  import React from 'react';
  import { BrowserRouter as Router } from 'react-router-dom';
  import Routes from './Routes';
  import Header from './components/layout/Header';
  import Footer from './components/layout/Footer';
  import ErrorBoundary from './components/ErrorBoundary';
  import { ErrorProvider } from './context/ErrorContext';
  import { AuthProvider } from './context/AuthContext';

  function App() {
    return (
      <ErrorBoundary>
        <ErrorProvider>
          <AuthProvider>
            <Router>
              <div className="app">
                <Header />
                <main className="container">
                  <Routes />
                </main>
                <Footer />
              </div>
            </Router>
          </AuthProvider>
        </ErrorProvider>
      </ErrorBoundary>
    );
  }

  export default App;

  // frontend/src/components/SomeComponent.js
  import React, { useState } from 'react';
  import { useError } from '../context/ErrorContext';
  import api from '../utils/api';

  const SomeComponent = () => {
    const [data, setData] = useState(null);
    const [loading, setLoading] = useState(false);
    const { handleAsyncError } = useError();

    const fetchData = async () => {
      setLoading(true);
```

```
    try {
      // Use the global error handler
      const result = await handleAsyncError(
        () => api.get('/api/some-endpoint'),
        'Failed to fetch data' // Custom error message
      );

      setData(result.data);
    } catch (err) {
      // Additional component-specific error handling if needed
      console.log('Component handled the error');
    } finally {
      setLoading(false);
    }
  };

  return (
    <div>
      <button onClick={fetchData} disabled={loading}>
        {loading ? 'Loading...' : 'Fetch Data'}
      </button>

      {data && (
        <div className="data-display">
          <pre>{JSON.stringify(data, null, 2)}</pre>
        </div>
      )}
    </div>
  );
};

export default SomeComponent;
```

Logging and Monitoring

Server-Side Logging with Winston

```
// backend/src/config/logger.js
const { createLogger, format, transports } = require('winston');
const { combine, timestamp, printf, colorize, json } = format;
const path = require('path');
const fs = require('fs');

// Create logs directory if it doesn't exist
const logDir = path.join(__dirname, '../../logs');
if (!fs.existsSync(logDir)) {
  fs.mkdirSync(logDir);
}

// Custom format for console output
```

```
const consoleFormat = printf(({ level, message, timestamp, ...metadata }) => {
  let msg = `${timestamp} [${level}]: ${message}`;

  if (Object.keys(metadata).length > 0) {
    msg += ` | ${JSON.stringify(metadata)}`;
  }

  return msg;
});

// Create logger
const logger = createLogger({
  level: process.env.NODE_ENV === 'production' ? 'info' : 'debug',
  format: combine(timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }), json()),
  defaultMeta: { service: 'api-service' },
  transports: [
    // Write to all logs with level 'info' and below to 'combined.log'
    new transports.File({
      filename: path.join(logDir, 'combined.log'),
      level: 'info',
    }),

    // Write to all logs with level 'error' and below to 'error.log'
    new transports.File({
      filename: path.join(logDir, 'error.log'),
      level: 'error',
    }),

    // Console output with colors in development
    new transports.Console({
      format: combine(
        colorize(),
        timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }),
        consoleFormat
      ),
    }),
  ],
  exceptionHandlers: [
    new transports.File({
      filename: path.join(logDir, 'exceptions.log'),
    }),
  ],
  rejectionHandlers: [
    new transports.File({
      filename: path.join(logDir, 'rejections.log'),
    }),
  ],
});

// Create stream for Morgan HTTP logger
logger.stream = {
  write: (message) => {
    logger.info(message.trim());
  },
```

```
};

module.exports = logger;

// backend/src/middleware/httpLogger.js
const morgan = require('morgan');
const logger = require('../config/logger');

// Create HTTP request logger middleware
const httpLogger = morgan(
  // Define log format
  ':remote-addr :method :url :status :res[content-length] - :response-time ms',
  // Options
  { stream: logger.stream }
);

module.exports = httpLogger;

// app.js
const express = require('express');
const logger = require('./config/logger');
const httpLogger = require('./middleware/httpLogger');
const app = express();

// HTTP request logger
app.use(httpLogger);

// Routes
app.use('/api/users', require('./routes/userRoutes'));
// More routes...

// Handle uncaught exceptions
process.on('uncaughtException', (err) => {
  logger.error({
    message: 'UNCAUGHT EXCEPTION - Shutting down...',
    error: err.message,
    stack: err.stack,
  });

  // Exit process
  process.exit(1);
});

// Handle unhandled promise rejections
process.on('unhandledRejection', (err) => {
  logger.error({
    message: 'UNHANDLED REJECTION - Shutting down...',
    error: err.message,
    stack: err.stack,
  });

  // Exit process
  server.close(() => {
    process.exit(1);
```

```
  });
});

// Using logger in controllers
const User = require('../models/User');
const logger = require('../config/logger');

exports.getUserById = async (req, res) => {
  try {
    logger.debug(`Getting user with ID: ${req.params.id}`);

    const user = await User.findById(req.params.id);

    if (!user) {
      logger.warn(`User not found with ID: ${req.params.id}`);
      return res.status(404).json({ message: 'User not found' });
    }

    logger.info(`User retrieved: ${user._id}`);
    res.json(user);
  } catch (err) {
    logger.error({
      message: `Error getting user with ID: ${req.params.id}`,
      error: err.message,
      stack: err.stack,
    });

    res.status(500).json({ message: 'Server error' });
  }
};
```

Monitoring with Express Status Monitor

```
// backend/src/middleware/monitor.js
const statusMonitor = require('express-status-monitor');

// Configure status monitor
const monitorOptions = {
  title: 'Express Status',
  theme: 'default.css',
  path: '/status',
  spans: [
    {
      interval: 1,
      retention: 60,
    },
    {
      interval: 5,
      retention: 60,
    },
    {
```

```
      interval: 15,
      retention: 60,
    },
  ],
  chartVisibility: {
    cpu: true,
    mem: true,
    load: true,
    responseTime: true,
    rps: true,
    statusCodes: true,
  },
  healthChecks: [
    {
      protocol: 'http',
      host: 'localhost',
      path: '/api/health',
      port: process.env.PORT || 5000,
    },
  ],
};

// Authentication middleware for status page
const monitorAuth = (req, res, next) => {
  // Only allow certain IPs or authenticated users
  const authorized =
    process.env.NODE_ENV !== 'production' ||
    (req.user && req.user.role === 'admin');

  if (!authorized) {
    return res.status(403).send('Forbidden');
  }

  next();
};

module.exports = {
  monitor: statusMonitor(monitorOptions),
  monitorAuth,
};

// backend/src/routes/healthRoutes.js
const express = require('express');
const router = express.Router();
const mongoose = require('mongoose');

// @route   GET /api/health
// @desc    Health check endpoint
// @access  Public
router.get('/', async (req, res) => {
  try {
    // Check database connection
    const dbStatus =
      mongoose.connection.readyState === 1 ? 'connected' : 'disconnected';
```

```
    // Check other dependencies (e.g., Redis, external APIs)
    // const redisStatus = await checkRedisConnection();
    // const externalApiStatus = await checkExternalApi();

    res.json({
      status: 'UP',
      timestamp: new Date(),
      services: {
        database: dbStatus,
        // redis: redisStatus,
        // externalApi: externalApiStatus
      },
    });
  } catch (err) {
    res.status(500).json({
      status: 'DOWN',
      timestamp: new Date(),
      error: err.message,
    });
  }
});

module.exports = router;

// app.js
const express = require('express');
const { monitor, monitorAuth } = require('./middleware/monitor');
const app = express();

// Status monitoring (with authentication)
app.use(monitor);
app.get('/status', monitorAuth, (req, res, next) => next());

// Health check route
app.use('/api/health', require('./routes/healthRoutes'));

// Routes
app.use('/api/users', require('./routes/userRoutes'));
// More routes...
```

Performance Monitoring

```
// backend/src/middleware/performanceMonitor.js
const onHeaders = require('on-headers');
const onFinished = require('on-finished');
const logger = require('../config/logger');

// Performance monitoring middleware
const performanceMonitor = (req, res, next) => {
  // Skip monitoring for certain routes
```

```
  if (req.path === '/status' || req.path === '/api/health') {
    return next();
  }

  // Start timer
  const start = process.hrtime();

  // Record response time when headers are sent
  onHeaders(res, () => {
    const diff = process.hrtime(start);
    const time = diff[0] * 1e3 + diff[1] * 1e-6; // Convert to ms
    res.setHeader('X-Response-Time', `${time.toFixed(3)}ms`);
  });

  // Log request completion
  onFinished(res, (err, res) => {
    const diff = process.hrtime(start);
    const time = diff[0] * 1e3 + diff[1] * 1e-6; // Convert to ms

    // Log slow responses (> 500ms)
    if (time > 500) {
      logger.warn({
        message: 'Slow response detected',
        method: req.method,
        url: req.originalUrl,
        statusCode: res.statusCode,
        responseTime: `${time.toFixed(3)}ms`,
      });
    }

    // Log performance metrics
    logger.debug({
      method: req.method,
      url: req.originalUrl,
      statusCode: res.statusCode,
      responseTime: `${time.toFixed(3)}ms`,
      contentLength: res.getHeader('content-length') || 0,
    });
  });

  next();
};

module.exports = performanceMonitor;

// backend/src/utils/profiler.js
const logger = require('../config/logger');

// Function profiler utility
const profile = (fn, name) => {
  return async (...args) => {
    const start = process.hrtime();

    try {
```

```
      // Execute the function
      const result = await fn(...args);

      // Calculate execution time
      const diff = process.hrtime(start);
      const time = diff[0] * 1e3 + diff[1] * 1e-6; // Convert to ms

      // Log performance
      logger.debug({
        message: 'Function profiling',
        function: name || fn.name,
        executionTime: `${time.toFixed(3)}ms`,
      });

      return result;
    } catch (err) {
      // Calculate execution time even on error
      const diff = process.hrtime(start);
      const time = diff[0] * 1e3 + diff[1] * 1e-6; // Convert to ms

      // Log error with performance
      logger.error({
        message: 'Function error',
        function: name || fn.name,
        executionTime: `${time.toFixed(3)}ms`,
        error: err.message,
      });

      throw err;
    }
  };
};

module.exports = profile;

// Using the profiler in services
const User = require('../models/User');
const profile = require('../utils/profiler');

// Original function
async function getUserWithPosts(userId) {
  return User.findById(userId).populate('posts');
}

// Profiled version
const getUserWithPostsProfiled = profile(getUserWithPosts, 'getUserWithPosts');

// Usage in controllers
exports.getUserProfile = async (req, res) => {
  try {
    const user = await getUserWithPostsProfiled(req.params.id);
    res.json(user);
  } catch (err) {
    res.status(500).json({ message: 'Server error' });
```

```
  }
};

// app.js
const express = require('express');
const performanceMonitor = require('./middleware/performanceMonitor');
const app = express();

// Performance monitoring middleware
app.use(performanceMonitor);

// Routes
app.use('/api/users', require('./routes/userRoutes'));
// More routes...
```

Deployment Strategies

Preparing for Production

```
// backend/src/config/config.js
require('dotenv').config();

module.exports = {
  // Application
  app: {
    port: process.env.PORT || 5000,
    nodeEnv: process.env.NODE_ENV || 'development',
    name: process.env.APP_NAME || 'my-node-app',
    url: process.env.APP_URL || 'http://localhost:5000'
  },

  // Database
  db: {
    uri: process.env.MONGODB_URI,
    options: {
      useNewUrlParser: true,
      useUnifiedTopology: true
    }
  },

  // JWT
  jwt: {
    secret: process.env.JWT_SECRET,
    expiresIn: process.env.JWT_EXPIRES_IN || '1d'
  },

  // CORS
  cors: {
    origin: process.env.CORS_ORIGIN || '*',
    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',
    preflightContinue: false,
```

```
    optionsSuccessStatus: 204,
    credentials: true
  },

  // Rate limiting
  rateLimit: {
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: process.env.RATE_LIMIT_MAX || 100, // limit each IP to 100 requests per
windowMs
    message: 'Too many requests from this IP, please try again later.'
  }
};

// package.json
{
  "name": "my-node-app",
  "version": "1.0.0",
  "scripts": {
    "start": "node src/server.js",
    "dev": "nodemon src/server.js",
    "test": "jest",
    "lint": "eslint .",
    "build": "npm run clean && npm run build:server",
    "build:server": "babel src -d dist",
    "clean": "rimraf dist",
    "postinstall": "npm run build"
  },
  "engines": {
    "node": ">=14.0.0",
    "npm": ">=6.0.0"
  },
  "dependencies": {
    // production dependencies
  },
  "devDependencies": {
    // development dependencies
  }
}

// .env.example
NODE_ENV=development
PORT=5000
MONGODB_URI=mongodb://localhost:27017/my-node-app
JWT_SECRET=your_jwt_secret_key
JWT_EXPIRES_IN=1d
CORS_ORIGIN=http://localhost:3000
RATE_LIMIT_MAX=100

// .gitignore
node_modules/
.env
.env.local
.env.development
.env.test
```

```
.env.production
logs/
dist/
coverage/
.DS_Store
npm-debug.log
yarn-debug.log
yarn-error.log

// backend/src/server.js (production-ready server)
const http = require('http');
const app = require('./app');
const config = require('./config/config');
const logger = require('./config/logger');
const mongoose = require('mongoose');

// Create HTTP server
const server = http.createServer(app);

// Connect to MongoDB
mongoose
  .connect(config.db.uri, config.db.options)
  .then(() => {
    logger.info('MongoDB connected');

    // Start server after database connection
    server.listen(config.app.port, () => {
      logger.info(`Server running in ${config.app.nodeEnv} mode on port
${config.app.port}`);
    });
  })
  .catch(err => {
    logger.error({
      message: 'MongoDB connection error',
      error: err.message,
      stack: err.stack
    });
    process.exit(1);
  });

// Handle server shutdown
const gracefulShutdown = (signal) => {
  logger.info(`${signal} received, shutting down gracefully`);

  server.close(() => {
    logger.info('HTTP server closed');

    mongoose.connection.close(false, () => {
      logger.info('MongoDB connection closed');
      process.exit(0);
    });

    // If close takes too long, force exit
    setTimeout(() => {
```

```
      logger.error('Forcing server to shut down');
      process.exit(1);
    }, 10000);
  });
};

// Listen for shutdown signals
process.on('SIGTERM', () => gracefulShutdown('SIGTERM'));
process.on('SIGINT', () => gracefulShutdown('SIGINT'));
```

Docker Deployment

```
# backend/Dockerfile
FROM node:16-alpine

# Set working directory
WORKDIR /usr/src/app

# Copy package files
COPY package*.json ./

# Install dependencies
RUN npm ci --only=production

# Copy app source
COPY . .

# Set environment variables
ENV NODE_ENV=production

# Expose port
EXPOSE 5000

# Start application
CMD ["node", "src/server.js"]
```

```
# docker-compose.yml
version: '3.8'

services:
  # Node.js API service
  api:
    build: ./backend
    image: my-node-app-api
    container_name: my-node-app-api
    restart: unless-stopped
    ports:
      - '5000:5000'
    depends_on:
```

```
      - mongodb
    env_file:
      - ./backend/.env
    networks:
      - app-network
    volumes:
      - ./backend/logs:/usr/src/app/logs
      - ./backend/uploads:/usr/src/app/uploads

  # MongoDB service
  mongodb:
    image: mongo:latest
    container_name: mongodb
    restart: unless-stopped
    environment:
      MONGO_INITDB_ROOT_USERNAME: ${MONGO_USERNAME}
      MONGO_INITDB_ROOT_PASSWORD: ${MONGO_PASSWORD}
      MONGO_INITDB_DATABASE: ${MONGO_DATABASE}
    ports:
      - '27017:27017'
    volumes:
      - mongo-data:/data/db
    networks:
      - app-network

  # Frontend React app (optional)
  frontend:
    build: ./frontend
    image: my-node-app-frontend
    container_name: my-node-app-frontend
    restart: unless-stopped
    ports:
      - '80:80'
    depends_on:
      - api
    networks:
      - app-network

# Define networks
networks:
  app-network:
    driver: bridge

# Define volumes
volumes:
  mongo-data:
    driver: local
```

Cloud Deployment (Heroku)

```
// backend/Procfile
web: node src/server.js

// package.json
{
  "name": "my-node-app",
  "version": "1.0.0",
  "engines": {
    "node": "16.x"
  },
  "scripts": {
    "start": "node src/server.js",
    "dev": "nodemon src/server.js",
    "heroku-postbuild": "NPM_CONFIG_PRODUCTION=false npm install --prefix frontend
&& npm run build --prefix frontend"
  }
}
```

Continuous Integration and Deployment (CI/CD)

```
# .github/workflows/main.yml (GitHub Actions)
name: CI/CD Pipeline

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [14.x, 16.x]
        mongodb-version: [4.4]

    steps:
      - uses: actions/checkout@v2

      - name: Setup Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v2
        with:
          node-version: ${{ matrix.node-version }}
          cache: 'npm'
          cache-dependency-path: backend/package-lock.json

      - name: Start MongoDB
        uses: supercharge/mongodb-github-action@1.7.0
```

```
        with:
          mongodb-version: ${{ matrix.mongodb-version }}

      - name: Install dependencies
        run: |
          cd backend
          npm ci

      - name: Lint
        run: |
          cd backend
          npm run lint

      - name: Test
        run: |
          cd backend
          npm test
        env:
          CI: true
          MONGODB_URI: mongodb://localhost:27017/test
          JWT_SECRET: test_jwt_secret

  deploy:
    needs: test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
      - uses: actions/checkout@v2

      - name: Deploy to Heroku
        uses: akhileshns/heroku-deploy@v3.12.12
        with:
          heroku_api_key: ${{ secrets.HEROKU_API_KEY }}
          heroku_app_name: ${{ secrets.HEROKU_APP_NAME }}
          heroku_email: ${{ secrets.HEROKU_EMAIL }}
          appdir: 'backend'
```

Performance Optimization

Backend Optimizations

```
// backend/src/middleware/compression.js
const compression = require('compression');

// Compress only certain types of responses
const shouldCompress = (req, res) => {
  // Don't compress responses with this request header
  if (req.headers['x-no-compression']) {
    return false;
  }
```

```
  // Skip compressing images, videos, and other already compressed formats
  const contentType = res.getHeader('Content-Type') || '';
  if (
    contentType.includes('image/') ||
    contentType.includes('video/') ||
    contentType.includes('audio/') ||
    contentType.includes('application/zip') ||
    contentType.includes('application/pdf')
  ) {
    return false;
  }

  // Use compression filter function (in this case, compress everything else)
  return compression.filter(req, res);
};

// Export compression middleware with custom options
module.exports = compression({
  level: 6, // Compression level (1-9, where 9 is best compression but slowest)
  threshold: 1024, // Only compress responses that are 1KB or larger
  filter: shouldCompress,
});

// backend/src/middleware/cache.js
const mcache = require('memory-cache');

// Cache middleware
const cache = (duration) => {
  return (req, res, next) => {
    // Skip caching for authenticated requests or non-GET requests
    if (req.user || req.method !== 'GET') {
      return next();
    }

    // Create a custom cache key based on URL and any query params
    const key = `__express__${req.originalUrl || req.url}`;
    const cachedBody = mcache.get(key);

    if (cachedBody) {
      // Return cached response
      res.send(cachedBody);
      return;
    }

    // Capture the original send function
    const originalSend = res.send;

    // Override res.send method to cache the response
    res.send = function (body) {
      // Only cache successful responses
      if (res.statusCode < 300 && res.statusCode >= 200) {
        mcache.put(key, body, duration * 1000);
      }
```

```
      // Call the original send method
      originalSend.call(this, body);
    };

    next();
  };
};

// Export cache middleware factory
module.exports = cache;

// app.js
const express = require('express');
const compression = require('./middleware/compression');
const cache = require('./middleware/cache');
const app = express();

// Compression middleware
app.use(compression);

// Routes with caching
app.get('/api/products', cache(60), productController.getProducts); // Cache for
60 seconds
app.get('/api/products/:id', cache(300), productController.getProductById); //
Cache for 5 minutes

// Static file serving with caching
app.use(
  '/public',
  express.static(path.join(__dirname, '../public'), {
    maxAge: '1d', // Cache static files for 1 day
  })
);
```

Database Optimizations

```
// backend/src/models/Product.js
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const productSchema = new Schema({
  name: {
    type: String,
    required: true,
    index: true, // Add index for frequently queried field
  },
  description: String,
  price: {
    type: Number,
    required: true,
```

```
    index: true, // Add index for sorting and filtering
  },
  category: {
    type: String,
    required: true,
    index: true, // Add index for filtering
  },
  tags: [String],
  inStock: {
    type: Boolean,
    default: true,
    index: true, // Add index for filtering
  },
  created: {
    type: Date,
    default: Date.now,
    index: true, // Add index for sorting
  },
});

// Compound index for common query pattern
productSchema.index({ category: 1, price: -1 });

// Create text index for search
productSchema.index(
  { name: 'text', description: 'text', tags: 'text' },
  { weights: { name: 10, description: 5, tags: 3 } }
);

// Pre-fetch related data
productSchema.virtual('reviews', {
  ref: 'Review',
  localField: '_id',
  foreignField: 'product',
});

// Optimize query with projection and pagination
const Product = mongoose.model('Product', productSchema);

// Optimized product search function
async function searchProducts(options) {
  const {
    search,
    category,
    minPrice,
    maxPrice,
    inStock,
    sort,
    page = 1,
    limit = 20,
    select,
  } = options;

  // Build query
```

```
    const query = {};

    // Full-text search
    if (search) {
      query.$text = { $search: search };
    }

    // Category filter
    if (category) {
      query.category = category;
    }

    // Price range
    if (minPrice !== undefined || maxPrice !== undefined) {
      query.price = {};
      if (minPrice !== undefined) query.price.$gte = minPrice;
      if (maxPrice !== undefined) query.price.$lte = maxPrice;
    }

    // In stock filter
    if (inStock !== undefined) {
      query.inStock = inStock;
    }

    // Build sort options
    const sortOptions = {};
    if (search) {
      // If text search, sort by relevance
      sortOptions.score = { $meta: 'textScore' };
    } else if (sort) {
      // Parse sort string (e.g., 'price:-1,name:1')
      sort.split(',').forEach((s) => {
        const [field, order] = s.split(':');
        sortOptions[field] = parseInt(order, 10);
      });
    } else {
      // Default sort
      sortOptions.created = -1;
    }

    // Calculate pagination
    const skip = (page - 1) * limit;

    // Execute query with projection and pagination
    const products = await Product.find(query)
      .select(select || '')
      .sort(sortOptions)
      .skip(skip)
      .limit(limit)
      .lean(); // Return plain objects instead of Mongoose documents for performance

    // Count total (in a separate query for performance)
    const total = await Product.countDocuments(query);
```

```
    return {
      products,
      pagination: {
        page,
        limit,
        total,
        pages: Math.ceil(total / limit),
      },
    };
  }

  module.exports = {
    Product,
    searchProducts,
  };
```

Comparison with React

Server-Side Rendering vs. Client-Side Rendering

Server-Side Rendering (SSR) Approach

In server-side rendering, HTML is generated on the server for each request and sent to the client.

Pros:

- Better initial load performance and First Contentful Paint (FCP)
- Improved SEO as crawlers can see the full content
- Works without JavaScript enabled
- Better performance on low-powered devices

Cons:

- Higher server load
- Full page reloads between routes (unless enhanced with client-side routing)
- More complex development setup
- Less interactive UI experience

Example of Server-Side Rendering with Express and EJS:

```
// Server-side rendered application with Express and EJS
const express = require('express');
const app = express();

// View engine setup
app.set('view engine', 'ejs');
app.set('views', 'views');

// Routes
app.get('/', async (req, res) => {
  // Fetch data from database
```

```
  const products = await Product.find().limit(10);

  // Render the template with data
  res.render('index', {
    title: 'Home Page',
    products,
  });
});

app.get('/products/:id', async (req, res) => {
  try {
    // Fetch product data
    const product = await Product.findById(req.params.id);

    if (!product) {
      return res.status(404).render('error', {
        message: 'Product not found',
      });
    }

    // Fetch related products
    const relatedProducts = await Product.find({
      category: product.category,
      _id: { $ne: product._id },
    }).limit(4);

    // Render the template with data
    res.render('product-detail', {
      title: product.name,
      product,
      relatedProducts,
    });
  } catch (err) {
    res.status(500).render('error', {
      message: 'Server error',
    });
  }
});

app.listen(3000);
```

```
<!-- views/index.ejs -->
<%- include('partials/header') %>

<main class="container">
  <h1><%= title %></h1>

  <div class="product-grid">
    <% products.forEach(function(product) { %>
    <div class="product-card">
      <h3><%= product.name %></h3>
```

```
      <p><%= product.description %></p>
      <span class="price">$<%= product.price.toFixed(2) %></span>
      <a href="/products/<%= product._id %>" class="btn">View Details</a>
    </div>
    <% }); %>
  </div>
</main>

<%- include('partials/footer') %>
```

Client-Side Rendering (CSR) Approach

In client-side rendering, the server provides a minimal HTML shell, and JavaScript builds the UI in the browser.

Pros:

- Rich, app-like user experience
- Faster navigation after initial load
- Reduced server load (after initial page load)
- Clearer separation of frontend and backend concerns

Cons:

- Slower initial load and First Contentful Paint
- SEO challenges (though improving with modern crawlers)
- Requires JavaScript to be enabled
- Potentially more complex state management

Example of Client-Side Rendering with React:

```
// Backend API (Express)
const express = require('express');
const router = express.Router();

// API routes
router.get('/api/products', async (req, res) => {
  try {
    const products = await Product.find().limit(10);
    res.json(products);
  } catch (err) {
    res.status(500).json({ message: 'Server error' });
  }
});

router.get('/api/products/:id', async (req, res) => {
  try {
    const product = await Product.findById(req.params.id);

    if (!product) {
      return res.status(404).json({ message: 'Product not found' });
    }
```

```
    res.json(product);
  } catch (err) {
    res.status(500).json({ message: 'Server error' });
  }
});

module.exports = router;
```

```
// Frontend (React)
// ProductList.jsx
import React, { useState, useEffect } from 'react';
import { Link } from 'react-router-dom';
import axios from 'axios';

const ProductList = () => {
  const [products, setProducts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchProducts = async () => {
      try {
        const res = await axios.get('/api/products');
        setProducts(res.data);
        setLoading(false);
      } catch (err) {
        setError('Failed to fetch products');
        setLoading(false);
      }
    };

    fetchProducts();
  }, []);

  if (loading) return <div>Loading...</div>;
  if (error) return <div className="error">{error}</div>;

  return (
    <div className="container">
      <h1>Products</h1>

      <div className="product-grid">
        {products.map(product => (
          <div key={product._id} className="product-card">
            <h3>{product.name}</h3>
            <p>{product.description}</p>
            <span className="price">${product.price.toFixed(2)}</span>
            <Link to={`/products/${product._id}`} className="btn">View
Details</Link>
          </div>
```

```
        ))}
      </div>
    </div>
  );
};

export default ProductList;

// ProductDetail.jsx
import React, { useState, useEffect } from 'react';
import { useParams, Link } from 'react-router-dom';
import axios from 'axios';

const ProductDetail = () => {
  const { id } = useParams();
  const [product, setProduct] = useState(null);
  const [relatedProducts, setRelatedProducts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const fetchProduct = async () => {
      try {
        setLoading(true);

        // Fetch product details
        const productRes = await axios.get(`/api/products/${id}`);
        setProduct(productRes.data);

        // Fetch related products
        const relatedRes = await axios.get(`/api/products/related/${id}`);
        setRelatedProducts(relatedRes.data);

        setLoading(false);
      } catch (err) {
        setError(
          err.response?.status === 404
            ? 'Product not found'
            : 'Error loading product'
        );
        setLoading(false);
      }
    };

    fetchProduct();
  }, [id]);

  if (loading) return <div>Loading...</div>;
  if (error) return <div className="error">{error}</div>;

  return (
    <div className="container">
      <div className="product-detail">
        <h1>{product.name}</h1>
```

```
          <p className="description">{product.description}</p>
          <div className="price-box">
            <span className="price">${product.price.toFixed(2)}</span>
            <button className="btn add-to-cart">Add to Cart</button>
          </div>

          {/* More product details... */}
        </div>

        {relatedProducts.length > 0 && (
          <div className="related-products">
            <h2>Related Products</h2>
            <div className="product-grid">
              {relatedProducts.map(item => (
                <div key={item._id} className="product-card">
                  <h3>{item.name}</h3>
                  <span className="price">${item.price.toFixed(2)}</span>
                  <Link to={`/products/${item._id}`} className="btn">View
Details</Link>
                </div>
              ))}
            </div>
          </div>
        )}
      </div>
    );
};

export default ProductDetail;
```

Architectural Differences

Traditional Server-Side Web Application

```
[ Browser ] <--HTML--> [ Server (Express + Template Engine) ] <---> [ Database ]
```

- Each page request triggers a full page load
- Server generates complete HTML
- Limited JavaScript for enhancements
- Sessions often stored server-side

Key Components:

1. Express.js server
2. Template engine (EJS, Pug, Handlebars)
3. Route handlers that render views
4. Forms that submit to server endpoints
5. Server-side data validation and processing

Single Page Application (SPA) with API Backend

```
[ Browser (React SPA) ] <--JSON--> [ API Server (Express) ] <---> [ Database ]
```

- Initial HTML shell, then API calls for data
- Client-side routing
- Rich JavaScript interactions
- State managed client-side
- Authentication via tokens (JWT)

Key Components:

1. React frontend application
2. Client-side routing (React Router)
3. State management (useState, useContext, Redux)
4. RESTful or GraphQL API
5. Express.js backend

Isomorphic/Universal JavaScript

```
                      ┌── [ Client-side React ] --┐
                      │                           │
[ Browser ] <--HTML--> [ Server (Express + React) ] <---> [ Database ]
                      └------------ JSON ----------┘
```

- Server renders initial HTML for fast loading
- Hydration enables client-side interactivity
- Same React components used server and client-side
- Shared code between frontend and backend

Key Components:

1. Server-side rendering setup
2. Hydration process
3. Shared routing configuration
4. Data fetching strategies

Data Flow Patterns

Server-Side Data Flow

```
1. User requests page → Server receives request
2. Server queries database → Gets data
3. Server processes data → Applies business logic
```

```
4. Server renders template with data → Generates HTML
5. Server sends complete HTML to browser → Browser displays page
```

Example:

```
// Server-side data flow
app.get('/users/:id', async (req, res) => {
  try {
    // Step 1: Authenticate user (if needed)
    if (!req.session.user) {
      return res.redirect('/login');
    }

    // Step 2: Fetch data from database
    const user = await User.findById(req.params.id);

    if (!user) {
      return res.status(404).render('error', { message: 'User not found' });
    }

    // Step 3: Check permissions
    if (req.session.user.id !== user.id && req.session.user.role !== 'admin') {
      return res.status(403).render('error', { message: 'Access denied' });
    }

    // Step 4: Fetch related data
    const posts = await Post.find({ user: user.id }).limit(5);

    // Step 5: Process data (if needed)
    const userViewModel = {
      id: user.id,
      name: user.name,
      email: user.email,
      joinDate: formatDate(user.createdAt),
      posts: posts.map((post) => ({
        id: post.id,
        title: post.title,
        excerpt: truncate(post.content, 100),
      })),
    };

    // Step 6: Render view with data
    res.render('user-profile', {
      title: `${user.name}'s Profile`,
      user: userViewModel,
    });
  } catch (err) {
    console.error(err);
    res.status(500).render('error', { message: 'Server error' });
  }
});
```

Client-Side Data Flow (React)

```
1. React component mounts → useEffect hook fires
2. Component makes API request → Calls backend API
3. API returns JSON data → Component updates state
4. Component re-renders with new state → UI updates
5. User interactions → Local state changes or new API calls
```

Example:

```jsx
// UserProfile.jsx - Client-side data flow
import React, { useState, useEffect } from 'react';
import { useParams, useNavigate } from 'react-router-dom';
import axios from 'axios';
import { useAuth } from '../context/AuthContext';
import Posts from '../components/Posts';
import ErrorAlert from '../components/ErrorAlert';

const UserProfile = () => {
  const { id } = useParams();
  const { user: currentUser } = useAuth();
  const navigate = useNavigate();

  const [user, setUser] = useState(null);
  const [posts, setPosts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    // Step 1: Check if user is logged in
    if (!currentUser) {
      navigate('/login');
      return;
    }

    // Step 2: Fetch user data
    const fetchUserData = async () => {
      try {
        setLoading(true);

        // Fetch user details
        const userRes = await axios.get(`/api/users/${id}`);
        setUser(userRes.data);

        // Step 3: Check permissions
        if (currentUser.id !== id && currentUser.role !== 'admin') {
          setError('You do not have permission to view this profile');
          setLoading(false);
          return;
```

```
        }

        // Step 4: Fetch related data
        const postsRes = await axios.get(`/api/users/${id}/posts`);
        setPosts(postsRes.data);

        setLoading(false);
      } catch (err) {
        setError(
          err.response?.status === 404
            ? 'User not found'
            : 'Error loading user profile'
        );
        setLoading(false);
      }
    };

    fetchUserData();
  }, [id, currentUser, navigate]);

  if (loading) return <div className="loading">Loading...</div>;
  if (error) return <ErrorAlert message={error} />;

  return (
    <div className="container">
      <h1>{user.name}'s Profile</h1>

      <div className="profile-info">
        <p><strong>Email:</strong> {user.email}</p>
        <p><strong>Joined:</strong> {new
Date(user.createdAt).toLocaleDateString()}</p>

        {/* More user details... */}
      </div>

      <div className="user-posts">
        <h2>Recent Posts</h2>
        {posts.length > 0 ? (
          <Posts posts={posts} />
        ) : (
          <p>No posts yet.</p>
        )}
      </div>
    </div>
  );
};

export default UserProfile;
```

Flux/Redux Data Flow

The Flux pattern (popularized by Redux) introduces a unidirectional data flow that centralizes state management.

```
1. User interaction → Dispatch an action
2. Action processed → Reducer updates store
3. Store changes → Components re-render
4. API interactions → Async actions via middleware
```

Example with Redux:

```
// actions.js
export const FETCH_USER_REQUEST = 'FETCH_USER_REQUEST';
export const FETCH_USER_SUCCESS = 'FETCH_USER_SUCCESS';
export const FETCH_USER_FAILURE = 'FETCH_USER_FAILURE';

export const fetchUserRequest = () => ({
  type: FETCH_USER_REQUEST
});

export const fetchUserSuccess = (user) => ({
  type: FETCH_USER_SUCCESS,
  payload: user
});

export const fetchUserFailure = (error) => ({
  type: FETCH_USER_FAILURE,
  payload: error
});

// Async action creator with Redux Thunk
export const fetchUser = (id) => {
  return async (dispatch) => {
    dispatch(fetchUserRequest());

    try {
      const response = await axios.get(`/api/users/${id}`);
      dispatch(fetchUserSuccess(response.data));
    } catch (error) {
      dispatch(fetchUserFailure(error.message));
    }
  };
};

// reducer.js
const initialState = {
  user: null,
  loading: false,
  error: null
};
```

```
const userReducer = (state = initialState, action) => {
  switch (action.type) {
    case FETCH_USER_REQUEST:
      return {
        ...state,
        loading: true,
        error: null
      };
    case FETCH_USER_SUCCESS:
      return {
        ...state,
        loading: false,
        user: action.payload
      };
    case FETCH_USER_FAILURE:
      return {
        ...state,
        loading: false,
        error: action.payload
      };
    default:
      return state;
  }
};

// UserProfile.jsx with Redux
import React, { useEffect } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { useParams } from 'react-router-dom';
import { fetchUser } from '../actions/userActions';

const UserProfile = () => {
  const { id } = useParams();
  const dispatch = useDispatch();
  const { user, loading, error } = useSelector(state => state.user);

  useEffect(() => {
    dispatch(fetchUser(id));
  }, [dispatch, id]);

  if (loading) return <div>Loading...</div>;
  if (error) return <div className="error">{error}</div>;
  if (!user) return <div>No user found</div>;

  return (
    <div className="container">
      <h1>{user.name}'s Profile</h1>
      {/* User details... */}
    </div>
  );
};

export default UserProfile;
```

Context API Data Flow

React's Context API provides a simpler alternative to Redux for managing global state.

```
// UserContext.js
import React, { createContext, useReducer, useContext, useEffect } from 'react';
import axios from 'axios';

// Create context
const UserContext = createContext();

// Define reducer
const userReducer = (state, action) => {
  switch (action.type) {
    case 'FETCH_REQUEST':
      return { ...state, loading: true, error: null };
    case 'FETCH_SUCCESS':
      return { ...state, loading: false, user: action.payload };
    case 'FETCH_FAILURE':
      return { ...state, loading: false, error: action.payload };
    default:
      return state;
  }
};

// Provider component
export const UserProvider = ({ children }) => {
  const [state, dispatch] = useReducer(userReducer, {
    user: null,
    loading: false,
    error: null
  });

  // Function to fetch user
  const fetchUser = async (id) => {
    dispatch({ type: 'FETCH_REQUEST' });

    try {
      const response = await axios.get(`/api/users/${id}`);
      dispatch({ type: 'FETCH_SUCCESS', payload: response.data });
    } catch (error) {
      dispatch({ type: 'FETCH_FAILURE', payload: error.message });
    }
  };

  return (
    <UserContext.Provider value={{ ...state, fetchUser }}>
      {children}
    </UserContext.Provider>
  );
};
```

```jsx
// Custom hook to use the context
export const useUser = () => useContext(UserContext);

// UserProfile.jsx with Context API
import React, { useEffect } from 'react';
import { useParams } from 'react-router-dom';
import { useUser } from '../context/UserContext';

const UserProfile = () => {
  const { id } = useParams();
  const { user, loading, error, fetchUser } = useUser();

  useEffect(() => {
    fetchUser(id);
  }, [fetchUser, id]);

  if (loading) return <div>Loading...</div>;
  if (error) return <div className="error">{error}</div>;
  if (!user) return <div>No user found</div>;

  return (
    <div className="container">
      <h1>{user.name}'s Profile</h1>
      {/* User details... */}
    </div>
  );
};

export default UserProfile;
```

Use Cases for Each Approach

When to Use Server-Side Rendering (Express + Templates)

Best for:

- Content-focused websites (blogs, news sites, documentation)
- SEO-critical applications
- Projects where initial load performance is critical
- Applications with limited client-side interactivity
- When supporting older browsers or low-powered devices is essential
- Projects where development speed is prioritized over complex UI interactions

Example Scenarios:

1. **Corporate Websites and Landing Pages**
    - Content is relatively static
    - SEO is a top priority
    - Performance metrics like First Contentful Paint are critical

2. **Content Management Systems**
    - Administrative interfaces with forms
    - Mixed content types (text, images, etc.)
    - User roles and permissions
3. **E-commerce Product Listings**
    - SEO for product pages is essential
    - Fast initial loading of product information
    - Simple filtering and sorting functionality

When to Use Client-Side Rendering (React SPA)

Best for:

- Highly interactive applications
- Real-time features and updates
- Complex user interfaces with rich interactions
- Applications where user experience after initial load is prioritized
- When building mobile-like web applications
- Dashboard and admin interfaces with complex state

Example Scenarios:

1. **Admin Dashboards**
    - Complex data visualization
    - Real-time updates
    - Multiple interactive components
    - Complex form handling
2. **Social Media Applications**
    - Real-time updates and notifications
    - Complex UI with multiple states
    - Heavy user interactions (likes, comments, shares)
3. **Collaborative Tools**
    - Real-time document editing
    - Chat and messaging features
    - Complex drag-and-drop interfaces

When to Use Hybrid Approaches

Best for:

- Applications that need both SEO and rich interactivity
- E-commerce platforms with product discovery and complex checkout
- Content platforms with interactive features

- Applications where different sections have different requirements

Example Scenarios:

1. **E-commerce Platforms**
    - SSR for product listings and SEO
    - CSR for cart, checkout, and account management
2. **Content Platforms with User-Generated Content**
    - SSR for main content and SEO
    - CSR for comments, ratings, and interactive elements
3. **Educational Platforms**
    - SSR for course content and discoverability
    - CSR for interactive lessons, quizzes, and progress tracking

Hybrid Approaches

Server-Side Rendering with Hydration

This approach renders HTML on the server first, then "hydrates" the page with JavaScript to make it interactive.

Next.js Example:

```
// pages/index.js
import { useState } from 'react';
import axios from 'axios';

// This function runs on the server
export async function getServerSideProps() {
  try {
    // Fetch data on the server
    const response = await axios.get('https://api.example.com/products');

    // Pass data to the page component as props
    return {
      props: {
        products: response.data,
        error: null
      }
    };
  } catch (error) {
    return {
      props: {
        products: [],
        error: 'Failed to fetch products'
      }
    };
```

```
  }
}

// This component renders on both server and client
export default function Home({ products, error }) {
  // Client-side state (only active after hydration)
  const [cart, setCart] = useState([]);

  // Client-side function
  const addToCart = (product) => {
    setCart([...cart, product]);
  };

  if (error) {
    return <div className="error">{error}</div>;
  }

  return (
    <div className="container">
      <h1>Products</h1>

      <div className="product-grid">
        {products.map(product => (
          <div key={product.id} className="product-card">
            <h3>{product.name}</h3>
            <p>{product.description}</p>
            <span className="price">${product.price.toFixed(2)}</span>

            {/* Client-side interactive element */}
            <button
              onClick={() => addToCart(product)}
              className="btn"
            >
              Add to Cart
            </button>
          </div>
        ))}
      </div>

      {/* Cart summary (client-side only) */}
      <div className="cart-summary">
        <h2>Cart ({cart.length} items)</h2>
        {cart.length > 0 ? (
          <ul>
            {cart.map((item, index) => (
              <li key={index}>{item.name} - ${item.price.toFixed(2)}</li>
            ))}
          </ul>
        ) : (
          <p>Your cart is empty</p>
        )}
      </div>
    </div>
```

```
  );
}
```

Progressive Enhancement

This approach starts with a functional baseline experience and enhances it with JavaScript when available.

Express + JavaScript Example:

```
// server.js
app.get('/products', async (req, res) => {
  try {
    const products = await Product.find();

    // Check if request accepts JSON (likely an AJAX request)
    if (req.xhr || req.headers.accept.includes('application/json')) {
      return res.json(products);
    }

    // Otherwise, render the full HTML page
    res.render('products', { products });
  } catch (err) {
    res.status(500).render('error', { message: 'Server error' });
  }
});
```

```
<!-- views/products.ejs -->
<%- include('partials/header') %>

<main class="container" id="products-container">
  <h1>Products</h1>

  <div class="product-grid">
    <% products.forEach(function(product) { %>
      <div class="product-card" data-product-id="<%= product._id %>">
        <h3><%= product.name %></h3>
        <p><%= product.description %></p>
        <span class="price">$<%= product.price.toFixed(2) %></span>

        <!-- Basic form submission works without JavaScript -->
        <form action="/cart/add" method="POST" class="add-to-cart-form">
          <input type="hidden" name="productId" value="<%= product._id %>">
          <button type="submit" class="btn">Add to Cart</button>
        </form>
      </div>
    <% }); %>
  </div>
</main>
```

```
<!-- JavaScript enhancement -->
<script>
  // Enhanced cart functionality with JavaScript
  document.addEventListener('DOMContentLoaded', function() {
    // Get all add-to-cart forms
    const forms = document.querySelectorAll('.add-to-cart-form');

    // Add event listeners to each form
    forms.forEach(form => {
      form.addEventListener('submit', async function(e) {
        // Prevent default form submission
        e.preventDefault();

        const productId = this.querySelector('input[name="productId"]').value;

        try {
          // Make AJAX request to add item to cart
          const response = await fetch('/cart/add', {
            method: 'POST',
            headers: {
              'Content-Type': 'application/json',
              'X-Requested-With': 'XMLHttpRequest'
            },
            body: JSON.stringify({ productId })
          });

          const data = await response.json();

          // Update cart UI without page reload
          updateCartUI(data.cart);

          // Show success message
          showNotification('Product added to cart!');
        } catch (err) {
          console.error('Error adding to cart:', err);
          showNotification('Failed to add product to cart', 'error');
        }
      });
    });

    // Helper functions
    function updateCartUI(cart) {
      const cartCount = document.getElementById('cart-count');
      if (cartCount) {
        cartCount.textContent = cart.items.length;
      }
    }

    function showNotification(message, type = 'success') {
      const notification = document.createElement('div');
      notification.className = `notification ${type}`;
      notification.textContent = message;

      document.body.appendChild(notification);
```

```
      // Remove after 3 seconds
      setTimeout(() => {
        notification.remove();
      }, 3000);
    }
  });
</script>

<%- include('partials/footer') %>
```

Micro Frontends

This architecture breaks down a frontend application into smaller, independently deployable pieces.

Using iframes or Web Components:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Micro Frontend Example</title>
  <link rel="stylesheet" href="/css/main.css">
</head>
<body>
  <header>
    <!-- Header micro frontend -->
    <div id="header-container"></div>
  </header>

  <main>
    <!-- Main content micro frontend based on route -->
    <div id="content-container"></div>
  </main>

  <aside>
    <!-- Shopping cart micro frontend -->
    <div id="cart-container"></div>
  </aside>

  <footer>
    <!-- Footer micro frontend -->
    <div id="footer-container"></div>
  </footer>

  <script>
    // Load micro frontends
    document.addEventListener('DOMContentLoaded', function() {
      // Load header (vanilla JavaScript micro frontend)
      fetch('/micro-frontends/header')
```

```
        .then(response => response.text())
        .then(html => {
          document.getElementById('header-container').innerHTML = html;
          // Execute any scripts
          const scripts = document.getElementById('header-
container').querySelectorAll('script');
          scripts.forEach(script => {
            const newScript = document.createElement('script');
            newScript.textContent = script.textContent;
            document.head.appendChild(newScript);
          });
        });

      // Load content based on route (React micro frontend)
      const route = window.location.pathname;

      if (route.startsWith('/products')) {
        // Load React micro frontend for products
        const script = document.createElement('script');
        script.src = '/micro-frontends/products/bundle.js';
        script.onload = function() {
          // Initialize the micro frontend
          window.ProductsMicroFrontend.mount('content-container');
        };
        document.head.appendChild(script);
      } else if (route.startsWith('/account')) {
        // Load Vue micro frontend for account
        const script = document.createElement('script');
        script.src = '/micro-frontends/account/bundle.js';
        script.onload = function() {
          // Initialize the micro frontend
          window.AccountMicroFrontend.mount('content-container');
        };
        document.head.appendChild(script);
      }

      // Load shopping cart (Web Component micro frontend)
      const cartScript = document.createElement('script');
      cartScript.src = '/micro-frontends/cart/bundle.js';
      cartScript.onload = function() {
        const cartEl = document.createElement('shopping-cart');
        document.getElementById('cart-container').appendChild(cartEl);
      };
      document.head.appendChild(cartScript);

      // Load footer
      fetch('/micro-frontends/footer')
        .then(response => response.text())
        .then(html => {
          document.getElementById('footer-container').innerHTML = html;
        });
    });
  </script>
```

```
</body>
</html>
```

API-First Development

This approach focuses on building a robust API first, then creating multiple frontends (web, mobile, etc.) that consume it.

Express API with Multiple Clients:

```
// Express API
const express = require('express');
const app = express();
const apiRoutes = require('./routes/api');

// API Routes
app.use('/api', apiRoutes);

// Web client (SSR)
app.use('/web', require('./routes/web'));

// Serve React SPA
app.use('/app', express.static('dist/app'));

// Serve mobile web version
app.use('/mobile', express.static('dist/mobile'));

app.listen(3000);
```

Performance Considerations

Server-Side Rendering Performance

Optimizing SSR Performance:

1. **Implement caching strategies**

```
const NodeCache = require('node-cache');
const pageCache = new NodeCache({ stdTTL: 60 }); // 1 minute cache

// Cache middleware
function cacheMiddleware(req, res, next) {
  // Skip cache for authenticated users
  if (req.session && req.session.user) {
    return next();
  }

  const key = `page_${req.originalUrl}`;
  const cachedBody = pageCache.get(key);
```

```
  if (cachedBody) {
    // Return cached response
    return res.send(cachedBody);
  }

  // Store original send
  const originalSend = res.send;

  // Override send
  res.send = function(body) {
    pageCache.set(key, body);
    originalSend.call(this, body);
  };

  next();
}

// Use for specific routes
app.get('/', cacheMiddleware, homeController.getHomePage);
```

2. **Use partial rendering**

```
app.get('/products/:id', async (req, res) => {
  try {
    const product = await Product.findById(req.params.id);

    if (!product) {
      return res.status(404).render('error', { message: 'Product not found'
});
    }

    // If AJAX request, render only the product details
    if (req.xhr) {
      return res.render('partials/product-detail', { product });
    }

    // Otherwise render the full page
    res.render('product-detail', {
      title: product.name,
      product
    });
  } catch (err) {
    res.status(500).render('error', { message: 'Server error' });
  }
});
```

3. **Optimize template rendering**

```
// Precompile templates
const ejs = require('ejs');
const fs = require('fs');
const path = require('path');

// Precompile a template
const templatePath = path.join(__dirname, 'views', 'product-card.ejs');
const templateStr = fs.readFileSync(templatePath, 'utf-8');
const compiledTemplate = ejs.compile(templateStr, { filename: templatePath
});

// Use the compiled template
app.get('/api/products/card/:id', async (req, res) => {
  const product = await Product.findById(req.params.id);
  const html = compiledTemplate({ product });
  res.send(html);
});
```

Client-Side Rendering Performance

Optimizing React Performance:

1. **Code splitting and lazy loading**

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Loading from './components/Loading';

// Lazy load components
const Home = lazy(() => import('./pages/Home'));
const Products = lazy(() => import('./pages/Products'));
const ProductDetail = lazy(() => import('./pages/ProductDetail'));
const Cart = lazy(() => import('./pages/Cart'));

function App() {
  return (
    <Router>
      <Suspense fallback={<Loading />}>
        <Switch>
          <Route exact path="/" component={Home} />
          <Route exact path="/products" component={Products} />
          <Route path="/products/:id" component={ProductDetail} />
          <Route path="/cart" component={Cart} />
        </Switch>
      </Suspense>
    </Router>
  );
}
```

2. **Memoization for expensive calculations**

```
import React, { useMemo } from 'react';

function ProductList({ products, category, sort }) {
  // Memoized filtering and sorting
  const filteredAndSortedProducts = useMemo(() => {
    console.log('Computing filtered and sorted products');

    // Filter products by category
    let result = category
      ? products.filter(p => p.category === category)
      : products;

    // Sort products
    if (sort === 'price-asc') {
      result = [...result].sort((a, b) => a.price - b.price);
    } else if (sort === 'price-desc') {
      result = [...result].sort((a, b) => b.price - a.price);
    }

    return result;
  }, [products, category, sort]); // Only recompute when these dependencies
change

  return (
    <div className="product-grid">
      {filteredAndSortedProducts.map(product => (
        <ProductCard key={product.id} product={product} />
      ))}
    </div>
  );
}
```

3. **Virtualized lists for large datasets**

```
import React from 'react';
import { FixedSizeList } from 'react-window';

function VirtualizedProductList({ products }) {
  const Row = ({ index, style }) => {
    const product = products[index];

    return (
      <div style={style} className="product-row">
        <h3>{product.name}</h3>
        <p>{product.description}</p>
        <span>${product.price.toFixed(2)}</span>
      </div>
    );
```

```
  };

  return (
    <FixedSizeList
      height={500}
      width="100%"
      itemCount={products.length}
      itemSize={100}
    >
      {Row}
    </FixedSizeList>
  );
}
```

Reference Section

Essential npm Packages and Their Purposes

Core Packages

Package Name	Description	Use Case
express	Web framework for Node.js	Building web applications and APIs
mongoose	MongoDB object modeling	Interacting with MongoDB databases
dotenv	Environment variables manager	Managing configuration across environments
body-parser	Request body parser	Parsing incoming request bodies
cors	Cross-Origin Resource Sharing	Enabling CORS for your Express app
helmet	Security middleware	Adding security headers to Express apps
morgan	HTTP request logger	Logging HTTP requests
winston	Logging library	Advanced logging capabilities
passport	Authentication middleware	Implementing authentication strategies
jsonwebtoken	JWT implementation	Token-based authentication
bcrypt	Password hashing	Securely storing passwords
express-validator	Input validation	Validating and sanitizing inputs
multer	File upload handler	Handling multipart/form-data for file uploads
socket.io	Real-time communication	Implementing WebSockets for real-time features
nodemailer	Email sending	Sending emails from Node.js applications
compression	Response compression	Compressing HTTP responses

Package Name	Description	Use Case
`express-rate-limit`	Rate limiting middleware	Limiting repeated requests to public APIs
`axios`	HTTP client	Making HTTP requests to external APIs
`moment`	Date manipulation	Working with dates and times
`uuid`	UUID generator	Generating unique identifiers
`joi`	Schema validator	Object schema validation
`lodash`	Utility library	General-purpose utility functions
`redis`	Redis client	Working with Redis for caching/sessions
`pg`	PostgreSQL client	Working with PostgreSQL databases
`sequelize`	SQL ORM	ORM for SQL databases

Template Engine Packages

Package Name	Description	Use Case
`ejs`	Embedded JavaScript templates	HTML templates with JavaScript
`pug`	Template engine (formerly Jade)	Simplified, whitespace-sensitive HTML
`handlebars`	Logic-less templating	Semantic templates with minimal logic
`express-handlebars`	Handlebars for Express	Integration of Handlebars with Express
`nunjucks`	Jinja2-inspired templates	Rich and powerful templating

Development Packages

Package Name	Description	Use Case
`nodemon`	Development server	Auto-restarting Node.js applications
`jest`	Testing framework	Writing and running tests
`mocha`	Testing framework	Alternative testing framework
`chai`	Assertion library	Making test assertions
`supertest`	HTTP testing	Testing HTTP servers/routes
`eslint`	Linting utility	Enforcing code style and catching errors
`prettier`	Code formatter	Automatic code formatting
`husky`	Git hooks	Running scripts before commit/push
`debug`	Debugging utility	Better debugging output
`concurrently`	Run multiple commands	Running backend and frontend simultaneously

Command Cheat Sheet

Node.js and npm Commands

```
# Node.js
node app.js                     # Run a JavaScript file
node --inspect app.js           # Run with inspector
node --inspect-brk app.js       # Run with inspector, break on first line
node -e "console.log('Hi')"     # Execute code from command line

# npm
npm init                        # Create a new package.json
npm init -y                     # Create with default values
npm install <package>           # Install a package locally
npm install -g <package>        # Install a package globally
npm install --save-dev <pkg>    # Install as development dependency
npm uninstall <package>         # Uninstall a package
npm update                      # Update all packages
npm update <package>            # Update a specific package
npm run <script>                # Run a script defined in package.json
npm list                        # List installed packages
npm outdated                    # Check for outdated packages
npm audit                       # Check for vulnerabilities
npm audit fix                   # Fix vulnerabilities
npm cache clean --force         # Clean npm cache
npm config list                 # List npm configuration
npm search <keyword>            # Search for packages
```

Express Commands and Setup

```
# Install Express
npm install express

# Install Express Generator globally
npm install -g express-generator

# Generate an Express application
express --view=ejs myapp
cd myapp
npm install

# Basic Express server
node -e "
const express = require('express');
const app = express();
app.get('/', (req, res) => res.send('Hello World'));
app.listen(3000, () => console.log('Server running on port 3000'));
"
```

MongoDB Commands

```
# Start MongoDB server
mongod --dbpath /data/db

# Connect to MongoDB shell
mongo

# MongoDB shell commands
show dbs                      # Show databases
use <database>                # Switch to database
show collections              # Show collections
db.<collection>.find()        # Find all documents
db.<collection>.insertOne({}) # Insert a document
db.<collection>.updateOne({}, {$set: {}}) # Update a document
db.<collection>.deleteOne({}) # Delete a document
db.<collection>.createIndex() # Create an index

# Mongoose connection
node -e "
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/test', {
  useNewUrlParser: true,
  useUnifiedTopology: true
}).then(() => console.log('Connected to MongoDB'))
  .catch(err => console.error('MongoDB connection error:', err));
"
```

Git Commands for Node.js Projects

```
# Initialize a new repository
git init

# Create .gitignore for Node.js
echo "node_modules/\n.env\ndist/\ncoverage/\n.DS_Store" > .gitignore

# Add and commit files
git add .
git commit -m "Initial commit"

# Create a new branch
git checkout -b feature/user-authentication

# Merge branches
git checkout main
git merge feature/user-authentication

# Push to remote repository
git remote add origin <repository-url>
git push -u origin main
```

Docker Commands for Node.js Applications

```
# Build a Docker image
docker build -t my-node-app .

# Run a Docker container
docker run -p 3000:3000 my-node-app

# Run with environment variables
docker run -p 3000:3000 -e "NODE_ENV=production" my-node-app

# Run with mounted volume
docker run -p 3000:3000 -v $(pwd):/usr/src/app my-node-app

# Docker Compose
docker-compose up
docker-compose down
docker-compose up -d  # Run in detached mode
docker-compose logs   # View logs
```

Configuration Templates

package.json Template

```
{
  "name": "node-express-app",
  "version": "1.0.0",
  "description": "A Node.js Express application",
  "main": "src/server.js",
  "scripts": {
    "start": "node src/server.js",
    "dev": "nodemon src/server.js",
    "test": "jest --runInBand",
    "test:watch": "jest --watch",
    "test:coverage": "jest --coverage",
    "lint": "eslint .",
    "lint:fix": "eslint . --fix"
  },
  "engines": {
    "node": ">=14.0.0"
  },
  "keywords": [
    "node",
    "express",
    "mongodb",
    "api"
  ],
```

```
  "author": "Your Name",
  "license": "MIT",
  "dependencies": {
    "bcrypt": "^5.1.0",
    "compression": "^1.7.4",
    "cors": "^2.8.5",
    "dotenv": "^16.0.3",
    "express": "^4.18.2",
    "express-rate-limit": "^6.7.0",
    "express-validator": "^7.0.1",
    "helmet": "^6.1.5",
    "jsonwebtoken": "^9.0.0",
    "mongoose": "^7.0.5",
    "morgan": "^1.10.0",
    "multer": "^1.4.5-lts.1",
    "winston": "^3.8.2"
  },
  "devDependencies": {
    "eslint": "^8.39.0",
    "jest": "^29.5.0",
    "mongodb-memory-server": "^8.12.2",
    "nodemon": "^2.0.22",
    "supertest": "^6.3.3"
  }
}
```

.env Template

```
# Application
NODE_ENV=development
PORT=3000
API_PREFIX=/api

# Database
MONGODB_URI=mongodb://localhost:27017/mydatabase
MONGODB_URI_TEST=mongodb://localhost:27017/mydatabase_test

# Authentication
JWT_SECRET=your-secret-key-here
JWT_EXPIRES_IN=7d
JWT_COOKIE_EXPIRES_IN=7

# Logging
LOG_LEVEL=debug

# Security
CORS_ORIGIN=http://localhost:3000
RATE_LIMIT_WINDOW_MS=15*60*1000
RATE_LIMIT_MAX=100

# File Upload
```

```
UPLOAD_PATH=./uploads
MAX_FILE_SIZE=5000000

# Mail
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USER=your-user
MAIL_PASS=your-pass
MAIL_FROM=noreply@example.com
```

Dockerfile Template

```
FROM node:16-alpine

# Set working directory
WORKDIR /usr/src/app

# Install dependencies
COPY package*.json ./
RUN npm ci --only=production

# Copy app source
COPY . .

# Set environment variables
ENV NODE_ENV=production

# Create a non-root user and switch to it
RUN addgroup -g 1001 -S nodejs
RUN adduser -S nodeuser -u 1001
RUN chown -R nodeuser:nodejs /usr/src/app
USER nodeuser

# Expose port
EXPOSE 3000

# Start application
CMD ["node", "src/server.js"]
```

docker-compose.yml Template

```
version: '3.8'

services:
  app:
    build: .
    image: node-express-app
    container_name: node-express-app
```

```
    restart: unless-stopped
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=production
      - PORT=3000
      - MONGODB_URI=mongodb://mongodb:27017/mydatabase
      - JWT_SECRET=your-secret-key-here
    depends_on:
      - mongodb
    volumes:
      - ./uploads:/usr/src/app/uploads
      - ./logs:/usr/src/app/logs
    networks:
      - app-network

  mongodb:
    image: mongo:latest
    container_name: mongodb
    restart: unless-stopped
    ports:
      - "27017:27017"
    volumes:
      - mongodb_data:/data/db
    networks:
      - app-network

networks:
  app-network:
    driver: bridge

volumes:
  mongodb_data:
    driver: local
```

.eslintrc.js Template

```
module.exports = {
  env: {
    node: true,
    es2021: true,
    jest: true,
  },
  extends: ['eslint:recommended'],
  parserOptions: {
    ecmaVersion: 'latest',
    sourceType: 'module',
  },
  rules: {
    'no-console': process.env.NODE_ENV === 'production' ? 'warn' : 'off',
    'no-debugger': process.env.NODE_ENV === 'production' ? 'error' : 'off',
```

```
    'semi': ['error', 'always'],
    'quotes': ['error', 'single'],
    'indent': ['error', 2],
    'comma-dangle': ['error', 'always-multiline'],
    'eqeqeq': ['error', 'always'],
    'no-unused-vars': ['warn'],
    'no-var': 'error',
    'prefer-const': 'error',
  },
};
```

jest.config.js Template

```
module.exports = {
  testEnvironment: 'node',
  testMatch: ['**/__tests__/**/*.js', '**/?(*.)+(spec|test).js'],
  collectCoverageFrom: [
    'src/**/*.js',
    '!src/**/*.test.js',
    '!**/node_modules/**',
    '!**/vendor/**',
  ],
  coverageDirectory: 'coverage',
  clearMocks: true,
  testTimeout: 30000,
};
```

Project Structure Recommendations

Basic Express API Structure

```
express-api/
├── src/
│   ├── config/                # Configuration files
│   │   ├── database.js        # Database configuration
│   │   ├── logger.js          # Logging configuration
│   │   └── index.js           # Main configuration
│   ├── controllers/           # Route controllers
│   │   ├── authController.js
│   │   └── userController.js
│   ├── middleware/            # Custom middleware
│   │   ├── auth.js            # Authentication middleware
│   │   ├── errorHandler.js    # Error handling middleware
│   │   └── validator.js       # Validation middleware
│   ├── models/                # Database models
│   │   ├── User.js
│   │   └── Token.js
│   ├── routes/                # Route definitions
```

```
│   │   ├── authRoutes.js
│   │   ├── userRoutes.js
│   │   └── index.js            # Route aggregator
│   ├── services/               # Business logic
│   │   ├── authService.js
│   │   └── emailService.js
│   ├── utils/                  # Utility functions
│   │   ├── errorResponse.js
│   │   └── validation.js
│   ├── app.js                  # Express app setup
│   └── server.js               # Server entry point
├── tests/                      # Tests
│   ├── integration/            # Integration tests
│   ├── unit/                   # Unit tests
│   └── fixtures/               # Test fixtures
├── uploads/                    # File uploads
├── logs/                       # Log files
├── .env                        # Environment variables
├── .env.example                # Example environment variables
├── .eslintrc.js                # ESLint configuration
├── .gitignore                  # Git ignore file
├── jest.config.js              # Jest configuration
├── package.json                # NPM package file
└── README.md                   # Project documentation
```

Full-Stack Application Structure

```
fullstack-app/
├── backend/                    # Backend application
│   ├── src/                    # Source code
│   │   ├── config/             # Configuration
│   │   ├── controllers/        # Controllers
│   │   ├── middleware/         # Middleware
│   │   ├── models/             # Database models
│   │   ├── routes/             # Routes
│   │   ├── services/           # Services
│   │   ├── utils/              # Utilities
│   │   ├── app.js              # Express app
│   │   └── server.js           # Server entry point
│   ├── tests/                  # Tests
│   ├── .env                    # Environment variables
│   └── package.json            # Backend dependencies
├── frontend/                   # Frontend application
│   ├── public/                 # Static files
│   ├── src/                    # Source code
│   │   ├── assets/             # Assets (images, styles)
│   │   ├── components/         # React components
│   │   ├── context/            # Context providers
│   │   ├── hooks/              # Custom hooks
│   │   ├── pages/              # Page components
│   │   ├── services/           # API services
```

```
│   │   ├── utils/              # Utility functions
│   │   ├── App.js              # Main component
│   │   └── index.js            # Entry point
│   ├── .env                    # Environment variables
│   └── package.json            # Frontend dependencies
├── shared/                     # Shared code between frontend/backend
│   ├── constants/              # Shared constants
│   ├── validators/             # Shared validators
│   └── types/                  # TypeScript types/interfaces
├── .gitignore                  # Git ignore file
├── package.json                # Root package.json for scripts
├── docker-compose.yml          # Docker Compose config
└── README.md                   # Project documentation
```

Monorepo Structure with Workspace Packages

```
monorepo/
├── packages/                   # Workspace packages
│   ├── api/                    # API package
│   │   ├── src/                # Source code
│   │   ├── tests/              # Tests
│   │   └── package.json        # Package config
│   ├── web/                    # Web frontend package
│   │   ├── public/             # Static files
│   │   ├── src/                # Source code
│   │   └── package.json        # Package config
│   ├── mobile/                 # Mobile frontend package
│   │   ├── src/                # Source code
│   │   └── package.json        # Package config
│   ├── common/                 # Shared code package
│   │   ├── src/                # Source code
│   │   └── package.json        # Package config
│   └── config/                 # Shared configurations package
│       ├── eslint/             # ESLint configs
│       ├── jest/               # Jest configs
│       └── package.json        # Package config
├── .github/                    # GitHub workflows
├── .husky/                     # Git hooks
├── .gitignore                  # Git ignore file
├── lerna.json                  # Lerna config
├── package.json                # Root package.json
└── README.md                   # Project documentation
```

Further Learning Resources

Official Documentation

- Node.js Documentation - Official Node.js documentation
- Express.js Documentation - Official Express framework documentation

- MongoDB Documentation - Official MongoDB documentation
- Mongoose Documentation - Official Mongoose ODM documentation
- React Documentation - Official React documentation

Online Courses and Tutorials

- Node.js & Express From Scratch - Comprehensive Node.js course
- The Complete Node.js Developer Course - Build real-world Node.js applications
- Advanced Node.js - Advanced concepts in Node.js
- MongoDB University - Free MongoDB courses
- Full Stack Open - Free full stack JavaScript course with Node.js, Express, and React

Books

- "Node.js Design Patterns" by Mario Casciaro and Luciano Mammino
- "Express in Action" by Evan Hahn
- "MongoDB: The Definitive Guide" by Shannon Bradshaw and Kristina Chodorow
- "Web Development with Node and Express" by Ethan Brown
- "Node.js 8 the Right Way" by Jim Wilson

Advanced Topics

- Node.js Streams - Working with streaming data
- Node.js Cluster - Running Node.js on multiple cores
- MongoDB Aggregation - Advanced data processing
- Authentication Strategies - Various authentication methods
- WebSockets with Socket.io - Real-time communication
- GraphQL with Apollo Server - Building GraphQL APIs
- Microservices Architecture - Breaking down applications into services
- Docker and Kubernetes - Containerization and orchestration
- Continuous Integration/Deployment - Automated testing and deployment

Community and Support

- Stack Overflow - Q&A for Node.js
- Node.js GitHub - Source code and issues
- Express GitHub - Express framework repository
- Mongoose GitHub - Mongoose repository
- Node.js Reddit - Reddit community for Node.js
- MongoDB Reddit - Reddit community for MongoDB

Newsletters and Blogs

- Node Weekly - Weekly Node.js news and articles
- JavaScript Weekly - Weekly JavaScript news
- The Node.js Blog - Official Node.js blog
- Smashing Magazine - Web development articles
- David Walsh Blog - JavaScript and Node.js tutorials

Podcasts

- Syntax - A podcast for web developers
- JavaScript Jabber - Weekly JavaScript podcast
- NodeUp - Node.js podcast
- The Changelog - Open source and software development

This comprehensive guide should provide you with a solid foundation to re-learn and master the Node.js ecosystem, covering everything from basic concepts to advanced practices with Express, MongoDB, template engines, and modern front-end integration.