ExpressJS - The Complete Interview Guide

What is Express.js?

Express.js is a **minimal, fast, and flexible Node.js web application framework** that provides a robust set of features for web and mobile applications. It simplifies the process of building servers and APIs by providing essential tools and functionalities like routing, middleware, and template engines. Express acts as a layer built on top of Node.js that helps in managing servers and routes more efficiently.[1] [2]

Key Features:

- **Minimalist Framework**: Built with simplicity in mind, providing only essential features[3]
- **Powerful Routing System**: Handles different HTTP methods and URL patterns effectively[3]
- **Middleware Support**: Extensive middleware ecosystem for extending functionality[3]
- **Template Engine Integration**: Built-in support for Pug, EJS, Handlebars, etc.[4]
- **RESTful API Development**: Ideal for building modern APIs and web services[5]
- **Fast Performance**: Optimized for high-performance applications[3]

Core Concepts

1. Routing

What it does: Routing determines how an application responds to client requests to specific endpoints (combination of URL path and HTTP method). It's the mechanism that maps HTTP requests to appropriate handler functions.[6]

Basic Route Structure:

```
// Basic syntax for defining routes
app.METHOD(PATH, HANDLER)
// METHOD: HTTP method (get, post, put, delete, etc.)
// PATH: URL pattern to match
// HANDLER: Function to execute when route is matched
```

Comprehensive Route Examples:

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

```
// Basic GET route - handles requests to the root URL
app.get('/', (req, res) => {
    res.send('Welcome to Express.js!');
});

// GET route with static path - returns list of users
app.get('/users', (req, res) => {
    res.json([
        { id: 1, name: 'John Doe' },
        { id: 2, name: 'Jane Smith' }
    ]);
});

// POST route - creates new user (typically from form data)
app.post('/users', (req, res) => {
    const newUser = req.body; // Contains submitted form data
    // Logic to save user to database would go here
    res.status(201).json({
        message: 'User created successfully',
        user: newUser
    });
});

// Route with URL parameters - :id is a placeholder for dynamic values
app.get('/users/:id', (req, res) => {
    const userId = req.params.id; // Extracts the id from URL
    // Logic to find user by ID would go here
    res.json({
        message: `Fetching user with ID: ${userId}`,
        userId: userId
    });
});

// Route with multiple parameters - both category and productId are dynamic
app.get('/products/:category/:productId', (req, res) => {
    const { category, productId } = req.params; // Destructure multiple params
    res.json({
        category: category,
        productId: productId,
        message: `Product ${productId} in category ${category}`
    });
});

// Route with query parameters - handles ?search=laptop&limit=10
app.get('/search', (req, res) => {
    const { search, limit } = req.query; // Extract query parameters
    res.json({
        searchTerm: search,
        resultsLimit: limit || 'No limit specified',
        message: `Searching for: ${search}`
    });
});

// Wildcard route - matches any path starting with /admin/
app.get('/admin/*', (req, res) => {
```

```
    const subPath = req.params[^0]; // Captures everything after /admin/
    res.json({
        message: 'Admin panel',
        subPath: subPath || 'dashboard'
    });
});
```

Advanced Routing with app.route() - handles multiple HTTP methods for same endpoint:

```
// Chain multiple HTTP methods for the same route path
app.route('/api/books')
    .get((req, res) => {
        // Retrieve all books from database
        res.json({ message: 'Get all books', books: [] });
    })
    .post((req, res) => {
        // Create new book from request body
        const newBook = req.body;
        res.status(201).json({ message: 'Book created', book: newBook });
    })
    .put((req, res) => {
        // Update existing book
        res.json({ message: 'Book updated' });
    })
    .delete((req, res) => {
        // Delete all books (usually not recommended)
        res.json({ message: 'All books deleted' });
    });

// Route with specific ID for individual book operations
app.route('/api/books/:id')
    .get((req, res) => {
        const bookId = req.params.id;
        res.json({ message: `Get book ${bookId}` });
    })
    .put((req, res) => {
        const bookId = req.params.id;
        const updatedBook = req.body;
        res.json({ message: `Book ${bookId} updated`, book: updatedBook });
    })
    .delete((req, res) => {
        const bookId = req.params.id;
        res.json({ message: `Book ${bookId} deleted` });
    });
```

Catch-All Route (404 Handler) - must be placed at the end:

```
// This middleware catches all requests that haven't matched any previous routes
// Place this AFTER all your defined routes
app.all('*', (req, res) => {
    res.status(404).json({
        error: 'Page Not Found',
        message: `The route ${req.originalUrl} does not exist`,
        statusCode: 404
```

```
    });
});
```

2. Middleware

What it does: Middleware functions are functions that have access to the **request object (req)**, **response object (res)**, and **next middleware function (next)**. They execute during the request-response cycle and can modify request/response objects, execute code, end the request-response cycle, or call the next middleware.[7]

Middleware Execution Flow:

```
// This demonstrates how middleware flows through the application
const express = require('express');
const app = express();

// Middleware execution order demonstration
app.use((req, res, next) => {
    console.log('1st Middleware: Request received at', new Date().toISOString());
    next(); // Pass control to next middleware
});

app.use((req, res, next) => {
    console.log('2nd Middleware: Processing request');
    req.customProperty = 'Added by middleware'; // Modify request object
    next(); // Continue to next middleware
});

// Route handler (also a type of middleware)
app.get('/', (req, res) => {
    console.log('3rd: Route handler executing');
    res.json({
        message: 'Hello World!',
        customProperty: req.customProperty // Use property added by middleware
    });
});
```

Types of Middleware:

Application-Level Middleware - runs for all or specific routes:

```
// Global middleware - runs for ALL routes and HTTP methods
app.use((req, res, next) => {
    console.log(`${req.method} ${req.url} at ${Date.now()}`);
    next(); // MUST call next() to continue to next middleware
});

// Path-specific middleware - only runs for routes starting with /api
app.use('/api', (req, res, next) => {
    console.log('API endpoint accessed');
    req.isAPICall = true; // Add property to request object
    next();
```

```
});

// Method-specific middleware - only runs for POST requests
app.use((req, res, next) => {
    if (req.method === 'POST') {
        console.log('POST request detected');
    }
    next();
});
```

Router-Level Middleware - works with Express Router:

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

// Middleware that applies to all routes in this router
router.use((req, res, next) => {
    console.log('Router-level middleware executed');
    req.routerTimestamp = Date.now();
    next();
});

// Specific route with its own middleware
router.get('/profile',
    // Route-specific middleware
    (req, res, next) => {
        console.log('Profile route middleware');
        next();
    },
    // Final route handler
    (req, res) => {
        res.json({
            message: 'User profile',
            timestamp: req.routerTimestamp
        });
    }
);

// Mount the router on the main app
app.use('/user', router); // All router routes will be prefixed with /user
```

Built-in Middleware - provided by Express:

```
// Parse JSON bodies (application/json)
app.use(express.json({
    limit: '10mb', // Maximum request body size
    strict: true   // Only parse arrays and objects
}));

// Parse URL-encoded bodies (application/x-www-form-urlencoded)
app.use(express.urlencoded({
    extended: true, // Use qs library for rich objects
    limit: '10mb'
}));
```

```
// Serve static files from 'public' directory
app.use(express.static('public', {
    maxAge: '1d',        // Cache files for 1 day
    etag: false,         // Disable ETag generation
    dotfiles: 'ignore'   // Ignore files starting with dot
}));

// Serve static files with custom route prefix
app.use('/assets', express.static('public'));
```

Third-Party Middleware - external packages:

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

// Enable Cross-Origin Resource Sharing
app.use(cors({
    origin: ['http://localhost:3000', 'https://myapp.com'],
    credentials: true,    // Allow cookies
    methods: ['GET', 'POST', 'PUT', 'DELETE']
}));

// Security middleware - adds various security headers
app.use(helmet({
    contentSecurityPolicy: {
        directives: {
            defaultSrc: ["'self'"],
            scriptSrc: ["'self'", "'unsafe-inline'"]
        }
    }
}));

// HTTP request logging
app.use(morgan('combined', {
    // Only log error responses
    skip: (req, res) => res.statusCode < 400
}));

// Rate limiting middleware
const limiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes window
    max: 100,                 // Limit each IP to 100 requests per window
    message: {
        error: 'Too many requests',
        retryAfter: '15 minutes'
    }
});
app.use(limiter);
```

Custom Middleware Examples:

```
// Authentication middleware - checks if user is logged in
const authenticate = (req, res, next) => {
    const token = req.headers.authorization?.split(' ')[^1]; // Extract Bearer token

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

    // In real app, verify token with JWT library or database
    if (token === 'valid-token') {
        req.user = { id: 1, name: 'John Doe', role: 'user' };
        next(); // User authenticated, continue
    } else {
        res.status(401).json({ error: 'Invalid token' });
    }
};

// Authorization middleware - checks user permissions
const authorize = (roles = []) => {
    return (req, res, next) => {
        if (!req.user) {
            return res.status(401).json({ error: 'Authentication required' });
        }

        if (roles.length && !roles.includes(req.user.role)) {
            return res.status(403).json({ error: 'Insufficient permissions' });
        }

        next();
    };
};

// Logging middleware with detailed information
const detailedLogger = (req, res, next) => {
    const start = Date.now();

    // Log when request finishes
    res.on('finish', () => {
        const duration = Date.now() - start;
        console.log({
            method: req.method,
            url: req.url,
            statusCode: res.statusCode,
            responseTime: `${duration}ms`,
            userAgent: req.get('User-Agent'),
            ip: req.ip
        });
    });

    next();
};

// Using custom middleware
app.get('/protected', authenticate, authorize(['admin']), (req, res) => {
    res.json({
        message: 'Access granted to admin area',
```

```
        user: req.user
    });
});

// Multiple middleware for same route
app.get('/api/data',
    detailedLogger,
    authenticate,
    (req, res) => {
        res.json({ data: 'Sensitive information' });
    }
);
```

3. Error Handling

What it does: Error handling in Express provides a centralized way to catch and process errors that occur during request processing. Error-handling middleware has a special signature with four parameters and runs when errors are passed to next().[8]

Basic Error-Handling Middleware:

```
// Error handler MUST have 4 parameters (err, req, res, next)
// Place this AFTER all routes and other middleware
app.use((err, req, res, next) => {
    console.error('Error occurred:', err.stack);

    // Set default error status code if not already set
    const statusCode = err.statusCode || err.status || 500;

    // Send error response to client
    res.status(statusCode).json({
        success: false,
        error: {
            message: err.message || 'Internal Server Error',
            statusCode: statusCode,
            // Only include stack trace in development
            ...(process.env.NODE_ENV === 'development' && { stack: err.stack })
        }
    });
});
```

Custom Error Class:

```
// Create a custom error class for better error handling
class AppError extends Error {
    constructor(statusCode, message, isOperational = true) {
        super(message);
        this.statusCode = statusCode;
        this.isOperational = isOperational; // Distinguish operational vs programming err

        // Capture stack trace, excluding constructor call from it
        Error.captureStackTrace(this, this.constructor);
    }
```

```
}

// Predefined error creators for common scenarios
class BadRequestError extends AppError {
    constructor(message = 'Bad Request') {
        super(400, message);
    }
}

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

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

// Usage in routes
app.get('/users/:id', (req, res, next) => {
    const userId = req.params.id;

    try {
        // Validate input
        if (!userId || isNaN(userId)) {
            throw new BadRequestError('User ID must be a valid number');
        }

        // Simulate database lookup
        const user = findUserById(userId); // This function would query database

        if (!user) {
            throw new NotFoundError(`User with ID ${userId} not found`);
        }

        res.json(user);
    } catch (error) {
        next(error); // Pass error to error handling middleware
    }
});
```

Async Error Handling Wrapper:

```
// Utility function to handle async errors automatically
const asyncHandler = (asyncFn) => {
    return (req, res, next) => {
        // Execute async function and catch any errors
        Promise.resolve(asyncFn(req, res, next)).catch(next);
    };
};

// Alternative implementation with explicit error handling
```

```
const catchAsync = (fn) => {
    return (req, res, next) => {
        fn(req, res, next).catch((err) => {
            console.log('Async error caught:', err.message);
            next(err);
        });
    };
};

// Usage with async route handlers
app.get('/async-route', asyncHandler(async (req, res) => {
    // This async function can throw errors without manual try-catch
    const data = await fetchDataFromDatabase(); // If this rejects, asyncHandler catches

    if (!data) {
        throw new NotFoundError('Data not found');
    }

    res.json(data);
}));

// Manual async error handling (traditional approach)
app.get('/manual-async', async (req, res, next) => {
    try {
        const result = await someAsyncOperation();
        res.json(result);
    } catch (error) {
        next(error); // Must manually pass to error handler
    }
});
```

Comprehensive Error Handler:

```
// Advanced error handling middleware with different error types
const errorHandler = (err, req, res, next) => {
    let error = { ...err };
    error.message = err.message;

    // Log error details for debugging
    console.error('Error Handler:', {
        message: err.message,
        stack: err.stack,
        url: req.url,
        method: req.method,
        ip: req.ip,
        userAgent: req.get('User-Agent')
    });

    // Handle specific error types
    if (err.name === 'CastError') {
        // Database ID casting error
        const message = 'Invalid ID format';
        error = new AppError(400, message);
    } else if (err.code === 11000) {
        // MongoDB duplicate key error
```

```
        const message = 'Duplicate field value entered';
        error = new AppError(400, message);
    } else if (err.name === 'ValidationError') {
        // Mongoose validation error
        const message = Object.values(err.errors).map(val => val.message).join(', ');
        error = new AppError(400, message);
    } else if (err.name === 'JsonWebTokenError') {
        // JWT error
        const message = 'Invalid token';
        error = new AppError(401, message);
    } else if (err.name === 'TokenExpiredError') {
        // JWT expired error
        const message = 'Token expired';
        error = new AppError(401, message);
    }

    // Send error response
    res.status(error.statusCode || 500).json({
        success: false,
        error: {
            message: error.message || 'Server Error',
            statusCode: error.statusCode || 500,
            ...(process.env.NODE_ENV === 'development' && {
                stack: err.stack,
                originalError: err
            })
        }
    });
};

// Handle unhandled routes (404 errors)
app.all('*', (req, res, next) => {
    const err = new NotFoundError(`Route ${req.originalUrl} not found`);
    next(err);
});

// Use the error handler (must be last middleware)
app.use(errorHandler);
```

4. Express Router

What it does: Router creates modular, mountable route handlers that help organize your application into smaller, manageable pieces. It acts like a mini-application capable of performing middleware and routing functions.[6]

Basic Router Setup:

```
// routes/users.js - Separate file for user-related routes
const express = require('express');
const router = express.Router();

// Middleware specific to this router - runs for ALL routes in this file
router.use((req, res, next) => {
    console.log('User routes middleware executed at:', new Date().toISOString());
```

```
    req.routeType = 'user'; // Add custom property
    next();
});

// Route handlers
router.get('/', (req, res) => {
    res.json({
        message: 'All users endpoint',
        routeType: req.routeType,
        totalUsers: 150
    });
});

router.get('/:id', (req, res) => {
    const userId = req.params.id;
    res.json({
        message: `User details for ID: ${userId}`,
        userId: parseInt(userId),
        user: {
            id: userId,
            name: 'John Doe',
            email: 'john@example.com'
        }
    });
});

router.post('/', (req, res) => {
    const userData = req.body;
    res.status(201).json({
        message: 'User created successfully',
        user: userData,
        id: Date.now() // Simulate generated ID
    });
});

router.put('/:id', (req, res) => {
    const userId = req.params.id;
    const updateData = req.body;
    res.json({
        message: `User ${userId} updated`,
        updatedFields: updateData
    });
});

router.delete('/:id', (req, res) => {
    const userId = req.params.id;
    res.json({
        message: `User ${userId} deleted successfully`,
        deletedAt: new Date().toISOString()
    });
});

// Export router to use in main app
module.exports = router;
```

Using Router in Main Application:

```
// app.js or server.js - Main application file
const express = require('express');
const app = express();

// Import route modules
const userRoutes = require('./routes/users');
const productRoutes = require('./routes/products');
const authRoutes = require('./routes/auth');

// Body parsing middleware
app.use(express.json());

// Mount routers with prefixes
app.use('/api/users', userRoutes);     // All user routes prefixed with /api/users
app.use('/api/products', productRoutes); // All product routes prefixed with /api/product
app.use('/auth', authRoutes);          // All auth routes prefixed with /auth

// Example of mounting router with middleware
app.use('/admin', authenticateAdmin, adminRoutes);

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

Advanced Router with Middleware Chain:

```
// routes/products.js - Product routes with various middleware
const express = require('express');
const router = express.Router();

// Route-specific middleware functions
const validateProduct = (req, res, next) => {
    const { name, price } = req.body;

    if (!name) {
        return res.status(400).json({ error: 'Product name is required' });
    }

    if (price && (isNaN(price) || price < 0)) {
        return res.status(400).json({ error: 'Price must be a positive number' });
    }

    next();
};

const logProductAccess = (req, res, next) => {
    console.log(`Product ${req.method} operation at ${new Date().toISOString()}`);
    next();
};

// Routes with multiple middleware
router.route('/')
    .get(logProductAccess, (req, res) => {
```

```
        // Get all products
        res.json({
            products: [
                { id: 1, name: 'Laptop', price: 999.99 },
                { id: 2, name: 'Phone', price: 599.99 }
            ],
            total: 2
        });
    })
    .post(logProductAccess, validateProduct, (req, res) => {
        // Create new product
        const productData = req.body;
        res.status(201).json({
            message: 'Product created',
            product: { id: Date.now(), ...productData }
        });
    });

// Parameterized routes with middleware
router.route('/:id')
    .get(logProductAccess, (req, res) => {
        const productId = req.params.id;
        res.json({
            product: {
                id: productId,
                name: 'Sample Product',
                price: 299.99
            }
        });
    })
    .put(logProductAccess, validateProduct, (req, res) => {
        const productId = req.params.id;
        const updateData = req.body;
        res.json({
            message: `Product ${productId} updated`,
            updatedData: updateData
        });
    })
    .delete(logProductAccess, (req, res) => {
        const productId = req.params.id;
        res.json({ message: `Product ${productId} deleted` });
    });

module.exports = router;
```

5. File Upload with Multer

What it does: Multer is middleware for handling `multipart/form-data`, primarily used for uploading files. It processes file uploads and makes them available on the request object, while also parsing text fields from forms.[9]

Basic File Upload Setup:

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

const app = express();

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

// Basic multer configuration - saves files to uploads folder
const upload = multer({
    dest: uploadDir, // Temporary storage location
    limits: {
        fileSize: 5 * 1024 * 1024 // 5MB file size limit
    }
});

// Single file upload endpoint
app.post('/upload-single', upload.single('avatar'), (req, res) => {
    try {
        // Check if file was uploaded
        if (!req.file) {
            return res.status(400).json({
                success: false,
                message: 'No file uploaded'
            });
        }

        // File information is available in req.file
        const fileInfo = {
            originalName: req.file.originalname,
            filename: req.file.filename,
            size: req.file.size,
            mimetype: req.file.mimetype,
            path: req.file.path
        };

        res.json({
            success: true,
            message: 'File uploaded successfully',
            file: fileInfo
        });
    } catch (error) {
        res.status(500).json({
            success: false,
            message: 'Upload failed',
            error: error.message
        });
    }
});
```

Advanced File Upload with Custom Storage:

```
// Custom storage configuration with file naming and destination control
const storage = multer.diskStorage({
    // Specify destination folder
    destination: (req, file, cb) => {
        const uploadPath = path.join(__dirname, 'uploads', file.fieldname);

        // Create directory if it doesn't exist
        if (!fs.existsSync(uploadPath)) {
            fs.mkdirSync(uploadPath, { recursive: true });
        }

        cb(null, uploadPath);
    },

    // Specify filename format
    filename: (req, file, cb) => {
        // Generate unique filename: timestamp-originalname
        const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1E9);
        const fileExtension = path.extname(file.originalname);
        const baseName = path.basename(file.originalname, fileExtension);

        cb(null, `${baseName}-${uniqueSuffix}${fileExtension}`);
    }
});

// File filter function to restrict file types
const fileFilter = (req, file, cb) => {
    // Define allowed file types
    const allowedTypes = ['image/jpeg', 'image/png', 'image/gif', 'application/pdf'];

    if (allowedTypes.includes(file.mimetype)) {
        cb(null, true); // Accept the file
    } else {
        cb(new Error(`File type ${file.mimetype} not allowed`), false);
    }
};

// Advanced multer configuration
const advancedUpload = multer({
    storage: storage,
    limits: {
        fileSize: 10 * 1024 * 1024, // 10MB limit
        files: 5 // Maximum 5 files
    },
    fileFilter: fileFilter
});

// Multiple files upload (same field name)
app.post('/upload-multiple', advancedUpload.array('documents', 5), (req, res) => {
    try {
        if (!req.files || req.files.length === 0) {
            return res.status(400).json({
                success: false,
                message: 'No files uploaded'
```

```
            });
        }

        // Process multiple files
        const filesInfo = req.files.map(file => ({
            originalName: file.originalname,
            filename: file.filename,
            size: file.size,
            mimetype: file.mimetype,
            path: file.path
        }));

        res.json({
            success: true,
            message: `${req.files.length} files uploaded successfully`,
            files: filesInfo,
            totalSize: req.files.reduce((total, file) => total + file.size, 0)
        });
    } catch (error) {
        res.status(500).json({
            success: false,
            message: 'Upload failed',
            error: error.message
        });
    }
});

// Mixed fields upload (different field names)
app.post('/upload-mixed', advancedUpload.fields([
    { name: 'avatar', maxCount: 1 },
    { name: 'documents', maxCount: 3 }
]), (req, res) => {
    try {
        const response = {
            success: true,
            message: 'Files uploaded successfully',
            files: {}
        };

        // Handle avatar file
        if (req.files.avatar) {
            response.files.avatar = {
                originalName: req.files.avatar[^0].originalname,
                filename: req.files.avatar[^0].filename,
                size: req.files.avatar[^0].size
            };
        }

        // Handle document files
        if (req.files.documents) {
            response.files.documents = req.files.documents.map(file => ({
                originalName: file.originalname,
                filename: file.filename,
                size: file.size
            }));
        }
```

```
        // Include form text fields
        if (req.body) {
            response.formData = req.body;
        }

        res.json(response);
    } catch (error) {
        res.status(500).json({
            success: false,
            message: 'Upload failed',
            error: error.message
        });
    }
});
```

6. Input Validation and Sanitization

What it does: Validation ensures data meets specific criteria, while sanitization cleans data to prevent security vulnerabilities like XSS and SQL injection. This is crucial for application security and data integrity.[10]

Express-Validator Setup:

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

app.use(express.json());

// Validation middleware to check for errors
const handleValidationErrors = (req, res, next) => {
    const errors = validationResult(req);

    if (!errors.isEmpty()) {
        return res.status(400).json({
            success: false,
            message: 'Validation failed',
            errors: errors.array().map(error => ({
                field: error.path,
                message: error.msg,
                value: error.value
            }))
        });
    }

    next(); // Continue if no validation errors
};

// User registration with comprehensive validation
app.post('/register',
    [
        // Email validation and sanitization
        body('email')
```

```
        .isEmail()
        .withMessage('Must be a valid email address')
        .normalizeEmail() // Converts to lowercase, removes dots from Gmail
        .custom(async (email) => {
            // Custom validation - check if email already exists
            // const existingUser = await User.findOne({ email });
            // if (existingUser) {
            //     throw new Error('Email already registered');
            // }
            return true;
        }),

    // Password validation
    body('password')
        .isLength({ min: 8, max: 128 })
        .withMessage('Password must be between 8-128 characters')
        .matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]/)
        .withMessage('Password must contain uppercase, lowercase, number, and special

    // Name validation and sanitization
    body('name')
        .trim() // Remove whitespace from start/end
        .isLength({ min: 2, max: 50 })
        .withMessage('Name must be 2-50 characters long')
        .matches(/^[a-zA-Z\s]+$/)
        .withMessage('Name can only contain letters and spaces')
        .escape(), // Convert HTML entities

    // Age validation
    body('age')
        .optional() // Field is optional
        .isInt({ min: 13, max: 120 })
        .withMessage('Age must be between 13-120')
        .toInt(), // Convert to integer

    // Phone number validation
    body('phone')
        .optional()
        .isMobilePhone('any', { strictMode: false })
        .withMessage('Must be a valid phone number'),

    // Date of birth validation
    body('dateOfBirth')
        .optional()
        .isISO8601()
        .withMessage('Date of birth must be in YYYY-MM-DD format')
        .toDate() // Convert to Date object
        .custom((date) => {
            if (date > new Date()) {
                throw new Error('Date of birth cannot be in the future');
            }
            return true;
        }),

    // Terms acceptance
    body('acceptTerms')
```

```
            .isBoolean()
            .withMessage('Terms acceptance must be true or false')
            .custom((value) => {
                if (!value) {
                    throw new Error('You must accept the terms and conditions');
                }
                return true;
            })
    ],
    handleValidationErrors,
    (req, res) => {
        // If we reach here, validation passed
        const { email, password, name, age, phone, dateOfBirth } = req.body;

        res.status(201).json({
            success: true,
            message: 'User registered successfully',
            user: {
                email,
                name,
                age,
                phone,
                dateOfBirth: dateOfBirth?.toISOString?.() || null
            }
        });
    }
);
```

Parameter and Query Validation:

```
// URL parameter validation
app.get('/users/:id',
    [
        param('id')
            .isMongoId() // For MongoDB ObjectId
            .withMessage('User ID must be a valid MongoDB ObjectId')
            // Alternative for numeric IDs:
            // .isInt({ min: 1 })
            // .withMessage('User ID must be a positive integer')
    ],
    handleValidationErrors,
    (req, res) => {
        const userId = req.params.id;
        res.json({
            message: `Fetching user with ID: ${userId}`,
            userId
        });
    }
);

// Query parameter validation for search/filtering
app.get('/products',
    [
        query('page')
            .optional()
```

```
            .isInt({ min: 1 })
            .withMessage('Page must be a positive integer')
            .toInt(),

        query('limit')
            .optional()
            .isInt({ min: 1, max: 100 })
            .withMessage('Limit must be between 1-100')
            .toInt(),

        query('category')
            .optional()
            .trim()
            .isLength({ min: 2, max: 30 })
            .withMessage('Category must be 2-30 characters')
            .isAlphanumeric()
            .withMessage('Category must contain only letters and numbers'),

        query('minPrice')
            .optional()
            .isFloat({ min: 0 })
            .withMessage('Minimum price must be non-negative')
            .toFloat(),

        query('maxPrice')
            .optional()
            .isFloat({ min: 0 })
            .withMessage('Maximum price must be non-negative')
            .toFloat()
            .custom((value, { req }) => {
                if (req.query.minPrice && value < parseFloat(req.query.minPrice)) {
                    throw new Error('Maximum price must be greater than minimum price');
                }
                return true;
            }),

        query('sort')
            .optional()
            .isIn(['name', 'price', 'date', '-name', '-price', '-date'])
            .withMessage('Sort must be one of: name, price, date (prefix with - for desce
    ],
    handleValidationErrors,
    (req, res) => {
        const { page = 1, limit = 10, category, minPrice, maxPrice, sort } = req.query;

        res.json({
            message: 'Products retrieved successfully',
            filters: {
                page,
                limit,
                category,
                priceRange: { min: minPrice, max: maxPrice },
                sortBy: sort
            },
            products: [] // Would contain actual products from database
        });
```

```
    }
);
```

Sanitization Examples:

```
// Content sanitization for blog posts or comments
app.post('/blog-post',
    [
        body('title')
            .trim()
            .isLength({ min: 5, max: 200 })
            .withMessage('Title must be 5-200 characters')
            .escape(), // Escape HTML entities

        body('content')
            .trim()
            .isLength({ min: 10, max: 10000 })
            .withMessage('Content must be 10-10000 characters')
            .customSanitizer((value) => {
                // Remove script tags and dangerous HTML
                return value.replace(/<script\b[^<]*(?:(?!<\/script>)<[^<]*)*<\/script>/g
            }),

        body('tags')
            .optional()
            .isArray({ max: 10 })
            .withMessage('Maximum 10 tags allowed')
            .customSanitizer((tags) => {
                // Sanitize each tag
                return tags.map(tag =>
                    tag.toString().trim().toLowerCase().replace(/[^a-z0-9]/g, '')
                );
            }),

        body('publishedAt')
            .optional()
            .isISO8601()
            .withMessage('Published date must be in ISO format')
            .toDate()
    ],
    handleValidationErrors,
    (req, res) => {
        const { title, content, tags, publishedAt } = req.body;

        res.status(201).json({
            success: true,
            message: 'Blog post created',
            post: {
                title,
                content: content.substring(0, 100) + '...', // Preview
                tags,
                publishedAt,
                createdAt: new Date()
            }
        });
```

```
    }
);
```

7. Template Engines

What it does: Template engines allow server-side rendering (SSR) by combining HTML templates with dynamic data. They enable you to generate dynamic HTML pages on the server before sending them to the client.[4]

EJS (Embedded JavaScript) Setup:

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

// Set EJS as the template engine
app.set('view engine', 'ejs');
app.set('views', './views'); // Directory containing template files

// Body parsing middleware
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Serve static files (CSS, JS, images)
app.use(express.static('public'));

// Home page route with dynamic data
app.get('/', (req, res) => {
    // Sample data that would come from database
    const pageData = {
        title: 'Welcome to My Express App',
        message: 'Hello from Express.js!',
        currentYear: new Date().getFullYear(),
        users: [
            { id: 1, name: 'John Doe', role: 'Admin' },
            { id: 2, name: 'Jane Smith', role: 'User' },
            { id: 3, name: 'Bob Johnson', role: 'Moderator' }
        ],
        isLoggedIn: true,
        userRole: 'Admin'
    };

    // Render the 'index.ejs' template with data
    res.render('index', pageData);
});

// User profile page with dynamic user ID
app.get('/profile/:id', (req, res) => {
    const userId = req.params.id;

    // Simulate fetching user data
    const userData = {
        title: `User Profile - ${userId}`,
        user: {
            id: userId,
```

```
            name: 'John Doe',
            email: 'john@example.com',
            joinDate: '2023-01-15',
            posts: 25,
            followers: 150
        },
        recentPosts: [
            { title: 'My First Post', date: '2024-01-20' },
            { title: 'Learning Express.js', date: '2024-01-18' }
        ]
    };

    res.render('profile', userData);
});
```

EJS Template Examples:

views/index.ejs - Main template:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title><%= title %></title> <!-- Dynamic title -->
    <link rel="stylesheet" href="/css/style.css">
</head>
<body>
    <!-- Include header partial -->
    <%- include('partials/header', { userRole, isLoggedIn }) %>

    <main>
        <h1><%= message %></h1>

        <!-- Conditional rendering -->
        <% if (isLoggedIn) { %>
            <p>Welcome back! You are logged in as: <strong><%= userRole %></strong></p>
        <% } else { %>
            <p><a href="/login">Please log in</a></p>
        <% } %>

        <!-- Loop through users array -->
        <h2>Users List:</h2>
        <% if (users && users.length > 0) { %>
            <ul class="users-list">
                <% users.forEach(function(user) { %>
                    <li>
                        <strong><%= user.name %></strong> -
                        <span class="role"><%= user.role %></span>
                        <a href="/profile/<%= user.id %>">View Profile</a>
                    </li>
                <% }); %>
            </ul>
        <% } else { %>
            <p>No users found.</p>
```

```
        <% } %>

        <!-- Using JavaScript expressions -->
        <p>Total users: <%= users.length %></p>
        <p>Admin users: <%= users.filter(user => user.role === 'Admin').length %></p>
    </main>

    <!-- Include footer partial -->
    <%- include('partials/footer', { currentYear }) %>

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

views/profile.ejs - User profile template:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title><%= title %></title>
</head>
<body>
    <div class="container">
        <h1>User Profile</h1>

        <!-- User information -->
        <div class="user-info">
            <h2><%= user.name %></h2>
            <p><strong>Email:</strong> <%= user.email %></p>
            <p><strong>Joined:</strong> <%= new Date(user.joinDate).toDateString() %></p>

            <!-- User statistics -->
            <div class="stats">
                <div class="stat">
                    <span class="number"><%= user.posts %></span>
                    <span class="label">Posts</span>
                </div>
                <div class="stat">
                    <span class="number"><%= user.followers %></span>
                    <span class="label">Followers</span>
                </div>
            </div>
        </div>

        <!-- Recent posts -->
        <div class="recent-posts">
            <h3>Recent Posts</h3>
            <% if (recentPosts && recentPosts.length > 0) { %>
                <% recentPosts.forEach(function(post) { %>
                    <div class="post">
                        <h4><%= post.title %></h4>
                        <p class="date"><%= new Date(post.date).toDateString() %></p>
                    </div>
```

```
                <% }); %>
            <% } else { %>
                <p>No recent posts.</p>
            <% } %>
        </div>

        <a href="/" class="back-link">← Back to Home</a>
    </div>
</body>
</html>
```

Partial Templates:

views/partials/header.ejs:

```
<header class="main-header">
    <nav>
        <div class="logo">
            <a href="/">My App</a>
        </div>
        <ul class="nav-links">
            <li><a href="/">Home</a></li>
            <li><a href="/about">About</a></li>
            <% if (isLoggedIn) { %>
                <li><a href="/dashboard">Dashboard</a></li>
                <% if (userRole === 'Admin') { %>
                    <li><a href="/admin">Admin Panel</a></li>
                <% } %>
                <li><a href="/logout">Logout</a></li>
            <% } else { %>
                <li><a href="/login">Login</a></li>
                <li><a href="/register">Register</a></li>
            <% } %>
        </ul>
    </nav>
</header>
```

views/partials/footer.ejs:

```
<footer class="main-footer">
    <p>&copy; <%= currentYear %> My Express App. All rights reserved.</p>
    <p>Built with Express.js and EJS</p>
</footer>
```

8. Sessions and Cookies

What it does: Sessions store user data on the server with only a session ID sent to the client via cookies. Cookies store data directly on the client browser. Both are essential for user authentication and maintaining state in web applications.[11]

Cookie Management Setup:

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

// Enable cookie parsing - allows access to req.cookies
app.use(cookieParser('your-secret-key')); // Secret key for signed cookies

// Route to set various types of cookies
app.get('/set-cookies', (req, res) => {
    // Basic cookie - expires when browser closes
    res.cookie('username', 'john_doe');

    // Cookie with expiration time (7 days from now)
    res.cookie('theme', 'dark', {
        maxAge: 7 * 24 * 60 * 60 * 1000, // 7 days in milliseconds
        httpOnly: false // Accessible via JavaScript on client
    });

    // Secure cookie with various options
    res.cookie('session_token', 'abc123xyz789', {
        maxAge: 24 * 60 * 60 * 1000,    // 24 hours
        httpOnly: true,                  // Not accessible via JavaScript (XSS protection
        secure: process.env.NODE_ENV === 'production', // Only sent over HTTPS in product
        sameSite: 'strict',             // CSRF protection
        signed: true                    // Sign the cookie with secret key
    });

    // Cookie with domain and path restrictions
    res.cookie('language', 'en', {
        domain: '.example.com',         // Available to all subdomains
        path: '/admin',                 // Only sent for paths starting with /admin
        maxAge: 30 * 24 * 60 * 60 * 1000 // 30 days
    });

    res.json({
        message: 'Cookies set successfully',
        cookiesSet: ['username', 'theme', 'session_token', 'language']
    });
});

// Route to read cookies
app.get('/get-cookies', (req, res) => {
    // Access regular cookies
    const username = req.cookies.username;
    const theme = req.cookies.theme;

    // Access signed cookies
    const sessionToken = req.signedCookies.session_token;

    res.json({
        message: 'Cookies retrieved',
        cookies: {
            username: username || 'Not set',
            theme: theme || 'Not set',
            sessionToken: sessionToken || 'Not set or invalid signature'
        },
```

```
        allCookies: req.cookies,
        allSignedCookies: req.signedCookies
    });
});

// Route to clear specific cookie
app.get('/clear-cookie/:name', (req, res) => {
    const cookieName = req.params.name;

    // Clear cookie by setting it with past expiration
    res.clearCookie(cookieName);

    res.json({
        message: `Cookie '${cookieName}' cleared successfully`
    });
});

// Route to clear all cookies
app.get('/clear-all-cookies', (req, res) => {
    // Clear all cookies by iterating through them
    Object.keys(req.cookies).forEach(cookieName => {
        res.clearCookie(cookieName);
    });

    // Clear signed cookies too
    Object.keys(req.signedCookies).forEach(cookieName => {
        res.clearCookie(cookieName);
    });

    res.json({ message: 'All cookies cleared' });
});
```

Session Management Setup:

```
const session = require('express-session');
const MongoStore = require('connect-mongo'); // For MongoDB session store

// Session configuration
const sessionConfig = {
    secret: process.env.SESSION_SECRET || 'your-super-secret-key',
    name: 'sessionId', // Name of session ID cookie (default: 'connect.sid')
    resave: false,     // Don't save session if unmodified
    saveUninitialized: false, // Don't create session until something stored

    cookie: {
        secure: process.env.NODE_ENV === 'production', // Require HTTPS in production
        httpOnly: true,    // Prevent XSS attacks
        maxAge: 24 * 60 * 60 * 1000, // 24 hours
        sameSite: 'strict' // CSRF protection
    },

    // Use MongoDB to store sessions (optional - defaults to memory store)
    store: MongoStore.create({
        mongoUrl: process.env.MONGODB_URI || 'mongodb://localhost:27017/myapp',
        collectionName: 'sessions',
```

```
        ttl: 24 * 60 * 60 // Session TTL in seconds (24 hours)
    })
};

app.use(session(sessionConfig));

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

    // In real app, verify credentials against database
    if (username === 'admin' && password === 'password') {
        // Store user data in session
        req.session.user = {
            id: 1,
            username: username,
            role: 'admin',
            loginTime: new Date()
        };

        // Mark session as authenticated
        req.session.isAuthenticated = true;

        // Session data
        req.session.visits = (req.session.visits || 0) + 1;

        res.json({
            success: true,
            message: 'Login successful',
            user: req.session.user,
            sessionId: req.sessionID,
            visits: req.session.visits
        });
    } else {
        res.status(401).json({
            success: false,
            message: 'Invalid credentials'
        });
    }
});

// Profile route - requires authentication
app.get('/profile', (req, res) => {
    // Check if user is authenticated via session
    if (!req.session.isAuthenticated) {
        return res.status(401).json({
            success: false,
            message: 'Please log in to access profile'
        });
    }

    // Update session activity
    req.session.lastAccess = new Date();

    res.json({
        success: true,
```

```
        message: 'Profile data retrieved',
        user: req.session.user,
        sessionInfo: {
            sessionId: req.sessionID,
            visits: req.session.visits,
            loginTime: req.session.user.loginTime,
            lastAccess: req.session.lastAccess
        }
    });
});

// Session data manipulation
app.post('/update-preferences', (req, res) => {
    if (!req.session.isAuthenticated) {
        return res.status(401).json({ message: 'Authentication required' });
    }

    const { theme, language, notifications } = req.body;

    // Store preferences in session
    req.session.preferences = {
        theme: theme || 'light',
        language: language || 'en',
        notifications: notifications !== false
    };

    res.json({
        success: true,
        message: 'Preferences updated',
        preferences: req.session.preferences
    });
});

// Shopping cart example using sessions
app.post('/cart/add', (req, res) => {
    const { productId, quantity = 1 } = req.body;

    // Initialize cart if it doesn't exist
    if (!req.session.cart) {
        req.session.cart = [];
    }

    // Check if product already in cart
    const existingItem = req.session.cart.find(item => item.productId === productId);

    if (existingItem) {
        existingItem.quantity += quantity;
    } else {
        req.session.cart.push({
            productId,
            quantity,
            addedAt: new Date()
        });
    }

    res.json({
```

```
        success: true,
        message: 'Product added to cart',
        cart: req.session.cart,
        cartTotal: req.session.cart.reduce((total, item) => total + item.quantity, 0)
    });
});

// Get session information
app.get('/session-info', (req, res) => {
    res.json({
        sessionId: req.sessionID,
        isAuthenticated: req.session.isAuthenticated || false,
        user: req.session.user || null,
        cart: req.session.cart || [],
        preferences: req.session.preferences || {},
        visits: req.session.visits || 0,
        cookie: req.session.cookie
    });
});

// Logout route - destroys session
app.post('/logout', (req, res) => {
    if (!req.session.isAuthenticated) {
        return res.json({ message: 'No active session to logout' });
    }

    const username = req.session.user?.username;

    // Destroy session
    req.session.destroy((err) => {
        if (err) {
            console.error('Session destruction error:', err);
            return res.status(500).json({
                success: false,
                message: 'Logout failed'
            });
        }

        // Clear session cookie
        res.clearCookie('sessionId');

        res.json({
            success: true,
            message: `Goodbye ${username}! Logged out successfully`
        });
    });
});
```

9. Logging with Winston and Morgan

What it does: Logging is crucial for monitoring application behavior, debugging issues, and maintaining audit trails. Morgan handles HTTP request logging while Winston provides comprehensive logging capabilities with multiple transports and formats.[12]

Winston Logger Configuration:

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

// Custom log format
const logFormat = winston.format.combine(
    winston.format.timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }),
    winston.format.errors({ stack: true }), // Include stack traces for errors
    winston.format.json(), // JSON format for structured logging
    winston.format.prettyPrint() // Pretty print for readability
);

// Create logs directory if it doesn't exist
const fs = require('fs');
const logDir = 'logs';
if (!fs.existsSync(logDir)) {
    fs.mkdirSync(logDir);
}

// Create Winston logger instance
const logger = winston.createLogger({
    level: process.env.LOG_LEVEL || 'info', // Default log level
    format: logFormat,

    // Default metadata to include with every log
    defaultMeta: {
        service: 'express-app',
        version: '1.0.0'
    },

    // Define transports (where logs go)
    transports: [
        // Write all logs with level 'error' and below to error.log
        new winston.transports.File({
            filename: path.join(logDir, 'error.log'),
            level: 'error',
            maxsize: 5242880, // 5MB max file size
            maxFiles: 5,      // Keep last 5 files
        }),

        // Write all logs to combined.log
        new winston.transports.File({
            filename: path.join(logDir, 'combined.log'),
            maxsize: 5242880,
            maxFiles: 5,
        }),

        // Console transport for development
        ...(process.env.NODE_ENV !== 'production' ? [
            new winston.transports.Console({
                format: winston.format.combine(
                    winston.format.colorize(),
                    winston.format.simple()
                )
            })
        ] : [])
    ],
```

```
    // Handle uncaught exceptions
    exceptionHandlers: [
        new winston.transports.File({
            filename: path.join(logDir, 'exceptions.log')
        })
    ],

    // Handle unhandled promise rejections
    rejectionHandlers: [
        new winston.transports.File({
            filename: path.join(logDir, 'rejections.log')
        })
    ]
});

// Export logger for use in other modules
module.exports = logger;
```

Morgan HTTP Request Logging:

```
const express = require('express');
const morgan = require('morgan');
const logger = require('./logger'); // Winston logger from above

const app = express();

// Custom token for Morgan - adds response time in different format
morgan.token('response-time-ms', (req, res) => {
    return `${Math.round(parseFloat(morgan['response-time'](req, res)))}ms`;
});

// Custom token for request ID (useful for tracking)
morgan.token('request-id', (req, res) => {
    return req.id || 'no-id';
});

// Create a write stream for Morgan to use Winston logger
const morganStream = {
    write: (message) => {
        // Remove trailing newline and log with Winston
        logger.http(message.trim());
    }
};

// Custom Morgan format with detailed information
const morganFormat = ':request-id :method :url :status :res[content-length] - :response-t

// Apply Morgan middleware
app.use(morgan(morganFormat, { stream: morganStream }));

// Middleware to add request ID for tracking
app.use((req, res, next) => {
    req.id = Date.now().toString() + Math.random().toString(36).substr(2, 9);
    next();
```

```
});

// Example routes with different log levels
app.get('/', (req, res) => {
    logger.info('Home page accessed', {
        requestId: req.id,
        userAgent: req.get('User-Agent'),
        ip: req.ip
    });

    res.json({ message: 'Welcome to the API' });
});

app.get('/users', (req, res) => {
    logger.info('Users endpoint accessed', {
        requestId: req.id,
        query: req.query
    });

    // Simulate user data
    const users = [
        { id: 1, name: 'John Doe' },
        { id: 2, name: 'Jane Smith' }
    ];

    logger.debug('User data retrieved', {
        userCount: users.length,
        requestId: req.id
    });

    res.json(users);
});

// Route that demonstrates error logging
app.get('/error-demo', (req, res) => {
    try {
        // Simulate an error
        throw new Error('Simulated application error');
    } catch (error) {
        logger.error('Error in error-demo route', {
            error: error.message,
            stack: error.stack,
            requestId: req.id,
            url: req.url,
            method: req.method,
            ip: req.ip
        });

        res.status(500).json({
            error: 'Internal server error',
            requestId: req.id
        });
    }
});

// Route with warning logs
```

```
app.get('/deprecated-endpoint', (req, res) => {
    logger.warn('Deprecated endpoint accessed', {
        endpoint: req.originalUrl,
        requestId: req.id,
        userAgent: req.get('User-Agent'),
        message: 'This endpoint will be removed in v2.0'
    });

    res.json({
        message: 'This endpoint is deprecated',
        alternative: '/api/v2/new-endpoint'
    });
});

// Global error handler with logging
app.use((err, req, res, next) => {
    // Log the error with full context
    logger.error('Unhandled error occurred', {
        error: {
            message: err.message,
            stack: err.stack,
            name: err.name
        },
        request: {
            id: req.id,
            method: req.method,
            url: req.originalUrl,
            headers: req.headers,
            body: req.body,
            query: req.query,
            params: req.params
        },
        user: req.user || 'unauthenticated'
    });

    res.status(err.statusCode || 500).json({
        success: false,
        error: process.env.NODE_ENV === 'production'
            ? 'Internal server error'
            : err.message,
        requestId: req.id
    });
});

// Log server startup
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    logger.info('Server started successfully', {
        port: PORT,
        environment: process.env.NODE_ENV || 'development',
        nodeVersion: process.version,
        timestamp: new Date().toISOString()
    });
});
```

Advanced Logging Patterns:

```
// Create child loggers with specific context
const createChildLogger = (context) => {
    return logger.child(context);
};

// Database operations logger
const dbLogger = createChildLogger({ component: 'database' });

// Authentication logger
const authLogger = createChildLogger({ component: 'authentication' });

// Example usage in routes
app.post('/login', (req, res) => {
    const { username } = req.body;

    authLogger.info('Login attempt', {
        username,
        ip: req.ip,
        userAgent: req.get('User-Agent')
    });

    // Simulate authentication logic
    if (username === 'admin') {
        authLogger.info('Login successful', { username });
        res.json({ success: true });
    } else {
        authLogger.warn('Login failed - invalid credentials', { username });
        res.status(401).json({ success: false });
    }
});

// Database operation example
app.get('/users/:id', async (req, res) => {
    const userId = req.params.id;

    try {
        dbLogger.info('Database query initiated', {
            query: 'findUserById',
            userId
        });

        // Simulate database operation
        const startTime = Date.now();
        // const user = await User.findById(userId);
        const queryTime = Date.now() - startTime;

        dbLogger.info('Database query completed', {
            query: 'findUserById',
            userId,
            executionTime: `${queryTime}ms`,
            success: true
        });

        res.json({ user: { id: userId, name: 'John Doe' } });
    } catch (error) {
        dbLogger.error('Database query failed', {
```

```
            query: 'findUserById',
            userId,
            error: error.message
        });

        res.status(500).json({ error: 'Database error' });
    }
});
```

10. Performance Optimization and Clustering

What it does: Clustering allows Node.js applications to utilize multiple CPU cores by creating worker processes. This dramatically improves performance and handles more concurrent requests by distributing the load across multiple processes.[13]

Basic Clustering Setup:

```
// cluster.js - Main clustering file
const cluster = require('cluster');
const os = require('os');
const path = require('path');

// Get number of CPU cores available
const numCPUs = os.cpus().length;

// Check if current process is the master/primary process
if (cluster.isMaster || cluster.isPrimary) {
    console.log(`Master process ${process.pid} is running`);
    console.log(`Starting ${numCPUs} worker processes...`);

    // Track worker performance
    const workerStats = {};

    // Fork workers for each CPU core
    for (let i = 0; i < numCPUs; i++) {
        const worker = cluster.fork();

        // Track worker statistics
        workerStats[worker.process.pid] = {
            startTime: new Date(),
            restarts: 0
        };

        console.log(`Worker ${worker.process.pid} started`);
    }

    // Listen for worker exit events
    cluster.on('exit', (worker, code, signal) => {
        console.log(`Worker ${worker.process.pid} died with code ${code} and signal ${sig

        // Track restart count
        const stats = workerStats[worker.process.pid];
        if (stats) {
            stats.restarts++;
```

```
        }

        // Restart the worker
        console.log('Starting a new worker...');
        const newWorker = cluster.fork();

        // Update stats for new worker
        workerStats[newWorker.process.pid] = {
            startTime: new Date(),
            restarts: stats ? stats.restarts : 0
        };
    });

    // Handle graceful shutdown
    process.on('SIGTERM', () => {
        console.log('Master process received SIGTERM, shutting down gracefully...');

        for (const id in cluster.workers) {
            cluster.workers[id].kill();
        }

        process.exit(0);
    });

    // Display worker statistics every 30 seconds
    setInterval(() => {
        console.log('\n=== Worker Statistics ===');
        Object.entries(workerStats).forEach(([pid, stats]) => {
            const uptime = Math.floor((new Date() - stats.startTime) / 1000);
            console.log(`Worker ${pid}: Uptime ${uptime}s, Restarts: ${stats.restarts}`);
        });
        console.log('========================\n');
    }, 30000);

} else {
    // Worker process - run the Express application
    require('./app.js');

    console.log(`Worker ${process.pid} started and listening`);
}
```

Express App with Performance Optimizations:

```
// app.js - Express application with performance optimizations
const express = require('express');
const compression = require('compression');
const helmet = require('helmet');
const rateLimit = require('express-rate-limit');
const slowDown = require('express-slow-down');
const redis = require('redis');
const session = require('express-session');
const RedisStore = require('connect-redis')(session);

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

```
// Redis client for caching and session storage
const redisClient = redis.createClient({
    host: process.env.REDIS_HOST || 'localhost',
    port: process.env.REDIS_PORT || 6379
});

// Performance Middleware
// 1. Compression - reduces response size
app.use(compression({
    level: 6, // Compression level (1-9, 6 is good balance)
    threshold: 1024, // Only compress responses larger than 1KB
    filter: (req, res) => {
        if (req.headers['x-no-compression']) {
            return false;
        }
        return compression.filter(req, res);
    }
}));

// 2. Security headers
app.use(helmet({
    contentSecurityPolicy: false, // Disable CSP for demo
    crossOriginEmbedderPolicy: false
}));

// 3. Rate limiting to prevent abuse
const globalLimiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 1000, // Limit each IP to 1000 requests per windowMs
    message: {
        error: 'Too many requests from this IP',
        retryAfter: 15 * 60 // seconds
    },
    standardHeaders: true,
    legacyHeaders: false,
});

// 4. Progressive delays for excessive requests
const speedLimiter = slowDown({
    windowMs: 15 * 60 * 1000, // 15 minutes
    delayAfter: 100, // allow 100 requests per windowMs without delay
    delayMs: 500 // add 500ms delay per request after delayAfter
});

app.use(globalLimiter);
app.use(speedLimiter);

// 5. Session configuration with Redis
app.use(session({
    store: new RedisStore({ client: redisClient }),
    secret: process.env.SESSION_SECRET || 'your-secret',
    resave: false,
    saveUninitialized: false,
    cookie: {
        maxAge: 24 * 60 * 60 * 1000, // 24 hours
```

```
        secure: process.env.NODE_ENV === 'production'
    }
}));

// Body parsing
app.use(express.json({ limit: '10mb' }));
app.use(express.urlencoded({ extended: true, limit: '10mb' }));

// Caching middleware for expensive operations
const cache = (duration = 300) => { // Default 5 minutes
    return async (req, res, next) => {
        if (req.method !== 'GET') {
            return next();
        }

        const key = `cache:${req.originalUrl}`;

        try {
            const cachedResponse = await redisClient.get(key);

            if (cachedResponse) {
                console.log(`Cache hit for ${req.originalUrl}`);
                res.setHeader('X-Cache', 'HIT');
                return res.json(JSON.parse(cachedResponse));
            }

            // Store original res.json method
            const originalJson = res.json;

            // Override res.json to cache response
            res.json = function(body) {
                // Cache the response
                redisClient.setex(key, duration, JSON.stringify(body));
                res.setHeader('X-Cache', 'MISS');

                // Call original json method
                return originalJson.call(this, body);
            };

            next();
        } catch (error) {
            console.error('Cache error:', error);
            next(); // Continue without caching
        }
    };
};

// Health check endpoint (no caching)
app.get('/health', (req, res) => {
    res.json({
        status: 'healthy',
        worker: process.pid,
        uptime: process.uptime(),
        memory: process.memoryUsage(),
        timestamp: new Date().toISOString()
    });
```

```
});

// Cached expensive operation
app.get('/expensive-data', cache(600), async (req, res) => {
    console.log(`Processing expensive operation on worker ${process.pid}`);

    // Simulate expensive computation
    const start = Date.now();

    // Generate large dataset
    const data = Array.from({ length: 10000 }, (_, i) => ({
        id: i + 1,
        name: `Item ${i + 1}`,
        value: Math.random() * 1000,
        category: ['A', 'B', 'C'][Math.floor(Math.random() * 3)]
    }));

    // Simulate processing time
    await new Promise(resolve => setTimeout(resolve, 2000));

    const processingTime = Date.now() - start;

    res.json({
        message: 'Expensive data processed',
        worker: process.pid,
        processingTime: `${processingTime}ms`,
        dataCount: data.length,
        data: data.slice(0, 10), // Return only first 10 items
        timestamp: new Date().toISOString()
    });
});

// CPU intensive route (demonstrates load distribution)
app.get('/cpu-intensive', (req, res) => {
    const start = Date.now();

    // Simulate CPU-intensive task
    let result = 0;
    for (let i = 0; i < 10000000; i++) {
        result += Math.sqrt(i);
    }

    const processingTime = Date.now() - start;

    res.json({
        message: 'CPU intensive task completed',
        worker: process.pid,
        result: Math.floor(result),
        processingTime: `${processingTime}ms`,
        timestamp: new Date().toISOString()
    });
});

// Database simulation with connection pooling concept
const simulateDatabase = async (query, params = {}) => {
    // Simulate database response time
```

```
    const delay = Math.random() * 100 + 50; // 50-150ms
    await new Promise(resolve => setTimeout(resolve, delay));

    return {
        query,
        params,
        results: Array.from({ length: 5 }, (_, i) => ({
            id: i + 1,
            data: `Result ${i + 1} for ${query}`
        })),
        executionTime: delay,
        worker: process.pid
    };
};

// Paginated API endpoint with caching
app.get('/users', cache(300), async (req, res) => {
    const { page = 1, limit = 10, search = '' } = req.query;
    const offset = (page - 1) * limit;

    try {
        // Simulate database query
        const dbResult = await simulateDatabase('SELECT * FROM users', {
            offset,
            limit,
            search
        });

        res.json({
            success: true,
            data: dbResult.results,
            pagination: {
                currentPage: parseInt(page),
                itemsPerPage: parseInt(limit),
                totalItems: 1000, // Simulated total
                totalPages: Math.ceil(1000 / limit)
            },
            meta: {
                worker: process.pid,
                executionTime: `${dbResult.executionTime}ms`,
                timestamp: new Date().toISOString()
            }
        });
    } catch (error) {
        res.status(500).json({
            success: false,
            error: 'Database error',
            worker: process.pid
        });
    }
});

// Error handling
app.use((err, req, res, next) => {
    console.error(`Error on worker ${process.pid}:`, err);
    res.status(500).json({
```

```
        error: 'Internal server error',
        worker: process.pid,
        timestamp: new Date().toISOString()
    });
});

// 404 handler
app.use((req, res) => {
    res.status(404).json({
        error: 'Route not found',
        worker: process.pid,
        path: req.originalUrl
    });
});

// Start server
app.listen(PORT, () => {
    console.log(`Worker ${process.pid} listening on port ${PORT}`);
});

// Graceful shutdown
process.on('SIGTERM', () => {
    console.log(`Worker ${process.pid} received SIGTERM, shutting down gracefully...`);
    process.exit(0);
});
```

PM2 Configuration for Production:

```
// ecosystem.config.js - PM2 configuration file
module.exports = {
    apps: [{
        name: 'express-app',
        script: './app.js',
        instances: 'max', // Use all available CPU cores
        exec_mode: 'cluster',
        env: {
            NODE_ENV: 'development',
            PORT: 3000
        },
        env_production: {
            NODE_ENV: 'production',
            PORT: 3000
        },
        // Performance monitoring
        max_memory_restart: '1G',
        max_restarts: 10,
        min_uptime: '10s',

        // Logging
        log_file: './logs/combined.log',
        out_file: './logs/out.log',
        error_file: './logs/error.log',
        log_date_format: 'YYYY-MM-DD HH:mm Z',

        // Advanced settings
```

```
        watch: false, // Don't watch files in production
        ignore_watch: ['node_modules', 'logs'],
        merge_logs: true,

        // Load balancing
        instance_var: 'INSTANCE_ID'
    }]
};

// Start with: pm2 start ecosystem.config.js --env production
// Monitor with: pm2 monit
// View logs: pm2 logs
// Reload: pm2 reload express-app
```

11. Testing with Mocha, Chai, and Supertest

What it does: Testing ensures your Express application works correctly by automatically verifying functionality, catching bugs early, and providing confidence when making changes. Mocha provides the testing framework, Chai offers assertion libraries, and Supertest enables HTTP endpoint testing.[14]

Test Setup and Configuration:

```
// test/setup.js - Test configuration and setup
const { expect } = require('chai');
const request = require('supertest');

// Global test configuration
process.env.NODE_ENV = 'test';

// Make expect and request available globally in tests
global.expect = expect;
global.request = request;

// Test database configuration
const mongoose = require('mongoose');
const { MongoMemoryServer } = require('mongodb-memory-server');

let mongoServer;

// Setup test database before all tests
before(async function() {
    this.timeout(60000); // Increase timeout for database setup

    mongoServer = await MongoMemoryServer.create();
    const mongoUri = mongoServer.getUri();

    await mongoose.connect(mongoUri, {
        useNewUrlParser: true,
        useUnifiedTopology: true
    });

    console.log('Test database connected');
});
```

```
// Clean up after all tests
after(async function() {
    await mongoose.disconnect();
    await mongoServer.stop();
    console.log('Test database disconnected');
});

// Clear database between tests
afterEach(async function() {
    const collections = mongoose.connection.collections;
    for (const key in collections) {
        await collections[key].deleteMany({});
    }
});
```

Unit Tests for Middleware and Utilities:

```
// test/unit/middleware.test.js - Testing custom middleware
const express = require('express');
const sinon = require('sinon');

// Import middleware to test
const { authenticate, authorize, validateInput } = require('../../middleware/auth');

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

    // Setup mock objects before each test
    beforeEach(() => {
        req = {
            headers: {},
            user: null
        };
        res = {
            status: sinon.stub().returns({ json: sinon.stub() }),
            json: sinon.stub()
        };
        next = sinon.stub();
    });

    describe('authenticate middleware', () => {
        it('should call next() when valid token is provided', () => {
            // Arrange
            req.headers.authorization = 'Bearer valid-token';

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

            // Assert
            expect(next.calledOnce).to.be.true;
            expect(req.user).to.exist;
            expect(req.user.id).to.equal(1);
        });
```

```
        it('should return 401 when no token is provided', () => {
            // Act
            authenticate(req, res, next);

            // Assert
            expect(res.status.calledWith(401)).to.be.true;
            expect(res.status().json.calledWith({ error: 'No token provided' })).to.be.tr
            expect(next.called).to.be.false;
        });

        it('should return 401 when invalid token is provided', () => {
            // Arrange
            req.headers.authorization = 'Bearer invalid-token';

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

            // Assert
            expect(res.status.calledWith(401)).to.be.true;
            expect(next.called).to.be.false;
        });

        it('should handle malformed authorization header', () => {
            // Arrange
            req.headers.authorization = 'InvalidFormat';

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

            // Assert
            expect(res.status.calledWith(401)).to.be.true;
        });
    });

    describe('authorize middleware', () => {
        beforeEach(() => {
            req.user = { id: 1, role: 'user' };
        });

        it('should allow access for authorized role', () => {
            // Arrange
            const authorizeUser = authorize(['user', 'admin']);

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

            // Assert
            expect(next.calledOnce).to.be.true;
        });

        it('should deny access for unauthorized role', () => {
            // Arrange
            const authorizeAdmin = authorize(['admin']);

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

```
            // Assert
            expect(res.status.calledWith(403)).to.be.true;
            expect(next.called).to.be.false;
        });

        it('should deny access when user is not authenticated', () => {
            // Arrange
            req.user = null;
            const authorizeUser = authorize(['user']);

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

            // Assert
            expect(res.status.calledWith(401)).to.be.true;
        });
    });
});

// test/unit/utils.test.js - Testing utility functions
const {
    validateEmail,
    hashPassword,
    comparePassword,
    generateToken,
    formatDate
} = require('../../utils/helpers');

describe('Utility Functions', () => {
    describe('validateEmail', () => {
        it('should return true for valid email addresses', () => {
            const validEmails = [
                'test@example.com',
                'user.name@domain.co.uk',
                'user+tag@example.org'
            ];

            validEmails.forEach(email => {
                expect(validateEmail(email)).to.be.true;
            });
        });

        it('should return false for invalid email addresses', () => {
            const invalidEmails = [
                'invalid-email',
                '@example.com',
                'test@',
                'test..test@example.com'
            ];

            invalidEmails.forEach(email => {
                expect(validateEmail(email)).to.be.false;
            });
        });
    });
```

```
    describe('Password utilities', () => {
        const testPassword = 'testPassword123!';
        let hashedPassword;

        it('should hash password correctly', async () => {
            hashedPassword = await hashPassword(testPassword);

            expect(hashedPassword).to.be.a('string');
            expect(hashedPassword).to.not.equal(testPassword);
            expect(hashedPassword.length).to.be.greaterThan(20);
        });

        it('should compare password correctly', async () => {
            const isMatch = await comparePassword(testPassword, hashedPassword);
            expect(isMatch).to.be.true;
        });

        it('should fail comparison with wrong password', async () => {
            const isMatch = await comparePassword('wrongPassword', hashedPassword);
            expect(isMatch).to.be.false;
        });
    });

    describe('generateToken', () => {
        it('should generate JWT token', () => {
            const payload = { id: 1, email: 'test@example.com' };
            const token = generateToken(payload);

            expect(token).to.be.a('string');
            expect(token.split('.')).to.have.lengthOf(3); // JWT has 3 parts
        });
    });

    describe('formatDate', () => {
        it('should format date correctly', () => {
            const date = new Date('2024-01-15T10:30:00Z');
            const formatted = formatDate(date, 'YYYY-MM-DD');

            expect(formatted).to.equal('2024-01-15');
        });

        it('should handle invalid date', () => {
            const formatted = formatDate('invalid-date');
            expect(formatted).to.equal('Invalid Date');
        });
    });
});
```

Integration Tests for API Endpoints:

```
// test/integration/auth.test.js - Testing authentication endpoints
const app = require('../../app');

describe('Authentication Endpoints', () => {
```

```
    describe('POST /auth/register', () => {
        const validUserData = {
            name: 'John Doe',
            email: 'john@example.com',
            password: 'StrongPass123!',
            confirmPassword: 'StrongPass123!'
        };

        it('should register a new user successfully', async () => {
            const res = await request(app)
                .post('/auth/register')
                .send(validUserData)
                .expect(201);

            expect(res.body).to.have.property('success', true);
            expect(res.body).to.have.property('user');
            expect(res.body.user).to.have.property('email', validUserData.email);
            expect(res.body.user).to.not.have.property('password');
            expect(res.body).to.have.property('token');
        });

        it('should not register user with invalid email', async () => {
            const invalidData = { ...validUserData, email: 'invalid-email' };

            const res = await request(app)
                .post('/auth/register')
                .send(invalidData)
                .expect(400);

            expect(res.body).to.have.property('success', false);
            expect(res.body).to.have.property('errors');
            expect(res.body.errors).to.be.an('array');
            expect(res.body.errors[^0]).to.have.property('field', 'email');
        });

        it('should not register user with weak password', async () => {
            const weakPasswordData = { ...validUserData, password: '123', confirmPasswor

            const res = await request(app)
                .post('/auth/register')
                .send(weakPasswordData)
                .expect(400);

            expect(res.body.errors.some(err => err.field === 'password')).to.be.true;
        });

        it('should not register user with mismatched passwords', async () => {
            const mismatchedData = { ...validUserData, confirmPassword: 'DifferentPass123

            const res = await request(app)
                .post('/auth/register')
                .send(mismatchedData)
                .expect(400);
        });

        it('should not register duplicate email', async () => {
```

```
            // First registration
            await request(app)
                .post('/auth/register')
                .send(validUserData)
                .expect(201);

            // Second registration with same email
            const res = await request(app)
                .post('/auth/register')
                .send(validUserData)
                .expect(400);

            expect(res.body.message).to.include('already registered');
        });
    });

    describe('POST /auth/login', () => {
        let testUser;

        beforeEach(async () => {
            // Create test user before each login test
            const userData = {
                name: 'Test User',
                email: 'test@login.com',
                password: 'TestPass123!',
                confirmPassword: 'TestPass123!'
            };

            const registerRes = await request(app)
                .post('/auth/register')
                .send(userData);

            testUser = registerRes.body.user;
        });

        it('should login with valid credentials', async () => {
            const loginData = {
                email: 'test@login.com',
                password: 'TestPass123!'
            };

            const res = await request(app)
                .post('/auth/login')
                .send(loginData)
                .expect(200);

            expect(res.body).to.have.property('success', true);
            expect(res.body).to.have.property('token');
            expect(res.body).to.have.property('user');
            expect(res.body.user.email).to.equal(loginData.email);
        });

        it('should not login with invalid email', async () => {
            const loginData = {
                email: 'nonexistent@example.com',
                password: 'TestPass123!'
```

```
            };

            const res = await request(app)
                .post('/auth/login')
                .send(loginData)
                .expect(401);

            expect(res.body).to.have.property('success', false);
            expect(res.body.message).to.include('Invalid credentials');
        });

        it('should not login with invalid password', async () => {
            const loginData = {
                email: 'test@login.com',
                password: 'WrongPassword123!'
            };

            const res = await request(app)
                .post('/auth/login')
                .send(loginData)
                .expect(401);

            expect(res.body).to.have.property('success', false);
        });

        it('should validate required fields', async () => {
            const res = await request(app)
                .post('/auth/login')
                .send({})
                .expect(400);

            expect(res.body.errors).to.be.an('array');
            expect(res.body.errors.length).to.be.greaterThan(0);
        });
    });
});

// test/integration/users.test.js - Testing user management endpoints
describe('User Management Endpoints', () => {
    let authToken;
    let testUserId;

    beforeEach(async () => {
        // Create and login test user to get auth token
        const userData = {
            name: 'Auth Test User',
            email: 'authtest@example.com',
            password: 'AuthTest123!',
            confirmPassword: 'AuthTest123!'
        };

        const registerRes = await request(app)
            .post('/auth/register')
            .send(userData);

        authToken = registerRes.body.token;
```

```
        testUserId = registerRes.body.user.id;
    });

    describe('GET /api/users', () => {
        it('should get users list with authentication', async () => {
            const res = await request(app)
                .get('/api/users')
                .set('Authorization', `Bearer ${authToken}`)
                .expect(200);

            expect(res.body).to.have.property('success', true);
            expect(res.body).to.have.property('users');
            expect(res.body.users).to.be.an('array');
        });

        it('should not get users without authentication', async () => {
            const res = await request(app)
                .get('/api/users')
                .expect(401);

            expect(res.body).to.have.property('error', 'No token provided');
        });

        it('should support pagination', async () => {
            const res = await request(app)
                .get('/api/users?page=1&limit=5')
                .set('Authorization', `Bearer ${authToken}`)
                .expect(200);

            expect(res.body).to.have.property('pagination');
            expect(res.body.pagination.currentPage).to.equal(1);
            expect(res.body.pagination.itemsPerPage).to.equal(5);
        });
    });

    describe('GET /api/users/:id', () => {
        it('should get specific user by ID', async () => {
            const res = await request(app)
                .get(`/api/users/${testUserId}`)
                .set('Authorization', `Bearer ${authToken}`)
                .expect(200);

            expect(res.body).to.have.property('user');
            expect(res.body.user.id).to.equal(testUserId);
        });

        it('should return 404 for non-existent user', async () => {
            const nonExistentId = '507f1f77bcf86cd799439011';

            const res = await request(app)
                .get(`/api/users/${nonExistentId}`)
                .set('Authorization', `Bearer ${authToken}`)
                .expect(404);

            expect(res.body.message).to.include('not found');
        });
```

```
        it('should return 400 for invalid user ID format', async () => {
            const res = await request(app)
                .get('/api/users/invalid-id')
                .set('Authorization', `Bearer ${authToken}`)
                .expect(400);
        });
    });

    describe('PUT /api/users/:id', () => {
        it('should update user profile', async () => {
            const updateData = {
                name: 'Updated Name',
                email: 'updated@example.com'
            };

            const res = await request(app)
                .put(`/api/users/${testUserId}`)
                .set('Authorization', `Bearer ${authToken}`)
                .send(updateData)
                .expect(200);

            expect(res.body).to.have.property('success', true);
            expect(res.body.user.name).to.equal(updateData.name);
            expect(res.body.user.email).to.equal(updateData.email);
        });

        it('should not update with invalid email format', async () => {
            const updateData = { email: 'invalid-email' };

            const res = await request(app)
                .put(`/api/users/${testUserId}`)
                .set('Authorization', `Bearer ${authToken}`)
                .send(updateData)
                .expect(400);
        });
    });
});
```

Test Coverage and Performance Tests:

```
// test/performance/load.test.js - Performance testing
describe('Performance Tests', () => {
    describe('API Response Times', () => {
        it('should respond to health check within 100ms', async () => {
            const start = Date.now();

            await request(app)
                .get('/health')
                .expect(200);

            const responseTime = Date.now() - start;
            expect(responseTime).to.be.below(100);
        });
```

```
        it('should handle concurrent requests efficiently', async () => {
            const concurrentRequests = 50;
            const requests = Array(concurrentRequests).fill().map(() =>
                request(app)
                    .get('/health')
                    .expect(200)
            );

            const start = Date.now();
            await Promise.all(requests);
            const totalTime = Date.now() - start;

            // All requests should complete within reasonable time
            expect(totalTime).to.be.below(5000); // 5 seconds
        });
    });

    describe('Memory Usage', () => {
        it('should not have memory leaks in middleware', async () => {
            const initialMemory = process.memoryUsage().heapUsed;

            // Make many requests to test for memory leaks
            for (let i = 0; i < 1000; i++) {
                await request(app)
                    .get('/health')
                    .expect(200);
            }

            // Force garbage collection if available
            if (global.gc) global.gc();

            const finalMemory = process.memoryUsage().heapUsed;
            const memoryIncrease = finalMemory - initialMemory;

            // Memory increase should be reasonable (less than 50MB)
            expect(memoryIncrease).to.be.below(50 * 1024 * 1024);
        });
    });
});

// package.json test scripts
/*
{
  "scripts": {
    "test": "mocha test/setup.js test/**/*.test.js --timeout 10000",
    "test:watch": "mocha test/setup.js test/**/*.test.js --watch --timeout 10000",
    "test:coverage": "nyc --reporter=text --reporter=html npm test",
    "test:unit": "mocha test/setup.js test/unit/*.test.js",
    "test:integration": "mocha test/setup.js test/integration/*.test.js",
    "test:performance": "mocha test/setup.js test/performance/*.test.js"
  }
}
*/
```

Complete Express Application Example

What it demonstrates: A production-ready Express application incorporating all the concepts covered above, with proper project structure, error handling, security, and performance optimizations.

```
// server.js - Main server file with all optimizations
const express = require('express');
const cors = require('cors');
const helmet = require('helmet');
const compression = require('compression');
const rateLimit = require('express-rate-limit');
const mongoSanitize = require('express-mongo-sanitize');
const xss = require('xss-clean');
const hpp = require('hpp');
const cookieParser = require('cookie-parser');
const session = require('express-session');
const MongoStore = require('connect-mongo');
const morgan = require('morgan');
const winston = require('winston');

// Import custom modules
const { connectDatabase } = require('./config/database');
const logger = require('./config/logger');
const AppError = require('./utils/AppError');
const globalErrorHandler = require('./middleware/errorHandler');

// Import routes
const authRoutes = require('./routes/auth');
const userRoutes = require('./routes/users');
const productRoutes = require('./routes/products');

const app = express();

// Trust proxy for deployment behind reverse proxy
app.set('trust proxy', 1);

// Template engine setup
app.set('view engine', 'ejs');
app.set('views', './views');

// Security Middleware
app.use(helmet({
    contentSecurityPolicy: {
        directives: {
            defaultSrc: ["'self'"],
            styleSrc: ["'self'", "'unsafe-inline'", "https://fonts.googleapis.com"],
            fontSrc: ["'self'", "https://fonts.gstatic.com"],
            imgSrc: ["'self'", "data:", "https:"],
            scriptSrc: ["'self'"]
        }
    }
}));

// CORS configuration
```

```
app.use(cors({
    origin: process.env.ALLOWED_ORIGINS?.split(',') || ['http://localhost:3000'],
    credentials: true,
    methods: ['GET', 'POST', 'PUT', 'DELETE', 'PATCH'],
    allowedHeaders: ['Content-Type', 'Authorization']
}));

// Rate limiting
const globalLimiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 1000, // requests per window
    message: {
        error: 'Too many requests from this IP',
        retryAfter: '15 minutes'
    }
});

const authLimiter = rateLimit({
    windowMs: 15 * 60 * 1000,
    max: 5, // Stricter limit for auth endpoints
    skipSuccessfulRequests: true,
    message: {
        error: 'Too many authentication attempts',
        retryAfter: '15 minutes'
    }
});

app.use(globalLimiter);

// Compression
app.use(compression());

// Body parsing
app.use(express.json({ limit: '10mb' }));
app.use(express.urlencoded({ extended: true, limit: '10mb' }));

// Cookie parsing
app.use(cookieParser(process.env.COOKIE_SECRET));

// Data sanitization
app.use(mongoSanitize()); // Against NoSQL injection
app.use(xss()); // Against XSS attacks
app.use(hpp()); // Against HTTP Parameter Pollution

// Session configuration
app.use(session({
    secret: process.env.SESSION_SECRET,
    name: 'sessionId',
    resave: false,
    saveUninitialized: false,
    store: MongoStore.create({
        mongoUrl: process.env.DATABASE_URI,
        collectionName: 'sessions'
    }),
    cookie: {
        secure: process.env.NODE_ENV === 'production',
```

```
        httpOnly: true,
        maxAge: 24 * 60 * 60 * 1000, // 24 hours
        sameSite: 'strict'
    }
}));

// HTTP request logging
if (process.env.NODE_ENV !== 'test') {
    const morganStream = {
        write: (message) => logger.http(message.trim())
    };

    app.use(morgan('combined', { stream: morganStream }));
}

// Static files
app.use(express.static('public', {
    maxAge: process.env.NODE_ENV === 'production' ? '1d' : 0,
    etag: false
}));

// Request ID middleware for tracking
app.use((req, res, next) => {
    req.id = Date.now().toString(36) + Math.random().toString(36).substr(2);
    res.setHeader('X-Request-ID', req.id);
    next();
});

// API Routes
app.use('/auth', authLimiter, authRoutes);
app.use('/api/users', userRoutes);
app.use('/api/products', productRoutes);

// Health check endpoint
app.get('/health', (req, res) => {
    res.json({
        status: 'healthy',
        timestamp: new Date().toISOString(),
        uptime: process.uptime(),
        environment: process.env.NODE_ENV,
        version: process.env.npm_package_version || '1.0.0'
    });
});

// API documentation redirect
app.get('/docs', (req, res) => {
    res.redirect('https://api-docs.example.com');
});

// Handle undefined routes (404)
app.all('*', (req, res, next) => {
    const error = new AppError(404, `Route ${req.originalUrl} not found on this server`);
    next(error);
});

// Global error handling middleware
```

```
app.use(globalErrorHandler);

// Database connection and server startup
const startServer = async () => {
    try {
        // Connect to database
        await connectDatabase();

        // Start server
        const PORT = process.env.PORT || 3000;
        const server = app.listen(PORT, () => {
            logger.info('Server started successfully', {
                port: PORT,
                environment: process.env.NODE_ENV,
                nodeVersion: process.version,
                timestamp: new Date().toISOString()
            });
        });

        // Graceful shutdown handling
        process.on('SIGTERM', () => {
            logger.info('SIGTERM received, shutting down gracefully');
            server.close(() => {
                logger.info('Process terminated');
                process.exit(0);
            });
        });

        process.on('SIGINT', () => {
            logger.info('SIGINT received, shutting down gracefully');
            server.close(() => {
                logger.info('Process terminated');
                process.exit(0);
            });
        });

    } catch (error) {
        logger.error('Failed to start server', {
            error: error.message,
            stack: error.stack
        });
        process.exit(1);
    }
};

// Handle uncaught exceptions
process.on('uncaughtException', (error) => {
    logger.error('Uncaught Exception:', {
        error: error.message,
        stack: error.stack
    });
    process.exit(1);
});

// Handle unhandled promise rejections
process.on('unhandledRejection', (reason, promise) => {
```

```
    logger.error('Unhandled Rejection:', {
        reason: reason,
        promise: promise
    });
    process.exit(1);
});

// Start the server
if (require.main === module) {
    startServer();
}

module.exports = app;
```

Interview Questions & Answers

Basic Questions (1-10)

1. What is Express.js and why is it used?

Answer: Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It simplifies the process of building web applications by providing a set of powerful tools and functionalities, such as routing, middleware, and template engines. Express.js is often used because it allows for rapid development of web applications and APIs.[2] [1]

Key benefits include:

- **Simplicity**: Minimal setup required to get started
- **Flexibility**: Can be used for APIs, web apps, or hybrid applications
- **Middleware ecosystem**: Extensive third-party middleware available
- **Performance**: Fast and lightweight framework
- **Community support**: Large community and extensive documentation

2. Explain the concept of middleware in Express.js.

Answer: 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. They can execute any code, make changes to the request and response objects, end the request-response cycle, or call the next middleware function in the stack. Middleware functions are used for tasks like logging, authentication, error handling, and more. They are invoked in the order they are added (using app.use()).[7] [2]

Middleware characteristics:

- **Sequential execution**: Middleware runs in the order it's defined
- **Request modification**: Can modify req and res objects

- **Flow control**: Must call next() to continue or end the cycle
- **Error handling**: Can catch and handle errors in the pipeline

3. How do you handle errors in Express.js?

Answer: In Express.js, errors can be handled using middleware functions. An error-handling middleware function has four arguments instead of three: (err, req, res, next). When an error is passed to next(), Express will skip all remaining middleware and route handlers and go directly to the error-handling middleware. For example:[8]

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

Error handling best practices:

- **Centralized error handling**: Use global error handler middleware
- **Custom error classes**: Create structured error objects
- **Async error handling**: Use try-catch blocks or async wrappers
- **Different error types**: Handle operational vs programming errors differently

4. What is the difference between app.get() and app.use() in Express.js?

Answer: app.get() is a route definition for the HTTP GET method. It takes a path and a callback function that will be executed when a GET request is made to that path. On the other hand, app.use() is used to mount middleware functions or routers at a specific path. The function(s) will be executed whenever a request is made to the path, regardless of the HTTP method.[2]

Key differences:

- **HTTP method specificity**: app.get() only handles GET requests, app.use() handles all methods
- **Path matching**: app.get() requires exact path match, app.use() matches path prefixes
- **Purpose**: app.get() for route handlers, app.use() for middleware
- **Execution timing**: app.use() runs before route handlers

5. How can you serve static files in Express.js?

Answer: Express.js provides a built-in middleware function called express.static to serve static files. For example, to serve images, CSS files, and JavaScript files from a directory named 'public', you would use:[2]

```
app.use(express.static('public'));
```

Advanced static file serving:

```
// With options
app.use(express.static('public', {
    maxAge: '1d',        // Cache for 1 day
    etag: false,         // Disable ETag
    dotfiles: 'ignore'   // Ignore hidden files
}));

// Multiple static directories
app.use('/assets', express.static('public'));
app.use('/files', express.static('uploads'));
```

6. How do you retrieve parameters from a URL in Express.js?

Answer: In Express.js, you can retrieve parameters from a URL using the req.params object. For example, in the route definition app.get('/users/:userId', callback), you can access the userId parameter in the callback function using req.params.userId.[2]

Different parameter types:

- **Route parameters**: req.params for dynamic URL segments
- **Query parameters**: req.query for ?key=value pairs
- **Body parameters**: req.body for POST/PUT data
- **Header parameters**: req.headers for HTTP headers

7. What is the purpose of next() in Express.js?

Answer: The next() function is used to pass control to the next middleware function in the stack. If not called, the request will be left hanging. It can also be used to pass errors to the next error-handling middleware function by providing an argument to next(), e.g., next(err).[2]

next() usage patterns:

- **next()**: Continue to next middleware
- **next(error)**: Skip to error handler
- **next('route')**: Skip remaining route callbacks
- **No next() call**: End the request-response cycle

8. How do you enable CORS in Express.js?

Answer: To enable CORS (Cross-Origin Resource Sharing) in Express.js, you can use the cors middleware. First, you need to install it using npm, and then you can include and use it in your application:[2]

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

Advanced CORS configuration:

```
app.use(cors({
    origin: ['http://localhost:3000', 'https://myapp.com'],
    credentials: true,
    methods: ['GET', 'POST', 'PUT', 'DELETE'],
    allowedHeaders: ['Content-Type', 'Authorization']
}));
```

9. How can you set up a route that will match multiple HTTP verbs in Express.js?

Answer: You can use the app.route() method to create a route path and then chain HTTP verb methods to it. For example:[2]

```
app.route('/book')
  .get((req, res) => {
    res.send('Get a book');
  })
  .post((req, res) => {
    res.send('Add a book');
  })
  .put((req, res) => {
    res.send('Update a book');
  });
```

Alternative approaches:

- **app.all()**: Matches all HTTP methods
- **Multiple method calls**: Chain multiple app.get(), app.post(), etc.
- **Router-level chaining**: Use express.Router() with route chaining

10. How do you use middleware for specific routes?

Answer: Middleware can be used for specific routes by providing the middleware function as an argument before the route's callback function. For example:[2]

```
const loggingMiddleware = (req, res, next) => {
    console.log('Logged:', req.url);
    next();
};

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

In the above example, the loggingMiddleware will only be executed for the /users route.

Advanced Questions (11-25)

11. Explain the difference between synchronous and asynchronous error handling in Express.

Answer: Synchronous errors in Express are automatically caught and passed to error handlers. However, asynchronous errors must be explicitly passed to next(). For async/await functions, you need to wrap them in try-catch blocks or use an async error handler wrapper:[15]

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

app.get('/async-route', asyncHandler(async (req, res) => {
    const data = await database.fetch(); // If this throws, it's caught
    res.json(data);
}));
```

Key differences:

- **Sync errors**: Automatically handled by Express
- **Async errors**: Must manually call next(error)
- **Promise rejections**: Need explicit handling or wrapper functions
- **Try-catch blocks**: Required for async/await error handling

12. How would you implement authentication using JWT in Express?

Answer: JWT authentication involves creating tokens on login and verifying them on protected routes:[16]

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

// Login route - generate token
app.post('/login', async (req, res) => {
    const user = await authenticateUser(req.body);
    const token = jwt.sign({ id: user.id }, 'secret', { expiresIn: '24h' });
    res.json({ token });
});

// Middleware to verify token
const verifyToken = (req, res, next) => {
    const token = req.headers.authorization?.split(' ')[^1];
    try {
        const decoded = jwt.verify(token, 'secret');
        req.user = decoded;
        next();
    } catch (error) {
        res.status(401).json({ message: 'Invalid token' });
```

```
    }
};
```

JWT implementation considerations:

- **Token storage**: Store securely on client (httpOnly cookies recommended)
- **Token expiration**: Set reasonable expiration times
- **Secret management**: Use environment variables for secrets
- **Refresh tokens**: Implement token refresh mechanism for better UX

13. What are the differences between sessions and cookies in Express?

Answer: Sessions store data on the server with only a session ID sent to the client, while cookies store data directly on the client:[11]

Sessions:

- **Server-side storage**: Data stored on server, more secure
- **Larger capacity**: No size limitations (depends on server memory/database)
- **Automatic expiration**: Can be configured with TTL
- **Requires middleware**: express-session package needed

Cookies:

- **Client-side storage**: Data stored in browser, less secure
- **Size limitation**: 4KB maximum per cookie
- **Manual management**: Must explicitly set and clear
- **Browser dependent**: Subject to browser cookie policies

14. How do you optimize Express.js performance in production?

Answer: Key performance optimizations include:[15]

- **Set NODE_ENV to 'production'**: Enables production optimizations
- **Use compression middleware**: Reduces response sizes
- **Enable caching**: Implement Redis/Memcached for frequently accessed data
- **Use clustering**: Utilize multiple CPU cores with cluster module or PM2
- **Implement rate limiting**: Prevent abuse and ensure fair usage
- **Optimize database queries**: Use indexing, pagination, and connection pooling
- **Use a reverse proxy**: Nginx for load balancing and static file serving

```
// Production optimizations
if (process.env.NODE_ENV === 'production') {
    app.use(compression());
    app.use(helmet());
```

```
    app.use(rateLimit({ windowMs: 15 * 60 * 1000, max: 100 }));
}
```

15. Explain the Express.js request-response lifecycle.

Answer: The Express lifecycle follows this pattern:

1. **Request received** - Express receives HTTP request
2. **Middleware execution** - Runs in order of definition
3. **Route matching** - Finds matching route handler
4. **Route handler execution** - Executes callback function
5. **Response sent** - Sends response back to client
6. **Cleanup** - Express cleans up resources

The key is that middleware runs sequentially using next(), and the first matching route handler is executed unless next() is called to continue to the next handler.

16. How do you handle file uploads in Express.js?

Answer: File uploads are handled using Multer middleware, which processes multipart/form-data:[9]

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

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

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

// Single file upload
app.post('/upload', upload.single('avatar'), (req, res) => {
    res.json({ file: req.file });
});
```

17. What is input validation and why is it important in Express?

Answer: Input validation ensures data meets specific criteria while sanitization cleans data to prevent security vulnerabilities. It's crucial for preventing XSS, SQL injection, and ensuring data integrity:[10]

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

app.post('/user', [
    body('email').isEmail().normalizeEmail(),
    body('password').isLength({ min: 8 }).trim(),
    body('name').trim().escape()
], (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
    // Process validated data
});
```

18. How do you implement logging in Express applications?

Answer: Logging is implemented using Winston for application logs and Morgan for HTTP request logs:[12]

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

// Winston logger
const logger = winston.createLogger({
    level: 'info',
    format: winston.format.json(),
    transports: [
        new winston.transports.File({ filename: 'error.log', level: 'error' }),
        new winston.transports.File({ filename: 'combined.log' })
    ]
});

// Morgan for HTTP logging
app.use(morgan('combined', {
    stream: { write: (message) => logger.http(message.trim()) }
}));
```

19. What is clustering and how does it improve Express app performance?

Answer: Clustering creates multiple worker processes to utilize all CPU cores, dramatically improving performance:[13]

```
const cluster = require('cluster');
const numCPUs = require('os').cpus().length;
```

```
if (cluster.isMaster) {
    // Fork workers
    for (let i = 0; i < numCPUs; i++) {
        cluster.fork();
    }

    cluster.on('exit', (worker) => {
        console.log(`Worker ${worker.process.pid} died`);
        cluster.fork();
    });
} else {
    // Worker process
    require('./app.js');
}
```

20. How do you test Express.js applications?

Answer: Testing involves unit tests for individual functions and integration tests for endpoints using Mocha, Chai, and Supertest:[14]

```
const request = require('supertest');
const app = require('../app');

describe('GET /users', () => {
    it('should return users list', async () => {
        const res = await request(app)
            .get('/users')
            .expect(200);

        expect(res.body).to.have.property('users');
        expect(res.body.users).to.be.an('array');
    });
});
```

21. Explain template engines in Express and when to use them.

Answer: Template engines enable server-side rendering by combining HTML templates with dynamic data. Use them when you need SEO-friendly pages or server-rendered content:[4]

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

app.get('/', (req, res) => {
    res.render('index', {
        title: 'My App',
        users: ['John', 'Jane']
    });
});
```

When to use template engines:

- **SEO requirements**: Server-rendered content is better for SEO
- **Progressive enhancement**: Start with server-rendered, add client-side features
- **Simple applications**: Less complexity than SPA frameworks
- **Multi-page applications**: Traditional web applications with multiple pages

22. How do you handle database connections in Express?

Answer: Database connections should be established once and reused throughout the application:

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

// Connection with error handling
const connectDB = async () => {
    try {
        await mongoose.connect(process.env.DATABASE_URI, {
            useNewUrlParser: true,
            useUnifiedTopology: true
        });
        console.log('Database connected');
    } catch (error) {
        console.error('Database connection failed:', error);
        process.exit(1);
    }
};

// Connection pooling for SQL databases
const pool = mysql.createPool({
    connectionLimit: 10,
    host: 'localhost',
    user: 'root',
    password: 'password',
    database: 'myapp'
});
```

23. What are Express.js best practices for security?

Answer: Security best practices include:[17] [18]

- **Use Helmet**: Adds security headers
- **Input validation**: Validate and sanitize all inputs
- **Rate limiting**: Prevent brute force attacks
- **HTTPS**: Use secure connections
- **Environment variables**: Store secrets securely
- **Keep dependencies updated**: Regular security updates

```
app.use(helmet());
app.use(rateLimit());
```

```
app.use(mongoSanitize());
app.use(xss());
```

24. How do you handle environment-specific configurations?

Answer: Use environment variables and configuration files for different environments:

```
// config/config.js
module.exports = {
    development: {
        port: 3000,
        database: 'mongodb://localhost/myapp-dev'
    },
    production: {
        port: process.env.PORT || 80,
        database: process.env.DATABASE_URI
    }
};

const config = require('./config')[process.env.NODE_ENV || 'development'];
```

25. What is the difference between res.send(), res.json(), and res.end()?

Answer: These methods end the request-response cycle with different behaviors:

- **res.send()**: Sends response, auto-detects content type
- **res.json()**: Sends JSON response, sets application/json header
- **res.end()**: Ends response without sending data

```
res.send('Hello'); // text/html
res.send({ message: 'Hello' }); // application/json
res.json({ message: 'Hello' }); // application/json (explicit)
res.end(); // No content, ends connection
```

Key differences:

- **Content-Type detection**: send() auto-detects, json() sets explicitly
- **Data formatting**: json() always stringifies objects
- **Performance**: end() is fastest for empty responses
- **Use cases**: send() for mixed content, json() for APIs, end() for completion signals

⁂

1. https://www.youtube.com/watch?v=fBzm9zja2Y8
2. https://www.geeksforgeeks.org/node-js/middleware-in-express-js/
3. https://www.geeksforgeeks.org/node-js/unique-features-of-express-js/
4. https://expressjs.com/en/guide/using-template-engines.html

5. https://www.ghazikhan.in/blog/expressjs-crash-course-build-restful-api-middleware
6. https://expressjs.com/en/guide/routing.html
7. https://expressjs.com/en/guide/using-middleware.html
8. https://expressjs.com/en/guide/error-handling.html
9. https://attacomsian.com/blog/express-file-upload-multer
10. https://www.digitalocean.com/community/tutorials/how-to-handle-form-inputs-efficiently-with-express-validator-in-express-js
11. https://www.geeksforgeeks.org/node-js/difference-between-sessions-and-cookies-in-express/
12. https://betterstack.com/community/guides/logging/how-to-install-setup-and-use-winston-and-morgan-to-log-node-js-applications/
13. https://kinsta.com/blog/node-js-clustering/
14. https://github.com/mfaisalkhatri/SuperTest_poc
15. https://expressjs.com/en/advanced/best-practice-performance.html
16. https://dev.to/michaelikoko/implementing-jwt-authentication-with-express-mongodb-and-passportjs-3fl7
17. https://escape.tech/blog/how-to-secure-express-js-api/
18. https://dev.to/tristankalos/expressjs-security-best-practices-1ja0
19. https://stackoverflow.com/questions/36202618/how-to-upload-file-using-multer-or-body-parser
20. https://www.linkedin.com/pulse/avoid-vulnerabilities-best-practices-input-sanitization-srikanth-r-x6cac
21. https://blog.appsignal.com/2021/02/03/improving-node-application-performance-with-clustering.html
22. https://blog.logrocket.com/multer-nodejs-express-upload-file/
23. https://www.educative.io/answers/how-to-handle-file-upload-in-expressjs
24. https://heynode.com/tutorial/how-validate-and-sanitize-expressjs-form/
25. https://nodejs.org/api/cluster.html
26. https://docs.nestjs.com/techniques/file-upload
27. https://www.nodejs-security.com/blog/input-validation-best-practices-for-nodejs
28. https://www.w3schools.com/nodejs/nodejs_cluster.asp
29. https://expressjs.com/en/resources/middleware/multer.html
30. https://expressjs.com/en/advanced/best-practice-security.html
31. https://blogs.halodoc.io/nodejs-clustering-using-pm2/
32. https://expressjs.com/en/resources/middleware/body-parser.html
33. https://express-validator.github.io/docs/
34. https://dev.to/romulogatto/expressjs-middleware-and-routing-advanced-concepts-56h9
35. https://moldstud.com/articles/p-in-depth-analysis-of-middleware-patterns-in-expressjs-best-practices-examples
36. https://documentation.solarwinds.com/en/success_center/loggly/content/admin/node-express-js-morgan-logging.htm
37. https://dev.to/mhmdlotfy96/testing-a-rest-api-in-node-js-with-express-using-mocha-and-chai-1258
38. https://www.w3schools.com/nodejs/nodejs_middleware.asp

39. https://www.youtube.com/watch?v=RxraE2rylMI
40. https://rrawat.com/blog/unit-test-express-api
41. https://signoz.io/blog/morgan-logger/
42. https://stackoverflow.com/questions/51137217/unit-testing-using-mocha-with-express-rest-api
43. https://expressjs.com/en/guide/writing-middleware.html
44. https://coralogix.com/blog/complete-winston-logger-guide-with-hands-on-examples/
45. https://www.reddit.com/r/node/comments/jajyap/mochachai_vs_jestsupertest_for_testing/
46. https://dev.to/a_shokn/step-up-your-expressjs-game-advanced-middleware-and-security-tips-for-beginners-3bk7
47. https://sematext.com/blog/node-js-logging/
48. https://keploy.io/blog/community/mastering-node-js-backend-testing-with-mocha-and-chai
49. https://lioncoding.com/logging-in-express-js-using-winston-and-morgan/