

# Complete Node.js Project Explanation

## Table of Contents

1. Project Overview
2. Execution Flow
3. File-by-File Explanation
4. Behind the Scenes Concepts
5. Important Terminologies

## Project Overview

This is a **REST API backend project** built with:

- **Express.js** (web framework)
- **MongoDB** (database via Mongoose)
- **JWT** (authentication)
- **Node.js** (runtime environment)

**Purpose:** User authentication system with registration, login, logout, and profile management.

## Execution Flow

### When You Start the Server (`npm start`)

1. server.js (Entry Point)  
↓
2. Connect to MongoDB Database  
↓
3. Start Express Server on PORT  
↓
4. app.js loads with all middlewares  
↓
5. Server is READY to accept requests

## File-by-File Explanation

### 1 server.js - The Entry Point

```
import app from './app.js';
```

**What:** Imports the Express application

**Why:** Separates server logic from app configuration (clean code practice)

**Behind the Scene:** Node's ES6 module system loads app.js

```
import { connectDB } from "./src/config/database.config.js";
```

**What:** Imports database connection function

**Why:** To establish MongoDB connection before starting server

**Behind the Scene:** Mongoose will create a connection pool

```
connectDB()
  .then(C) => {
```

**What:** Calls connectDB function which returns a Promise

**Why:** Database connection is async, so we wait for it

**Behind the Scene:**

- JavaScript Promise handling
- If connection succeeds → `.then()` executes
- If fails → `.catch()` executes

```
app.listen(process.env.PORT, (err) => {
```

**What:** Starts HTTP server

**Why:** To listen for incoming HTTP requests

**Behind the Scene:**

- Node's `http` module creates a server
- Binds to the PORT number
- `process.env.PORT` reads from .env file
- Callback function executes when server starts

**Why here?** Server should start ONLY after database is connected, otherwise API won't work.

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

**What:** Terminates Node.js process

**Why:** Gracefully shutdown if errors occur

**Behind the Scene:** Exit code 1 means error occurred

## 2 database.config.js - Database Connection

```
import mongoose from "mongoose";
```

**What:** MongoDB ODM (Object Data Modeling) library

**Why:** To interact with MongoDB in JavaScript way

**Behind the Scene:** Mongoose provides schema validation, middleware, query building

```
export const connectDB = async () => {
```

**What:** Async function declaration

**Why:** Database operations are asynchronous (take time)

**Behind the Scene:** Returns a Promise automatically

```
let client = await mongoose.connect(process.env.MONGODB_URI);
```

**What:** Connects to MongoDB

**Why:** Establish database connection

**Behind the Scene:**

- Creates connection pool (reusable connections)
- Opens TCP socket to MongoDB server
- Authenticates with credentials from URI
- `await` pauses execution until connected

```
console.log(`Database connected to ${client.connection.host}`);
```

**What:** Logs connection success

**Why:** Debugging & confirmation

**Behind the Scene:** Accesses connection metadata

## 3 app.js - Application Configuration

```
import dotenv from "dotenv";
dotenv.config({ quiet: true });
```

**What:** Loads environment variables from .env file

**Why:** Keep sensitive data (passwords, keys) secure

**Behind the Scene:**

- Reads .env file
- Adds variables to `process.env`
- `quiet: true` → suppress warnings

**Why at top?** Environment variables must load BEFORE other imports use them.

```
import cookieParser from "cookie-parser";
```

**What:** Middleware to parse cookies

**Why:** To read JWT tokens stored in cookies

**Behind the Scene:** Parses Cookie header into `req.cookies` object

```
import express from "express";
```

**What:** Web framework for Node.js

**Why:** To create API routes and handle HTTP requests

**Behind the Scene:** Express is a function that creates an application

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

**What:** Creates Express application instance

**Why:** This is your web server object

**Behind the Scene:** Returns object with methods like `.use()`, `.listen()`, `.get()`, etc.

```
app.use(cookieParser());
```

**What:** Registers middleware

**Why:** To parse cookies from ALL incoming requests

**Behind the Scene:**

- `app.use()` adds function to middleware chain
- Executes for EVERY request
- Must be before routes that need cookies

**Why here?** Order matters! Middleware executes top-to-bottom.

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

**What:** Built-in middleware

**Why:** Parses JSON request bodies

**Behind the Scene:**

- Reads `Content-Type: application/json` header
- Parses JSON string into JavaScript object
- Attaches to `req.body`

**Example:**

```
Request: { "username": "john" }
After middleware: req.body = { username: "john" }
```

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

**What:** Parses URL-encoded data (form submissions)**Why:** To handle HTML form data**Behind the Scene:**

- `extended: true` → uses `qs` library (supports nested objects)
- `extended: false` → uses `querystring` library (simple key-value)

**Example:**

```
Form: name=John&age=25
After middleware: req.body = { name: "John", age: "25" }
```

```
app.use("/api/user", userRoutes);
```

**What:** Mounts router on path**Why:** All user-related routes start with /api/user**Behind the Scene:**

- Prefixes all routes in `userRoutes`
- `/api/user/register` → calls `registerUser`
- Route modularization

**Why here?** After body parsers, because routes need parsed data.

```
app.use(errorMiddleware);
```

**What:** Global error handler**Why:** Catches all errors from routes/middlewares**Behind the Scene:**

- Must have 4 parameters: (err, req, res, next)

- Must be LAST middleware
- Express automatically calls it when `next(error)` is called

**Why last?** Catches errors from all previous middlewares/routes.

## 4 user.route.js - Route Definitions

```
import { Router } from "express";
```

**What:** Express Router class

**Why:** To create modular route handlers

**Behind the Scene:** Mini-application that can handle requests

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

**What:** Creates router instance

**Why:** Group related routes together

**Behind the Scene:** Similar to `app` but mountable

```
router.post("/register", validate(registerSchema), registerUser);
```

**What:** Defines POST route

**Why:** Handle user registration

**Behind the Scene:**

1. Request comes to `/api/user/register`
2. Goes through `validate(registerSchema)` middleware
3. If validation passes → `registerUser` controller executes
4. If fails → error middleware catches it

**Middleware Chain:**

```
Request → validate → registerUser → Response
      ↓ (if error)
      error middleware
```

```
router.post("/logout", authenticate, logoutUser);
```

**What:** Protected route

**Why:** Only logged-in users can logout

**Behind the Scene:**

1. `authenticate` checks JWT token
2. If valid → adds `req.myUser` and calls `next()`
3. Then `logoutUser` executes

```
router.patch("/update-profile", validate(updateProfileSchema), authenticate, updateProfile);
```

**What:** Multiple middlewares**Why:** Validate data AND authenticate user**Behind the Scene:** Executes left-to-right**Order matters!**

- Validate FIRST (reject bad data fast)
- Authenticate SECOND (check if user logged in)
- Controller LAST (business logic)

```
router.get("/current", authenticate, currentUser);
```

**What:** Check auth status**Why:** Frontend can verify if user is logged in**Behind the Scene:** Simply returns success if token is valid**5 auth.middleware.js - Authentication Logic**

```
export const authenticate = expressAsyncHandler(async (req, res, next) => {
```

**What:** Async middleware function**Why:** Database queries are async**Behind the Scene:** `expressAsyncHandler` catches async errors

```
const token = req?.cookies?.token;
```

**What:** Optional chaining to get token**Why:** Safely access nested properties**Behind the Scene:**

- `req.cookies` exists because of cookieParser middleware
- `?.` returns undefined if property doesn't exist (no error)

```
if (!token) next(new CustomError(401, "Please login to access this route"));
```

**What:** Check if token exists

**Why:** Reject unauthenticated requests

**Behind the Scene:**

- `next(error)` → skips to error middleware
- Doesn't execute remaining code in this function

```
const decodedToken = jwt.verify(token, process.env.JWT_SECRET_KEY);
```

**What:** Verify and decode JWT

**Why:** Ensure token is valid and not tampered

**Behind the Scene:**

- Checks signature using secret key
- Decodes payload (contains user ID)
- Throws error if invalid/expired

```
const user = await UserModel.findById(decodedToken.id);
```

**What:** Fetch user from database

**Why:** Verify user still exists

**Behind the Scene:**

- MongoDB query: `db.users.findOne({ _id: decodedToken.id })`
- Returns user document or null

```
req.myUser = user;
```

**What:** Attach user to request object

**Why:** Make user available to next middleware/controller

**Behind the Scene:** Mutates request object

**Flow:**

```
Request → Extract Token → Verify → Fetch User → Attach to req → next()
```

```
next();
```

**What:** Pass control to next middleware

**Why:** Continue request processing

**Behind the Scene:** Calls next function in middleware chain

---

## 6 validate.middleware.js - Input Validation

```
export const validate = (schema) => {
```

**What:** Higher-order function (returns a function)

**Why:** Reusable validation with different schemas

**Behind the Scene:** Closure captures `schema` parameter

```
return (req, res, next) => {
```

**What:** Returns middleware function

**Why:** Express needs a function with (req, res, next)

**Behind the Scene:** This is the actual middleware that executes

**Pattern:**

```
validate(schema) → returns middleware → Express uses it
```

```
const { error, value } = schema.validate(req.body, {
  abortEarly: false,
});
```

**What:** Joi validation

**Why:** Validate request body against schema

**Behind the Scene:**

- `abortEarly: false` → collect ALL errors, not just first one
- Returns `error` object if validation fails
- Returns sanitized `value` if succeeds

```
if (error) {
  next(new CustomError(400, `#${error.details.map((ele) => ele.message)}`));
}
```

**What:** Handle validation errors

**Why:** Return meaningful error messages

**Behind the Scene:**

- `error.details` is array of all validation errors
- `.map()` extracts just the messages
- Template literal converts to string

```
req.body = value;
```

**What:** Replace req.body with validated value

**Why:** Joi sanitizes data (removes extra fields, type coercion)

**Behind the Scene:** Overwrites original request body

## 7 user.controller.js - Business Logic

```
export const registerUser = expressAsyncHandler(async (req, res, next) => {
```

**What:** Controller function

**Why:** Handle registration logic

**Behind the Scene:** `expressAsyncHandler` wraps function to catch errors

```
const { username, email, password, contactNumber } = req.body;
```

**What:** Destructuring

**Why:** Extract values from req.body

**Behind the Scene:** JavaScript destructuring syntax

```
const newUser = await UserModel.create({
  username,
  email,
  password,
  contactNumber,
});
```

**What:** Create user in database

**Why:** Store user data

**Behind the Scene:**

- Mongoose calls MongoDB insert
- Schema validation runs
- Pre-save hooks execute (password hashing!)
- Returns created document

```
let emailVerificationToken = newUser.generateEmailVerificationToken();
```

**What:** Generate token for email verification

**Why:** Verify user's email address

**Behind the Scene:**

- Calls instance method on user document
- Creates random token
- Hashes and stores in database
- Returns unhashed token (to send via email)

```
await newUser.save();
```

**What:** Save updated user

**Why:** Token fields were modified

**Behind the Scene:**

- MongoDB update operation
- Pre-save hooks run again (but password not modified, so not hashed again)

```
new ApiResponse(201, "User Registered Successfully", newUser).send(res);
```

**What:** Send success response

**Why:** Inform client of success

**Behind the Scene:**

- Creates ApiResponse instance
- Calls `.send()` method
- Sends JSON response

```
export const loginUser = expressAsyncHandler(async (req, res, next) => {
```

```
let existingUser = await UserModel.findOne({ email });
```

**What:** Find user by email

**Why:** Check if user exists

**Behind the Scene:** MongoDB query `db.users.findOne({ email: email })`

```
if (!existingUser) next(new CustomError(400, "Email Not Found!!!!"));
```

**What:** Error handling

**Why:** User doesn't exist

**Behind the Scene:** Jumps to error middleware

```
let matchPassword = await existingUser.comparePassword(password);
```

**What:** Verify password

**Why:** Authenticate user

**Behind the Scene:**

- Calls instance method
- Uses bcrypt.compare() to check hashed password

```
let token = generateToken(existingUser._id);
```

**What:** Create JWT token

**Why:** Maintain user session

**Behind the Scene:**

- Signs payload with secret key
- Sets expiration time
- Returns signed token string

```
res.cookie("token", token, {
  maxAge: process.env.JWT_TOKEN_EXPIRY * 60 * 60 * 1000,
  httpOnly: true,
});
```

**What:** Set cookie with token

**Why:** Store token in browser

**Behind the Scene:**

- Sends **Set-Cookie** HTTP header
- **maxAge** → cookie lifetime in milliseconds
- **httpOnly: true** → prevents JavaScript access (XSS protection)

**Why cookie?** More secure than localStorage for storing tokens.

```
export const updateProfile = expressAsyncHandler(async (req, res, next) => {
```

```
const updatedUser = await UserModel.findByIdAndUpdate(
  req.myUser._id,
  req.body,
  {
    new: true,
    runValidators: true,
  }
);
```

**What:** Update user document

**Why:** Modify user data

**Behind the Scene:**

- `req.myUser._id` → from authenticate middleware
- `req.body` → validated data from validate middleware
- `new: true` → returns updated document (not old one)
- `runValidators: true` → runs schema validation on update

**Why options needed?**

- Without `new: true` → returns OLD document before update
- Without `runValidators` → validation doesn't run on updates

```
export const changePassword = expressAsyncHandler(async (req, res, next) => {
```

```
  const existingUser = await UserModel.findById(req.myUser._id);
```

**What:** Fetch full user document

**Why:** Need to modify and save (to trigger pre-save hook)

**Behind the Scene:** Gets complete document from database

```
  existingUser.password = req.body.password;
  await existingUser.save();
```

**What:** Update password and save

**Why:** Trigger pre-save middleware to hash password

**Behind the Scene:**

- Sets new password (plain text)
- `.save()` triggers pre-save hook
- Hook detects password changed
- Hashes password before saving

**Why not `findByIdAndUpdate`?** Because it doesn't trigger pre-save hooks! Password wouldn't be hashed.

---

## 8 user.model.js - Data Schema & Methods

```
const userSchema = new mongoose.Schema({ ... }, { timestamps: true, toJSON: { ... }, toObject: { ... } });
```

**What:** Define user structure

**Why:** Validate and structure data

**Behind the Scene:**

- `timestamps: true` → adds `createdAt`, `updatedAt` fields automatically
- `toJSON` → transforms document when converted to JSON
- `toObject` → transforms when converted to plain object

```
toJSON: {
  transform(doc, ret) {
    delete ret.password;
    delete ret.__v;
    ret.id = ret._id;
    delete ret._id;
  },
},
```

**What:** Customize JSON output

**Why:** Remove sensitive data, clean response

**Behind the Scene:**

- Runs automatically when `.toJSON()` called (like in `res.json()`)
- `doc` → original document
- `ret` → object to return
- Modifies `ret` in place

**Result:**

```
// Before
{ _id: "123", password: "hash", __v: 0, username: "john" }

// After
{ id: "123", username: "john" }
```

```
userSchema.pre("save", async function (next) {
```

**What:** Pre-save hook (middleware)

**Why:** Hash password before saving

**Behind the Scene:** Executes BEFORE document saved to database

```
if (!this.isModified("password")) return next();
```

**What:** Check if password changed

**Why:** Don't rehash unchanged passwords

**Behind the Scene:**

- `.isModified()` → Mongoose method
- Tracks which fields changed
- If password not modified → skip hashing

**Why important?** When updating profile (not password), don't rehash the already-hashed password!

```
let salt = await bcrypt.genSalt(10);
this.password = await bcrypt.hash(this.password, salt);
```

**What:** Hash password

**Why:** Security (never store plain text passwords)

**Behind the Scene:**

- `genSalt(10)` → creates random salt (10 rounds =  $2^{10}$  iterations)
- `hash()` → combines password + salt + hashing algorithm
- Result: irreversible hash

**Example:**

```
Plain: "password123"
Hash: "$2a$10$N9qo8uL0ickgx2ZMRZoMyeIjZAgcf17p921dGxad68LJZdL17lhWY"
```

```
next();
```

**What:** Continue save operation

**Why:** Tell Mongoose to proceed

**Behind the Scene:** Finishes middleware chain

```
userSchema.methods.comparePassword = async function (enteredPassword) {
  return await bcrypt.compare(enteredPassword, this.password);
};
```

**What:** Instance method

**Why:** Verify password during login

**Behind the Scene:**

- Available on every user document
- `this` → current user document
- `bcrypt.compare()` → hashes entered password with same salt and compares

**Usage:**

```
user.comparePassword("password123") → true/false
```

```
userSchema.methods.generateEmailVerificationToken = function () {
```

**What:** Generate token for email verification

**Why:** Verify user owns email

**Behind the Scene:** Two-token system

```
const randomBytes = crypto.randomBytes(32).toString("hex");
```

**What:** Create random token

**Why:** Unpredictable token

**Behind the Scene:**

- Creates 32 random bytes
- Converts to hexadecimal string
- Result: 64 character string

```
this.emailVerificationToken = crypto
  .createHash("sha256")
  .update(randomBytes)
  .digest("hex");
```

**What:** Hash the token

**Why:** Store securely in database

**Behind the Scene:**

- SHA-256 hashing algorithm
- One-way hash (can't reverse)
- Stores hash in database

```
this.emailVerificationTokenExpiry = Date.now() + 10 * 60 * 1000;
```

**What:** Set expiration (10 minutes)

**Why:** Token shouldn't work forever

**Behind the Scene:** Current timestamp + 10 minutes in milliseconds

```
return randomBytes;
```

**What:** Return unhashed token

**Why:** Send this in email link

**Behind the Scene:** This goes in verification URL

### Two-Token System:

1. randomBytes → sent in email
2. hashed version → stored in database

When user clicks link:

1. Extract randomBytes from URL
2. Hash it
3. Compare with database hash
4. If match → verify email

**Why this approach?** If database is compromised, attacker can't use tokens (they're hashed).

## 9 error.middleware.js - Global Error Handler

```
export const errorMiddleware = (err, req, res, next) => {
```

**What:** Error handling middleware

**Why:** Centralized error handling

**Behind the Scene:** Express calls this when `next(error)` is called

**4 parameters required!** Express recognizes error middleware by signature.

```
let statusCode = err.statusCode || 500;
let message = err.message || "something went wrong";
```

**What:** Default values

**Why:** Handle unexpected errors

**Behind the Scene:** If error doesn't have statusCode/message, use defaults

```
if (err.name === "ValidationError") {
  statusCode = 400;
  message = `${Object.values(err.errors).map((ele) => ele.message)}`;
```

**What:** Handle Mongoose validation errors

**Why:** Send meaningful error messages

**Behind the Scene:**

- Mongoose throws ValidationError with **errors** object
- Each field has its error message
- Extract all messages

```
if (err.code === 11000) {
  statusCode = 409;
  message = `${Object.keys(err.keyValue)[0]} already used`;
```

**What:** Handle MongoDB duplicate key errors

**Why:** Unique constraint violations (email already exists)

**Behind the Scene:**

- 11000** → MongoDB error code for duplicates
- keyValue** → contains the duplicate field

**Example:**

```
Error: email "john@email.com" already exists
Response: "email already used"
```

```
if (err.name === "CastError") {
  statusCode = 400;
  message = "Invalid MongoDB ID";
}
```

**What:** Handle invalid ObjectId

**Why:** User sends malformed ID

**Behind the Scene:** Mongoose throws CastError when ID format is wrong

```
if (err.name === "JsonWebTokenError") {
  statusCode = 401;
  message = "Invalid Session, Please login again";
}
```

**What:** Handle JWT errors

**Why:** Invalid/tampered tokens

**Behind the Scene:** jwt.verify() throws this error

```
res
  .status(statusCode)
  .json({ success: false, message, errObj: err, errLine: err.stack });
```

**What:** Send error response

**Why:** Inform client

**Behind the Scene:**

- `err.stack` → full error stack trace (for debugging)

## 10 Utility Files

### ApiResponse.util.js

```
class ApiResponse {
  constructor(statusCode, message, data) {
    this.statusCode = statusCode;
    this.message = message;
    this.data = data;
  }

  send(res) {
    let responseObject = {
      success: true,
      message: this.message,
    };

    if (this.data) {
      responseObject.data = this.data;
    }

    res.status(this.statusCode).json(responseObject);
  }
}
```

**What:** Standardized response format

**Why:** Consistent API responses

**Behind the Scene:** Builder pattern

**Usage:**

```

new ApiResponse(200, "Success", userData).send(res);

// Sends:
{
  success: true,
  message: "Success",
  data: { ...userData }
}

```

## CustomError.util.js

```

class CustomError extends Error {
  constructor(statusCode, message) {
    super();
    this.message = message;
    this.statusCode = statusCode;
  }
}

```

**What:** Custom error class

**Why:** Add statusCode to errors

**Behind the Scene:** Extends built-in Error class

**Usage:**

```

throw new CustomError(404, "User not found");
// or
next(new CustomError(401, "Unauthorized"));

```

## jwt.util.js

```

export const generateToken = (id) => {
  return jwt.sign({ id }, process.env.JWT_SECRET_KEY, {
    expiresIn: "1d",
  });
};

```

**What:** Create JWT token

**Why:** User authentication

**Behind the Scene:**

- Payload: `{ id: userId }`
- Signs with secret key
- Sets expiration

**JWT Structure:**

```
header.payload.signature
eyJhbGc.eyJpZCI6Ij.SflKxwRJ
```

Decoded payload: { id: "123", iat: 1234567890, exp: 1234654290 }

## user.validator.js

```
export const registerSchema = Joi.object({
  username: Joi.string().min(5).max(50).required(),
  email: Joi.string().min(5).max(50).required().email(),
  password: Joi.string().min(5).max(50).required(),
  contactNumber: Joi.string()
    .length(10)
    .required()
    .pattern(/^[6-9]\d{9}$/)
    .message("Invalid Mobile Number"),
});
```

**What:** Validation schema

**Why:** Input validation rules

**Behind the Scene:** Joi validates against these rules

**Rules Explained:**

- `min(5)` → minimum length
- `email()` → valid email format
- `pattern(/^[6-9]\d{9}$/)` → Indian mobile number (starts with 6-9, 10 digits)

## Behind the Scenes Concepts

### 🔍 Middleware Chain

Middleware functions execute **sequentially**:

```
Request
  ↓
cookieParser() → parses cookies → req.cookies available
  ↓
express.json() → parses JSON → req.body available
  ↓
authenticate() → verifies token → req.myUser available
  ↓
validate() → validates data → sanitized req.body
  ↓
controller() → business logic → sends response
  ↓
Response
```

If any middleware calls `next(error)` → jumps to error middleware.

## Password Security Flow

### Registration:

```
User sends: password123
↓
Pre-save hook triggers
↓
Generate salt (random string)
↓
Hash password + salt
↓
Store in DB: $2a$10$abc...xyz
```

### Login:

```
User sends: password123
↓
Fetch hashed password from DB
↓
bcrypt.compare(password123, hashedPassword)
↓
Hashes password123 with SAME salt
↓
Compares results
↓
Returns true/false
```

## JWT Authentication Flow

### Login:

1. User provides email + password
2. Server verifies credentials
3. Server creates JWT with user ID
4. Server sends JWT in cookie
5. Browser stores cookie

### Protected Routes:

1. Browser sends cookie with request
2. authenticate middleware extracts token
3. jwt.verify() checks signature
4. If valid → decode payload → get user ID
5. Fetch user from DB
6. Attach user to req.myUser
7. Continue to controller

## Request-Response Cycle

1. Client sends HTTP request  
↓
2. Express receives request  
↓
3. cookieParser → parses cookies  
↓
4. express.json() → parses body  
↓
5. Router matches path  
↓
6. Middleware chain executes  
↓
7. Controller handles logic  
↓
8. Database operations  
↓
9. Response sent back  
↓
10. Client receives response

## MVC Pattern (Modified)

This project uses a **modified MVC** pattern:

```
Routes (R) → Controllers (C) → Models (M)
      ↓
Middlewares (additional layer)
```

- **Routes:** Define endpoints
- **Middlewares:** Process requests (auth, validation)
- **Controllers:** Business logic
- **Models:** Database interaction

## Important Terminologies

### Express Concepts

#### app.use()

- **What:** Registers middleware
- **Why:** Add functionality to request pipeline
- **Where:** app.js (before routes)
- **Order:** Top to bottom (matters!)

#### Middleware

- **What:** Function with (req, res, next)

- **Why:** Process requests before reaching controller
- **Types:** Application-level, router-level, error-handling

## Router

- **What:** Mini Express application
  - **Why:** Modular route organization
  - **Usage:** `const router = Router()`
- 

# MongoDB/Mongoose

## Schema

- **What:** Structure definition for documents
- **Why:** Validation and type casting
- **Example:** `username: { type: String, required: true }`

## Model

- **What:** Class based on schema
- **Why:** Interact with database
- **Usage:** `UserModel.find()`, `UserModel.create()`

## Document

- **What:** Instance of model
- **Why:** Represents single record
- **Example:** `const user = await UserModel.findById(id)`

## Pre-save Hook

- **What:** Middleware at schema level
- **Why:** Execute code before saving
- **Usage:** Password hashing

## Instance Methods

- **What:** Methods available on documents
  - **Why:** Document-specific operations
  - **Example:** `user.comparePassword()`
- 

# Authentication

## JWT (JSON Web Token)

- **Structure:** header.payload.signature
- **Usage:** Stateless authentication
- **Storage:** httpOnly cookie (secure)

## Hashing

- **What:** One-way encryption
- **Why:** Can't reverse to get original value
- **Libraries:** bcrypt, crypto

## Salt

- **What:** Random string added to password
- **Why:** Same password → different hashes
- **Example:** Two users with "password123" get different hashes

## Token Verification

- **What:** Checking JWT signature
  - **Why:** Ensure token not tampered
  - **How:** Uses secret key
- 

## Async Programming

### async/await

- **What:** Syntactic sugar for Promises
- **Why:** Makes async code look synchronous
- **Example:**

```
// Without async/await
UserModel.findById(id)
  .then(user => console.log(user))
  .catch(err => console.log(err));

// With async/await
try {
  const user = await UserModel.findById(id);
  console.log(user);
} catch (err) {
  console.log(err);
}
```

## Promise

- **What:** Object representing eventual completion
- **States:** Pending → Fulfilled / Rejected
- **Methods:** .then(), .catch(), .finally()

### expressAsyncHandler

- **What:** Wrapper for async route handlers
- **Why:** Automatically catches errors
- **Without it:** Must use try-catch in every controller

# HTTP Concepts

## Status Codes

- **200:** Success
- **201:** Created (registration)
- **400:** Bad Request (validation error)
- **401:** Unauthorized (auth required)
- **404:** Not Found
- **409:** Conflict (duplicate)
- **500:** Server Error

## HTTP Methods

- **GET:** Retrieve data
- **POST:** Create data
- **PATCH:** Partial update
- **PUT:** Full update
- **DELETE:** Remove data

## Cookies

- **What:** Small data stored in browser
- **Why:** Maintain state (sessions)
- **Attributes:**
  - `httpOnly`: Prevents JavaScript access
  - `secure`: Only sent over HTTPS
  - `maxAge`: Expiration time

# JavaScript Concepts

## Destructuring

```
const { username, email } = req.body;
// Same as:
const username = req.body.username;
const email = req.body.email;
```

## Optional Chaining (?.)

```
const token = req?.cookies?.token;
// Safe access, returns undefined if any part is null/undefined
```

## Template Literals

```
`Database connected to ${client.connection.host}`  
// Embeds variables in strings
```

## Arrow Functions

```
// Regular function  
function add(a, b) { return a + b; }  
  
// Arrow function  
const add = (a, b) => a + b;
```

## Spread Operator

```
{ ...req.body, userId: req.myUser._id }  
// Copies all properties from req.body and adds userId
```

## Common Questions & Answers

### ? Why is order important in app.use()?

**Answer:** Middleware executes top-to-bottom. If you put routes before body parsers, `req.body` will be undefined!

#### Correct Order:

```
app.use(express.json());           // 1. Parse body  
app.use("/api/user", routes);    // 2. Then handle routes  
app.use(errorMiddleware);        // 3. Catch errors last
```

### ? Why separate app.js and server.js?

#### Answer:

- **app.js:** Configuration (can be tested independently)
- **server.js:** Startup logic (database, server listen)
- **Benefit:** Easier testing, cleaner code

### ? Why use cookies instead of localStorage?

#### Comparison:

Feature	Cookie ( <code>httpOnly</code> )	<code>localStorage</code>
JavaScript Access	✗ No	✓ Yes
XSS Vulnerability	✓ Protected	✗ Vulnerable
Sent with Requests	✓ Automatic	✗ Manual
Storage Limit	4KB	5-10MB

**Answer:** Cookies with `httpOnly` are more secure for storing authentication tokens.

---

## ? Why hash password in pre-save hook?

**Answer:** So hashing happens automatically whenever password changes, whether during registration or password update. Don't need to remember to hash it in every controller.

---

## ? Why check `isModified("password")`?

**Answer:** When updating profile (not password), the pre-save hook still runs. Without this check, it would hash the already-hashed password!

```
// Scenario: Update username only
user.username = "newname";
await user.save(); // Pre-save hook runs

// Without check: Would hash the hash!
// With check: Skips hashing
```

## ? Why use `findByIdAndUpdate` vs `save()`?

### `findByIdAndUpdate`:

- ✓ Single database operation (faster)
- ✗ Doesn't trigger pre-save hooks
- **Use for:** Simple updates (profile)

### `save()`:

- ✗ Two operations (find + update)
- ✓ Triggers pre-save hooks
- **Use for:** Updates needing hooks (password)

## ? What happens when `next(error)` is called?

### Flow:

```

Controller: next(new CustomError(404, "Not Found"))
↓
Express skips remaining middleware
↓
Jumps to error middleware
↓
errorMiddleware(err, req, res, next)
↓
Sends error response

```

## ? Why validate before authenticate?

**Answer:** Fail fast! If data is invalid, reject immediately. No need to check authentication for bad requests.

**Efficient Order:**

```

router.post("/update",
  validate(schema),      // 1. Reject bad data (fast)
  authenticate,          // 2. Check auth (DB query)
  controller            // 3. Process (business logic)
);

```

## Complete Request Flow Example

### Scenario: User Login

#### 1. Client Request

```

POST /api/user/login HTTP/1.1
Content-Type: application/json

{
  "email": "john@example.com",
  "password": "password123"
}

```

#### 2. Express Receives Request

```

// app.js
app.use(express.json()); // Parses body
// req.body now = { email: "john@example.com", password: "password123" }

```

#### 3. Route Matching

```
// user.route.js
router.post("/login", validate(loginSchema), loginUser);
// Matches! Execute middleware chain
```

## 4. Validation Middleware

```
// validate.middleware.js
const { error, value } = loginSchema.validate(req.body);
// Checks: email is valid format, password length
// If valid: req.body = value (sanitized)
// If invalid: next(error) → jump to error middleware
```

## 5. Controller Execution

```
// user.controller.js - loginUser
const { email, password } = req.body;

// Find user in database
let existingUser = await UserModel.findOne({ email });
// MongoDB Query: db.users.findOne({ email: "john@example.com" })

// Verify password
let matchPassword = await existingUser.comparePassword(password);
// bcrypt.compare("password123", "$2a$10$hashedPassword")

// Generate token
let token = generateToken(existingUser._id);
// jwt.sign({ id: "userId" }, "secretKey", { expiresIn: "1d" })

// Set cookie
res.cookie("token", token, {
  maxAge: 24 * 60 * 60 * 1000, // 1 day
  httpOnly: true              // Secure
});

// Send response
new ApiResponse(200, "User Logged In Successfully").send(res);
```

## 6. Response Sent

```
HTTP/1.1 200 OK
Set-Cookie: token=eyJhbGc...; Max-Age=86400; HttpOnly
Content-Type: application/json

{
  "success": true,
  "message": "User Logged In Successfully"
}
```

## 7. Browser Stores Cookie

```
Cookie: token=eyJhbGc...
Automatically sent with future requests to this domain
```

## Scenario: Access Protected Route

### 1. Client Request

```
GET /api/user/current HTTP/1.1
Cookie: token=eyJhbGc...
```

### 2. Cookie Parser

```
// app.js
app.use(cookieParser());
// req.cookies = { token: "eyJhbGc..." }
```

### 3. Route Matching

```
// user.route.js
router.get("/current", authenticate, currentUser);
```

### 4. Authentication Middleware

```
// auth.middleware.js
const token = req.cookies.token;
// Extracts: "eyJhbGc..."

const decodedToken = jwt.verify(token, process.env.JWT_SECRET_KEY);
// Verifies signature, decodes payload
// decodedToken = { id: "userId", iat: ..., exp: ... }

const user = await UserModel.findById(decodedToken.id);
// Fetches user from database

req.myUser = user;
// Attaches user to request
next();
// Continue to controller
```

### 5. Controller Execution

```
// user.controller.js - currentUser
new ApiResponse(200, "User is Logged in").send(res);
```

### 6. Response

HTTP/1.1 200 OK

```
{
  "success": true,
  "message": "User is Logged in"
}
```

## Advanced Concepts Explained

### Security Best Practices in This Project

#### 1. Password Hashing

- Uses bcrypt (industry standard)
- Automatic via pre-save hook
- Never stores plain text

#### 2. JWT Storage

- httpOnly cookies (XSS protection)
- Not accessible via JavaScript
-  Should add `secure: true` in production (HTTPS only)

#### 3. Input Validation

- Joi validation before processing
- Prevents injection attacks
- Type coercion and sanitization

#### 4. Error Handling

- Never exposes sensitive data
- Generic error messages to client
-  Should remove `errObj` and `errLine` in production

#### 5. Authentication

- Token verification on protected routes
- Database check (user still exists)
-  Should implement token refresh

### Design Patterns Used

#### 1. MVC Pattern (Modified)

Models: Data structure & database logic  
Controllers: Business logic  
Routes: Endpoint definitions  
+ Middlewares: Request processing layer

## 2. Middleware Pattern

- Chain of responsibility
- Each middleware does one thing
- Can short-circuit with next(error)

## 3. Repository Pattern

- Mongoose models abstract database operations
- Controllers don't write raw queries

## 4. Dependency Injection

- Configuration via environment variables
- Easier testing and deployment

## 5. Error Handling Pattern

- Custom error class
  - Centralized error middleware
  - Consistent error responses
-

# Project Structure Explained

```

backend/
|
+-- src/
|   +-- config/           # Configuration files
|   |   |-- database.config.js  # DB connection
|   |   |-- nodemailer.config.js # Email setup
|   |
|   +-- controllers/       # Business logic
|   |   +-- user/
|   |   |   |-- user.controller.js
|   |   +-- shop/
|   |   |   |-- address.controller.js
|   |
|   +-- middlewares/       # Request processors
|   |   |-- auth.middleware.js    # Authentication
|   |   |-- validate.middleware.js # Input validation
|   |   |-- error.middleware.js   # Error handling
|   |
|   +-- models/            # Database schemas
|   |   |-- user.model.js
|   |   |-- address.model.js
|   |
|   +-- routes/             # API endpoints
|   |   +-- user/
|   |   |   |-- user.route.js
|   |   +-- shop/
|   |   |   |-- address.route.js
|   |
|   +-- utils/              # Helper functions
|   |   |-- ApiResponse.util.js
|   |   |-- CustomError.util.js
|   |   |-- jwt.util.js
|   |
|   +-- validators/          # Validation schemas
|   |   |-- user.validator.js
|
+-- app.js                  # Express app config
+-- server.js                # Entry point
+-- .env                      # Environment variables
+-- package.json               # Dependencies

```

## Why this structure?

- **Separation of concerns:** Each folder has specific responsibility
- **Scalability:** Easy to add new features
- **Maintainability:** Find code quickly
- **Testability:** Test components independently

## Mongoose Document Lifecycle

### 1. Creation

```
const user = new UserModel({ username, email, password });
// Document created in memory (not in DB yet)

await user.save();
// Triggers: validation → pre-save hooks → save to DB
```

## 2. Update (Method 1: save)

```
const user = await UserModel.findById(id);
user.username = "newname";
await user.save();
// Triggers: validation → pre-save hooks → update DB
```

## 3. Update (Method 2: findByIdAndUpdate)

```
await UserModel.findByIdAndUpdate(id, { username: "newname" });
// Direct DB update → NO hooks triggered
```

## 4. Query

```
const user = await UserModel.findById(id);
// Returns Mongoose document with all methods
```

## 5. Delete

```
await UserModel.findByIdAndDelete(id);
// Removes from database
```

## 🧪 Testing Considerations

### Unit Testing (test individual functions):

```
// Test password hashing
const user = new UserModel({ password: "test123" });
await user.save();
expect(user.password).not.toBe("test123");
expect(await user.comparePassword("test123")).toBe(true);
```

### Integration Testing (test API endpoints):

```
// Test registration endpoint
const response = await request(app)
  .post("/api/user/register")
  .send({ username, email, password, contactNumber });

expect(response.status).toBe(201);
expect(response.body.success).toBe(true);
```

## Performance Optimization

### 1. Database Indexes

```
// user.model.js
userSchema.index({ email: 1 }); // Speed up email lookups
```

### 2. Select Only Needed Fields

```
// Instead of fetching entire document
const user = await UserModel.findById(id).select("username email");
```

### 3. Connection Pooling

- Mongoose automatically manages connection pool
- Reuses connections instead of creating new ones

### 4. Async Operations

- All database operations use async/await
- Non-blocking I/O

## Environment Variables (.env)

```
# Server Configuration
PORT=5000

# Database
MONGODB_URI=mongodb://localhost:27017/myapp

# JWT
JWT_SECRET_KEY=your-secret-key-here
JWT_TOKEN_EXPIRY=24 # hours

# Nodemailer
NODEMAILER_HOST=smtp.gmail.com
NODEMAILER_PORT=587
NODEMAILER_SECURE=false
NODEMAILER_EMAIL=your-email@gmail.com
NODEMAILER_PASSWORD=your-app-password
```

## Why .env file?

- Keeps secrets out of code
- Different values for dev/prod
- Never committed to Git
- Easy configuration

## Incomplete Features in Project

**address.controller.js** needs implementation:

```
export const getAddresses = expressAsyncHandler(async (req, res, next) => {
    // TODO: Fetch all addresses for logged-in user
    const addresses = await AddressModel.find({ userId: req.myUser._id });
    new ApiResponse(200, "Addresses fetched", addresses).send(res);
});

export const getAddress = expressAsyncHandler(async (req, res, next) => {
    // TODO: Fetch single address
    const { id } = req.params;
    const address = await AddressModel.findOne({
        _id: id,
        userId: req.myUser._id
    });
    if (!address) return next(new CustomError(404, "Address not found"));
    new ApiResponse(200, "Address fetched", address).send(res);
});

export const updateAddress = expressAsyncHandler(async (req, res, next) => {
    // TODO: Update address
    const { id } = req.params;
    const address = await AddressModel.findOneAndUpdate(
        { _id: id, userId: req.myUser._id },
        req.body,
        { new: true, runValidators: true }
    );
    if (!address) return next(new CustomError(404, "Address not found"));
    new ApiResponse(200, "Address updated", address).send(res);
});

export const deleteAddress = expressAsyncHandler(async (req, res, next) => {
    // TODO: Delete address
    const { id } = req.params;
    const address = await AddressModel.findOneAndDelete({
        _id: id,
        userId: req.myUser._id
    });
    if (!address) return next(new CustomError(404, "Address not found"));
    new ApiResponse(200, "Address deleted").send(res);
});
```

## Missing Features:

1. Email verification implementation
2. Password reset functionality
3. Address routes not connected
4. Input validation for addresses

5. Logout doesn't invalidate token (stateless JWT issue)

---

## Key Takeaways

### What You Should Remember

1. **Middleware order matters** - Parse body before using it
  2. **Always use async/await** - Database operations are asynchronous
  3. **Hash passwords** - Never store plain text
  4. **Validate input** - Don't trust client data
  5. **Handle errors** - Use try-catch or error middleware
  6. **Separate concerns** - Routes, controllers, models
  7. **Use environment variables** - Never hardcode secrets
  8. **Document lifecycle** - Know when hooks trigger
  9. **JWT in httpOnly cookies** - More secure than localStorage
  10. **Check authentication** - Protect sensitive routes
- 

### Learning Path

#### Beginner → Intermediate:

1.  Understand basic Express (routing, middleware)
2.  Learn MongoDB/Mongoose (CRUD operations)
3.  Master async/await
4.  Implement authentication (JWT)
5.  Input validation (Joi)
6.  Error handling patterns

#### Intermediate → Advanced:

1.  Add refresh tokens
  2.  Implement rate limiting
  3.  Add caching (Redis)
  4.  Write tests (Jest, Supertest)
  5.  Deploy to production
  6.  Monitor and log (Winston, Morgan)
- 

## Final Notes

This project is a **solid foundation** for a Node.js backend. It demonstrates:

-  Clean architecture

- Security best practices (mostly)
- Proper error handling
- Scalable structure

### **Areas for improvement:**

- Add comprehensive tests
  - Implement missing features (email verification, password reset)
  - Add API documentation (Swagger)
  - Implement rate limiting
  - Add logging system
  - Use TypeScript for type safety
  - Add refresh token mechanism
- 

## Quick Reference

### Common Operations

#### **Create User:**

```
const user = await UserModel.create({ username, email, password });
```

#### **Find User:**

```
const user = await UserModel.findOne({ email });
const user = await UserModel.findById(id);
```

#### **Update User:**

```
const user = await UserModel.findByIdAndUpdate(id, data, { new: true });
```

#### **Delete User:**

```
await UserModel.findByIdAndDelete(id);
```

#### **Generate Token:**

```
const token = generateToken(userId);
```

**Verify Token:**

```
const decoded = jwt.verify(token, process.env.JWT_SECRET_KEY);
```

**Send Response:**

```
new ApiResponse(200, "Success", data).send(res);
```

**Send Error:**

```
next(new CustomError(400, "Error message"));
```

---

 You now have complete understanding of this Node.js project!

Every line, every concept, every flow explained. Use this as your reference guide while coding. 