**Backend Project Setup Notes (Vidora Example)**

## **1. Core Concepts**

Before we dive into files/folders, we must understand **why** they exist.

### **1.1 DB (Database)**

* **What it is:** A structured storage for your application’s data.
* **Why we need it:** So that data persists between app restarts (in-memory data would disappear when server restarts).
* **Types:**
  + SQL (MySQL, PostgreSQL) — Table based, relational.
  + NoSQL (MongoDB) — Document based, flexible schema.
* **For Vidora:** MongoDB stores users, videos, comments, likes, subscriptions, etc.

### **1.2 Model**

* **What it is:** The **blueprint** for how data looks and behaves in the database.
* **Why we need it:** Ensures consistency in data structure (e.g., a Video always has a title, url, uploaderId).
* **For Vidora:**  
  Example: video.model.js defines { title: String, description: String, url: String, uploadedBy: ObjectId }

### **1.3 Controller**

* **What it is:** The **logic layer** that handles requests from the client.
* **Why we need it:** Keeps routes clean and separates **business logic** from routing.
* **For Vidora:**  
  video.controller.js → Functions like:
  + uploadVideo(req, res)
  + getAllVideos(req, res)

### **1.4 Route**

* **What it is:** The **entry point** for HTTP requests.
* **Why we need it:** Defines **URLs and HTTP methods** for specific features.
* **For Vidora:**
  + /videos → GET → fetch videos
  + /videos/upload → POST → upload new video

### **1.5 Middleware**

* **What it is:** Functions that run **between** request and controller.
* **Why we need it:** For reusable logic (auth, logging, validation).
* **For Vidora:**  
  Example: auth.middleware.js checks if a user is logged in before letting them upload videos.

### **1.6 Utils**

* **What it is:** Helper functions that can be used anywhere.
* **Why we need it:** Avoid repeating code.
* **For Vidora:**  
  Example: generateVideoThumbnail(url) or formatVideoDuration(seconds)

## **2. Folder Structure in** src

src/

app.js → Main Express app setup

constants.js → App-wide constants (port numbers, env keys)

index.js → Entry point (starts the server)

controllers/ → Business logic for features

db/ → Database connection setup

middlewares/ → Reusable request middlewares

models/ → MongoDB Mongoose models (schemas)

routes/ → Express routes mapping

utils/ → Helper functions/utilities

## **3. Config Files**

### 3.1 .prettierrc

* **Purpose:** Code formatting rules for Prettier.
* **Your config:**
* {
* "singleQuote": false, // Use double quotes
* "bracketSameLine": true, // Keep HTML/JSX closing bracket on same line
* "tabWidth": 2, // Indentation width = 2 spaces
* "trailingComma": "es5", // Add commas where valid in ES5
* "semi": true // Always end with ;

}

### 3.2 .prettierignore

* **Purpose:** Tells Prettier which files/folders NOT to format.
* **Your ignore list:**
* /.vscode // VSCode settings
* /node\_modules // Installed packages (never format)
* ./dist // Build output
* \*.env // Environment variable files
* .env

.env.\* // All env variations

### 3.3 .gitignore

* **Purpose:** Tells Git what NOT to commit.
* Similar to .prettierignore but for **Git**, not Prettier.

## **4. Commands & Flags**

### **4.1** npm init

* Creates package.json → Stores metadata about the project (name, version, dependencies, scripts, etc.)

### **4.2** npm i **vs** npm i -D

* npm i <pkg> → Installs as **dependency** (needed for production).
  + Example: express, mongoose
* npm i -D <pkg> → Installs as **devDependency** (needed only for development).
  + Example: nodemon, prettier

### **4.3** git add .

* Stages all changes for commit.

### **4.4** git commit -m "message"

* Saves staged changes into Git history with a message.

### **4.5** git push -u origin main

* Pushes changes to remote GitHub repo.
* -u sets the **upstream** so future pushes don’t require specifying branch.

### **4.6 Nodemon**

* Watches files for changes and restarts the server automatically.
* Installed with npm i -D nodemon (only needed for dev).

### **4.7 Dev Dependency vs Dependency**

| **Type** | **Installed with** | **Used in Production?** | **Example** |
| --- | --- | --- | --- |
| Dependency | npm i | Yes | express, mongoose |
| DevDependency | npm i -D | No | nodemon, prettier |

## **5. Step-by-step Setup Process (Reusable)**

1. **Initialize project**  
   npm init → fill in name, description, author.
2. **Create repo & push**  
   git init → git remote add origin <url> → git push -u origin main
3. **Install backend dependencies**  
   npm i express mongoose cors dotenv
4. **Install dev dependencies**  
   npm i -D nodemon prettier
5. **Setup folder structure**
6. src/
7. controllers/
8. db/
9. middlewares/
10. models/
11. routes/
12. utils/
13. **Create entry files:** app.js, constants.js, index.js
14. **Configure prettier:** .prettierrc, .prettierignore
15. **Setup Git ignore:** .gitignore
16. **Write server startup code** in index.js
17. **Commit & push**

## **1. What’s a DevDependency?**

In Node.js (or any project using package.json):

* **Dependencies** = packages needed **for your app to run** in production.  
  Example: If you’re using **Express** to serve your app, it’s required when your app is live.
* **DevDependencies** = packages needed **only during development**, not in production.  
  Example: Testing libraries (like Jest), linters (like ESLint), or build tools (like Vite, Webpack).

**In package.json:**

"dependencies": {

"express": "^4.18.2"

},

"devDependencies": {

"eslint": "^8.0.0"

}

When installing:

* npm install <package> → goes to "dependencies"
* npm install <package> --save-dev or -D → goes to "devDependencies"

## **2. Production vs Development**

**Development Environment**

* You’re coding on your machine or test server.
* You use debugging tools, detailed error logs, hot-reloading, etc.
* You include **devDependencies** here.

**Production Environment**

* Your app is deployed for real users.
* You need performance, security, and minimal size.
* You don’t include devDependencies here (to reduce build size & avoid exposing dev tools).

**Rule of thumb:**

* If the code/package is **needed for the app to work** → dependencies
* If it’s **only needed while developing/testing** → devDependencies

## **3. Flags**

A **flag** is an extra option you pass to a command to modify how it runs.  
Example:

npm install express --save-dev

Here --save-dev is a **flag**. It tells npm **“install this as a devDependency”**.

Flags usually start with:

* - (short form) → -D (short for --save-dev)
* -- (long form) → --save-dev

**Examples of flags:**

npm run build --verbose # Shows more output

git commit -m "Message" # -m flag adds a commit message

node server.js --port=3000 # Custom port

## **1. Body**

* **What it is**  
  The body is the data sent by the client to the server inside the HTTP request.  
  It is **not visible in the URL**; it is in the request payload.
* **When used**  
  Usually in POST, PUT, or PATCH requests to send data like forms, JSON, or files.
* **Analogy**  
  Think of sending a letter: the **body** is the content written inside the envelope.
* **Example**  
  Request:
* POST /login
* Content-Type: application/json
* {
* "username": "paxto",
* "password": "1234"
* }

Express:

app.use(express.json()); // middleware to read JSON body

app.post('/login', (req, res) => {

console.log(req.body.username); // "paxto"

});

## **2. Cookies**

* **What it is**  
  Small pieces of data stored on the client’s browser, automatically sent with every request to the server.
* **Purpose**
  + Keep track of logged-in users (sessions)
  + Store preferences like theme or language
* **Analogy**  
  Like a stamp on your hand when entering a club — you don’t need to reintroduce yourself each time; they check the stamp.
* **Example**  
  Request header:
* Cookie: sessionId=abc123; theme=dark

Express with cookie-parser:

app.use(cookieParser());

app.get('/', (req, res) => {

console.log(req.cookies.sessionId); // "abc123"

});

## **3. Params (Route Parameters)**

* **What it is**  
  Dynamic variables embedded in the URL path.
* **Analogy**  
  Like placeholders in an address: /users/:id means “some specific user’s profile”.
* **Example**  
  URL:
* GET /users/42

Express:

app.get('/users/:id', (req, res) => {

console.log(req.params.id); // "42"

});

## **4. req.params**

* **Definition**  
  An object holding the route parameters from the URL.
* **Example**
* app.get('/products/:productId', (req, res) => {
* console.log(req.params); // { productId: '123' }
* });

## **5. req.body**

* **Definition**  
  Holds the parsed data from the request body.
* **Middleware needed**
  + express.json() for JSON
  + express.urlencoded() for form data
* **Example**
* app.use(express.json());
* app.post('/register', (req, res) => {
* console.log(req.body); // { username: 'paxto', password: 'secret' }
* });

## **6. req.cookies**

* **Definition**  
  An object with cookies from the client.
* **Requires**  
  cookie-parser middleware
* **Example**
* app.use(cookieParser());
* app.get('/dashboard', (req, res) => {
* console.log(req.cookies); // { authToken: 'xyz456' }
* });

## **7. cookie-parser**

* **What it does**  
  Reads cookies from the Cookie header and puts them in req.cookies.
* **Install**
* npm install cookie-parser
* **Use**
* import cookieParser from 'cookie-parser';
* app.use(cookieParser());

## **8. app.get(path, handler)**

* **What it does**  
  Handles HTTP GET requests for the given path.
* **Analogy**  
  Like a receptionist answering when you knock on a specific door.
* **Example**
* app.get('/about', (req, res) => {
* res.send('About Page');
* });

## **9. app.on(event, callback)**

* **What it does**  
  Listens for events on the app instance (e.g., errors).
* **Example**
* app.on('error', (err) => {
* console.error('Server error:', err);
* });

## **10. app.listen(port, callback)**

* **What it does**  
  Starts the Express server and listens for incoming requests on the given port.
* **Analogy**  
  Like opening the store doors for customers.
* **Example**
* app.listen(3000, () => {
* console.log('Server running on port 3000');
* });

## **11. app.use(middleware)**

* **What it does**  
  Mounts middleware functions that run **before** route handlers.
* **Why mostly used with middlewares**  
  Middlewares process or modify requests/responses, check authentication, log activity, etc.
* **Analogy**  
  Like airport security — every passenger (request) must pass through before entering the terminal (route).
* **Example**
* app.use(express.json()); // parse JSON body
* app.use(cookieParser()); // parse cookies
* app.use((req, res, next) => { // custom middleware
* console.log(req.method, req.url);
* next();
* });

## **Request Flow (Simplified)**

1. **Client sends request** → may have URL params, body, cookies
2. **app.use() middlewares run** → parse body, read cookies, log request
3. **Route handler** (like app.get, app.post) processes request
4. **Response is sent back**

## **app.js code explanation:**

import expres from "express";

Purpose: Imports the Express library, which is the web framework you use to create your server and routes.

Note: There is a typo here — it should be express, not expres.

import cors from "cors";

Purpose: Imports the cors middleware.

What is CORS?

Cross-Origin Resource Sharing (CORS) is a security feature in browsers that blocks requests from one domain to another unless explicitly allowed.

Why needed:

Your frontend (like localhost:3000) and backend (like localhost:5000) are on different origins during development. cors() allows your frontend to call your backend.

import cookieParser from "cookie-parser";

Purpose: Imports middleware that parses cookies from incoming requests.

After using cookieParser(), you can access cookies in req.cookies.

const app = expres();

Purpose: Creates an instance of an Express app.

Think of it as your “server object” — all routes, middlewares, and listeners attach to this app.

Correction: Should be express(), otherwise this will throw an error.

app.use(

cors({

origin: process.env.CORS\_ORIGIN || "http://localhost:3000",

credentials: true, // Allow cookies to be sent with requests

})

);

app.use(): Mounts middleware globally for all incoming requests.

cors({...}): Configures CORS policy.

origin: Which frontends are allowed to make requests.

Uses process.env.CORS\_ORIGIN if set, otherwise defaults to localhost:3000.

credentials: true: Allows cookies (like auth tokens) to be sent along with requests.

Analogy: Like giving a VIP pass to specific websites so they can enter your backend safely.

app.use(express.json({ limit: "16kb" })); // Parse JSON bodies

Purpose: Middleware that parses incoming JSON request bodies and stores them in req.body.

limit: "16kb": Limits body size to prevent huge requests that can crash your server.

Use case: Login forms, API POST requests with JSON data.

app.use(express.urlencoded({ extended: true, limit: "16kb" })); // Parse URL-encoded bodies

Purpose: Middleware that parses form submissions (application/x-www-form-urlencoded).

extended: true: Allows rich objects/arrays to be encoded in the URL-encoded format.

Use case: Standard HTML forms, form submissions from frontend apps.

app.use(express.static("public")); // Parse cookies

Purpose: Serves static files (images, CSS, JS) from the public folder.

Example: If you put logo.png in public, it can be accessed via http://localhost:PORT/logo.png.

Note: The comment // Parse cookies is incorrect here — this line has nothing to do with cookies.

app.use(cookieParser()); // Parse cookies

Purpose: Middleware that parses cookies from requests.

After this, you can access cookies like:

console.log(req.cookies); // { sessionId: 'abc123' }

export default app;

Purpose: Exports your configured Express app instance, so you can import it in index.js or wherever you start your server.

Summary Table

Line / Middleware Purpose / Use Case

express() Create Express app instance

cors() Allow frontend from another domain to access backend

credentials: true Enable cookies for cross-origin requests

express.json() Parse JSON request body → req.body

express.urlencoded() Parse HTML form data → req.body

express.static("public") Serve static files

cookieParser() Parse cookies → req.cookies

**Middleware in Express.js**

### **1. What is Middleware?**

* Middleware is a **function that runs in the middle** of receiving a client request and sending a response from your server.
* It can **inspect, modify, or act on requests or responses** before they reach your route handlers or after the response is generated.
* Middleware is essential because it allows you to **add functionality** without repeating code in every route.

**Analogy:**  
Imagine a request is a package going through a warehouse:

1. First worker checks the ID (authentication middleware).
2. Second worker inspects the contents (body parser).
3. Third worker logs information (logger).
4. The package reaches the main processing line (your route handler).

Middleware is each worker in that sequence.

### **2. How Middleware Works**

* Middleware functions in Express have this signature:

function middleware(req, res, next) {

// Do something with req or res

next(); // Pass control to next middleware or route handler

}

* **Parameters:**
  1. req → the request object, containing request data.
  2. res → the response object, used to send data back.
  3. next → a function that passes control to the **next middleware** or route.
* If you **do not call next()**, the request will stop there and the client will never get a response.

### **3. Types of Middleware**

1. **Application-level Middleware**
   * Runs on **all requests** or on requests to specific paths.
   * Example:

app.use((req, res, next) => {

console.log(`${req.method} ${req.url}`);

next();

});

1. **Route-level Middleware**
   * Runs only for specific routes.
   * Example:

const authMiddleware = (req, res, next) => {

if (!req.cookies.token) return res.status(401).send('Unauthorized');

next();

};

app.get('/dashboard', authMiddleware, (req, res) => {

res.send('Dashboard page');

});

1. **Third-party Middleware**
   * Provided by libraries like cors, cookie-parser, express.json().
   * Example:

app.use(cors()); // Enable cross-origin requests

app.use(cookieParser()); // Parse cookies from requests

1. **Error-handling Middleware**
   * Special middleware with **four parameters**: (err, req, res, next).
   * Used to catch and handle errors from other middleware or routes.

app.use((err, req, res, next) => {

console.error(err.stack);

res.status(500).send('Internal Server Error');

});

### **4. Why Middleware is Important**

* **Separation of Concerns:** Keep logging, authentication, validation, and parsing separate from route logic.
* **Reusability:** Write middleware once and apply it to multiple routes.
* **Flexibility:** Stack multiple middleware for a single route.
* **Control:** Decide whether to continue request flow (next()) or terminate early (res.send()).

### **5. Common Middleware Examples and Use Cases**

|  |  |  |
| --- | --- | --- |
| Middleware | Purpose | Example Use Case |
| express.json() | Parse JSON request body | Handling API POST requests |
| express.urlencoded() | Parse form submissions | HTML forms with application/x-www-form-urlencoded |
| cookie-parser | Parse cookies into req.cookies | Session authentication |
| cors() | Allow requests from other domains | Frontend (React) → Backend (Express) communication |
| morgan | Log incoming requests | Debugging and monitoring |
| Custom authentication middleware | Protect private routes | Only logged-in users can access /dashboard |

### **6. Request Flow with Middleware**

* Middleware sits **between the request and the route**:

Client Request --> Middleware 1 --> Middleware 2 --> Route Handler --> Middleware 3 (optional) --> Client Response

* Each middleware can:
  + Read request info (req)
  + Modify request or response (req.body, headers)
  + Terminate the request (res.send())
  + Pass control to next middleware using next()

**Example Flow:**

app.use((req, res, next) => {

console.log("Logging request");

next();

});

app.use((req, res, next) => {

console.log("Checking authentication");

next();

});

app.get('/hello', (req, res) => {

res.send('Hello World');

});

When GET /hello is called:

Logging request

Checking authentication

Hello World (response sent)

### **7. Key Takeaways**

* Middleware = **functions executed before or after routes**.
* Use app.use() for **global middleware**.
* Use route-specific middleware for **specific routes**.
* next() passes control to the **next function** in the chain.
* Middleware handles **parsing, logging, authentication, error handling, and more**.
* Middleware is **core to Express**, without it many features like JSON parsing, cookies, CORS, and authentication would be repetitive.

# ****What is Async Route Handling in Express?****

### **Normal (synchronous) route example**

app.get('/users/:id', (req, res) => {

const user = User.findById(req.params.id); // pretend this is instant

res.json(user);

});

* Works fine if everything happens instantly.
* The server does its job **like a cashier at a small shop**: someone asks for info, you hand it over immediately.

### **The problem with async operations**

In real-world apps, most tasks are **slow**:

* Fetching a user from a database
* Calling another API
* Reading/writing files

Example:

app.get('/users/:id', async (req, res) => {

const user = await User.findById(req.params.id); // takes time

if (!user) throw new Error('User not found');

res.json(user);

});

* The await pauses your code until the database responds.
* If an **error happens** (e.g., DB is down or user not found), **Express does NOT catch it automatically**.
* The server might **crash** or the request might **hang forever**.

**Analogy:**

Think of your server as a **restaurant waiter**.

* Normal request: A customer asks for water → you hand it over immediately.
* Async request: A customer asks for a pizza from another city → it takes time. If the pizza place messes up, the waiter doesn’t know what to do. The customer waits forever or gets frustrated.

# ****2️ How Async Route Handling Solves This****

We wrap async functions in a **helper** called asyncHandler.

**Why:**

* Catches any errors from async operations
* Passes them to Express’s **central error handler**
* No need to write try/catch in every single route

**Analogy:**

* Think of asyncHandler as a **manager** for the waiter.
* If the pizza place messes up, the manager immediately handles it, informs the customer, and logs the issue.

# ****3️ First Approach (Promise.resolve + next)****

const asyncHandler = (requestHandler) => {

return (req, res, next) => {

Promise.resolve(requestHandler(req, res, next)).catch(next);

};

};

**What it does:**

* Wraps your async route function in a promise.
* Any error goes to next().
* The **central error middleware** will handle it.

**Analogy:**

* Waiter asks the manager: “I can’t handle this order, please take over.”
* The manager (error middleware) decides what to tell the customer and logs it.

**Example:**

app.get('/users/:id', asyncHandler(async (req, res) => {

const user = await User.findById(req.params.id);

if (!user) throw new Error('User not found');

res.json(user);

}));

* If the user is not found, asyncHandler passes the error to your **central error handler**, not directly to the client.

**Syntax:**

const asyncHandler = (requestHandler) => {

(req, res, next) => {

Promise.resolve(requestHandler(req, res, next)).catch((err) => next(err));

};

};

**Breakdown**

const asyncHandler =

Declares a constant variable named asyncHandler.

This variable holds a function.

(requestHandler) => { ... }

Arrow function that takes one parameter called requestHandler.

requestHandler represents any route handler function you pass in later.

(req, res, next) => { ... }

This is another arrow function inside the first one.

It takes Express route parameters:

req → the HTTP request object

res → the HTTP response object

next → the function to call the next middleware

Promise.resolve(requestHandler(req, res, next))

Wraps the route handler call in a Promise to ensure any returned value (or thrown error) is handled as a promise.

.catch((err) => next(err))

If the promise fails, call next(err) to pass the error to Express's error middleware.

export default asyncHandler;

Exports this function as the default export from the file so you can import it elsewhere.

# ****4️ Second Approach (try/catch + response)****

const asyncHandler = (fn) => async (req, res, next) => {

try {

await fn(req, res, next);

} catch (error) {

res.status(error.status || 500).json({

success: false,

message: error.message || "Internal Server Error",

});

}

};

**What it does:**

* Uses try/catch inside the route.
* Sends **error response immediately** to the client.
* Does not use central middleware.

**Analogy:**

* Waiter sees the pizza is late → tells the customer directly: “Pizza failed, here’s an apology.”
* No manager involved, but it’s quick.

**Example:**

app.get('/users/:id', asyncHandler(async (req, res) => {

const user = await User.findById(req.params.id);

if (!user) throw new Error('User not found');

res.json(user);

}));

* The client immediately gets JSON: { success: false, message: "User not found" }.

# ****5️ Key Differences (Easy Version)****

|  |  |  |
| --- | --- | --- |
| Feature | First Approach | Second Approach |
| Handles errors? | ✅ Yes, central middleware | ✅ Yes, immediate JSON |
| Sends response? | ❌ Not directly | ✅ Directly to client |
| Best for | Big apps with centralized logging | Small APIs, quick errors |
| Flexibility | High | Low |
| Analogy | Manager handles problem | Waiter tells customer directly |

# ****6️ Why You Need Async Route Handling****

1. **Prevents crashes** when async code fails.
2. **Avoids repeating try/catch** in every route.
3. **Centralizes error handling** (first approach) for logging, monitoring, or custom messages.

**Real-world example:**

* A user signs in → your app queries the DB → if DB is down, asyncHandler ensures:
  + The app doesn’t crash
  + Client receives an error message
  + Error is logged for debugging

Breakdown

1. **const asyncHandler =**
   * Declares a **constant variable** named asyncHandler.
2. **(fn) =>**
   * Arrow function that takes **one parameter** (fn) representing a route handler.
3. **async (req, res, next) => { ... }**
   * Returns an **async arrow function**.
   * Async is important because it allows the use of **await** inside.
   * Receives Express **route parameters**:
     + req, res, next
4. **try { await fn(req, res, next); }**
   * Calls the route handler with await to **pause execution** until it completes.
5. **catch (error) { ... }**
   * If an error occurs, it **catches it** and sends a JSON response with status code 500 (or the one provided in error.status).

# ****error.status****

* **What it is:**  
  In JavaScript, an **error object** can have custom properties.  
  status is a **custom property** usually added to represent the **HTTP status code** of the error.
* **Example:**
* const error = new Error("Not Found");
* error.status = 404; // Custom property
* console.log(error.message); // "Not Found"
* console.log(error.status); // 404
* **Usage in Express:**
* app.get("/user/:id", async (req, res, next) => {
* const user = await getUser(req.params.id);
* if (!user) {
* const err = new Error("User not found");
* err.status = 404;
* return next(err); // passes error to error middleware
* }
* res.json(user);
* });

So error.status is **the HTTP status code you want to return with the error**.

# ****res.status****

* **What it is:**  
  res.status(code) sets the **HTTP response status code** that the client receives.
* **Example:**
* res.status(200).send("Success"); // 200 OK
* res.status(404).json({ message: "Not found" }); // 404 Not Found
* **Analogy:**  
  Think of **res.status** as a **label on a package** that tells the recipient the type of message:
  + 200 = All good
  + 404 = Not found
  + 500 = Internal server error

# ****Async / Await****

### **What it is:**

* async/await is **modern syntax to work with Promises** in a way that looks **synchronous**.
* Makes your code easier to read than chaining .then() and .catch().

### **How it works:**

1. Mark a function as async → it **automatically returns a Promise**.
2. Use await before a Promise → pauses execution until the Promise resolves.

### **Example:**

async function getData() {

const result = await fetchData(); // waits here until fetchData() completes

console.log(result);

}

* Without async/await:

fetchData().then(result => {

console.log(result);

}).catch(err => {

console.error(err);

});

**Analogy:**

* Promise = you order food online. You don’t know when it will arrive, but you can handle it when it does.
* async/await = you wait at the table for the food, but your code reads **line by line**, just like normal synchronous code.

# ****Promise****

* **What it is:**  
  A **Promise** is an object representing a value that **might not be available yet**, but will be **resolved or rejected in the future**.
* **Syntax:**

const myPromise = new Promise((resolve, reject) => {

const success = true;

if(success) {

resolve("Done!");

} else {

reject("Failed!");

}

});

myPromise.then(result => console.log(result)).catch(err => console.error(err));

* **States of a Promise:**
  1. **Pending** – Not finished yet
  2. **Resolved / Fulfilled** – Successfully completed
  3. **Rejected** – Failed

**Analogy:**

* Promise is like **ordering a package online**:
  + Pending → Package is on the way
  + Fulfilled → Package arrived
  + Rejected → Package lost

# ****How They Connect****

* async/await works with **Promises**.
* error.status is used with res.status to **send proper HTTP error codes**.
* Example combining all:

app.get("/user/:id", async (req, res) => {

try {

const user = await getUser(req.params.id); // async/await

if (!user) throw Object.assign(new Error("User not found"), { status: 404 });

res.status(200).json(user); // send success

} catch (error) {

res.status(error.status || 500).json({ message: error.message }); // send error

}

});

## res.status() **– Directly sets what the client sees**

* **What it does:**  
  res.status(code) tells Express: “Hey, when you send this response to the client, **use this HTTP status code**.”
* **Who sees it:**  
  The **client/browser** receives it directly.
* **Example:**

app.get('/ok', (req, res) => {

res.status(200).json({ message: 'Everything is fine!' });

});

* When the client calls /ok, they get:

{

"message": "Everything is fine!"

}

with **HTTP 200 OK**.

**Analogy:**

* Imagine you’re sending a letter. res.status() is the **stamp on the envelope** that says “Priority” or “Return to sender” — the recipient sees it.

## **2️** error.status **– Holds the status for internal use, often passed to** res.status() **later**

* **What it does:**  
  error.status is **just a property of an Error object**.  
  It **does not automatically go to the client**. Instead, it’s used in **error-handling logic** to decide what HTTP status code to send with the response.
* **Who “receives” it:**  
  The **error-handling middleware** (or your catch block) reads error.status and then uses it to set the response code via res.status(error.status).
* **Example:**

app.get('/user/:id', async (req, res, next) => {

try {

const user = await getUser(req.params.id);

if (!user) {

const error = new Error('User not found');

error.status = 404; // internal property

throw error; // passes to next(error)

}

res.status(200).json(user);

} catch (err) {

// error-handling middleware or catch block

res.status(err.status || 500).json({ message: err.message });

}

});

* Flow here:
  1. error.status = 404 → **stored in the Error object**.
  2. Catch block reads err.status → sends it to client using res.status(err.status).

**Analogy:**

* error.status is like a **note you attach to a damaged package** saying “return as fragile – 404”.
* The **delivery person (error handler)** reads that note and then decides which stamp (res.status) to put on the envelope so the recipient (client) understands.

## **3️ Key Difference**

|  |  |  |  |
| --- | --- | --- | --- |
| Property | Purpose | Who “receives” it | Automatic? |
| res.status(code) | Sets HTTP response code sent to client | Client directly | Yes |
| error.status | Stores status code for error handling | Server-side code (middleware/catch block) | No, must be used with res.status |

## **4️ Simple Analogy for Entire Flow**

* Client → Server request → Something goes wrong.
* Server detects an error:
  + error.status = 404 → “I internally know it’s a Not Found error.”
* Server then responds:
  + res.status(error.status) → sends **actual 404 code to client**.
* Without using res.status(), the client wouldn’t know the error code; error.status alone doesn’t automatically get sent.

# **Hooks and JWT**

## **Bcrypt (Password Hashing Library)**

**What is bcrypt?**

* A library for **secure password hashing**.
* Hashing = one-way transformation → original password cannot be recovered from hash.
* Includes **salt** (random value) to make identical passwords unique.

**Code in your model:**

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

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

this.password = await bcrypt.hash(this.password, 10);

next();

});

**Explanation step-by-step:**

1. pre("save") → this middleware runs **before saving** a user to MongoDB.
2. this.isModified("password") → ensures **only new or changed passwords** are hashed.
3. bcrypt.hash(this.password, 10) → hashes password with **10 salt rounds**.
   * Example: MyStrongPassword123 → $2a$10$kUzvPj4uZPlbG2...
4. next() → continues to save the document in DB.

**Login password check:**

userSchema.methods.isPasswordCorrect = async function(password) {

return await bcrypt.compare(password, this.password);

};

* Compares the **plain password** input by user with the **hashed password** in DB.
* Returns true → password matches, false → incorrect.

**Key takeaway:**

* Passwords are **never stored in plain text**.
* Even if the DB is breached, attackers cannot recover the original password.

## **2️ JWT (JSON Web Token)**

**Definition:**

* A **secure, self-contained token** used to identify users without storing sessions server-side.

**Structure:**

HEADER.PAYLOAD.SIGNATURE

1. **Header** → metadata about the token

{

"alg": "HS256",

"typ": "JWT"

}

1. **Payload** → contains user info (claims)

{

"\_id": "66bcdd0e4fae882c07d92b30",

"username": "johndoe",

"email": "john@example.com",

"fullName": "John Doe"

}

1. **Signature** → ensures token integrity

* Generated using **secret key** (ACCESS\_TOKEN\_SECRET or REFRESH\_TOKEN\_SECRET)
* Backend uses it to verify token authenticity

**Why JWT?**

* Server doesn’t need to store session data (stateless auth)
* Payload tells the backend **who the user is**

## **3️ Tokens**

**Access Token**

* Short-lived (e.g., 1 day)
* Full payload → backend knows which user is making requests
* Example usage: Authorization: Bearer <ACCESS\_TOKEN>
* Automatically expires → increases security

**Refresh Token**

* Long-lived (e.g., 10 days)
* Minimal payload → usually just \_id
* Stored securely (HTTP-only cookie on client & in DB)
* When access token expires, backend uses refresh token to **issue a new access token** without asking user to login again

**Analogy:**

* Password = your **passport** → proves identity once
* Access token = **entry ticket** → short-lived, allows movement
* Refresh token = **backup ticket** → get a new entry ticket without showing passport again

## **4️ Full User Schema (Vidora)**

### **Schema fields**

username, email, fullName, avatar, coverImage, watchHistory, password, refreshToken

* username, email → unique, indexed, lowercase
* watchHistory → references Video objects
* password → stored as hash
* refreshToken → stored to allow session continuation

### **5️ Signup Flow**

1. Client sends signup data:

{

"username": "JohnDoe",

"email": "john@example.com",

"fullName": "John Doe",

"avatar": "https://vidora.com/avatar.jpg",

"password": "MyStrongPassword123"

}

1. Mongoose creates a new User instance:

const user = new User(req.body);

await user.save();

1. pre("save") middleware hashes password → stored as hash in DB
2. Database entry example:

{

"\_id": "66bcdd0e4fae882c07d92b30",

"username": "johndoe",

"email": "john@example.com",

"fullName": "John Doe",

"avatar": "https://vidora.com/avatar.jpg",

"password": "$2a$10$kUzvPj4uZPlbG2...",

"refreshToken": null

}

### **6️ Login Flow**

1. Client sends:

{ "email": "john@example.com", "password": "MyStrongPassword123" }

1. Server finds user:

const user = await User.findOne({ email: req.body.email });

1. Password check:

const isMatch = await user.isPasswordCorrect(req.body.password);

1. If correct → generate tokens:

const accessToken = user.generateAccessToken();

const refreshToken = user.generateRefreshToken();

1. Tokens sent to client:

* Access token → stored in memory / headers
* Refresh token → stored in HTTP-only cookie and DB

### **7️ Access & Refresh Token Workflow**

**During usage:**

1. Access token is sent with each API request
2. Backend verifies token (jwt.verify)
3. If access token expired → client sends **refresh token** to get a new access token
4. If refresh token also expired → user must log in again

**Security:**

* Password never leaves client after login
* Tokens are disposable and replaceable
* Refresh tokens allow **continuous session without user annoyance**

### **8️ Notes Table**

|  |  |
| --- | --- |
| Concept | Purpose |
| bcrypt | Hashes passwords securely before storing |
| jwt | Creates signed tokens for stateless authentication |
| Access Token | Short-lived token for API authorization |
| Refresh Token | Long-lived token to refresh access token |
| Payload | User info inside JWT → backend knows who the user is |
| pre("save") | Middleware to hash password before saving |
| isPasswordCorrect | Verifies input password against DB hash |

### **9️ Flow Diagram (Textual)**

**Signup:**

[Client] → POST /signup → [Server]

- Hash password

- Save user in DB

- Return success

**Login:**

[Client] → POST /login

- Check user exists

- Compare password (bcrypt)

- Generate access & refresh tokens

- Send tokens to client

**API Request:**

[Client] → GET /upload-video

- Authorization: Bearer <ACCESS\_TOKEN>

- Server verifies JWT

- If expired, use refresh token to issue new access token

✅ **Summary for Notes**

* **Password:** hashed using bcrypt, never stored in plain text
* **Access Token:** short-lived, contains user info, sent with every request
* **Refresh Token:** long-lived, only user ID, used to refresh access token
* **JWT:** signed, verifiable, stateless token
* **Flow:** Signup → password hashed → login → bcrypt check → tokens generated → access token used for requests → refresh token issues new access token when expired

# ****Video Model (video.model.js)****

### **1️ Imports**

import mongoose, { Schema, model } from "mongoose";

import mongooseAggregatePaginate from "mongoose-aggregate-paginate-v2";

* mongoose → ODM (Object Data Modeling) library for MongoDB in Node.js.
* Schema → defines structure, types, and rules for documents.
* model → compiles schema into a Mongoose model to interact with MongoDB.
* mongoose-aggregate-paginate-v2 → plugin to **paginate aggregate queries** (important for large datasets like videos).

### **2️ Video Schema**

const videoSchema = new Schema(

{

videoFile: { type: String, required: true },

thumbnail: { type: String, required: true },

title: { type: String, required: true },

description: { type: String, required: true },

duration: { type: Number, required: true },

views: { type: Number, default: 0 },

isPublished: { type: Boolean, default: false },

owner: { type: Schema.Types.ObjectId, ref: "User" },

},

{ timestamps: true }

);

**Field Breakdown:**

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Type | Required / Default | Description |
| videoFile | String | Required | URL or path of the video file |
| thumbnail | String | Required | URL or path of video thumbnail image |
| title | String | Required | Video title |
| description | String | Required | Video description |
| duration | Number | Required | Video duration in seconds |
| views | Number | Default: 0 | Count of how many times video was watched |
| isPublished | Boolean | Default: false | Whether the video is live/public |
| owner | Schema.Types.ObjectId | Optional | Reference to the User who uploaded the video |
| timestamps | Mongoose option | Automatic | Adds createdAt and updatedAt fields automatically |

**Notes:**

* owner uses **MongoDB population** → links video to a User document.
* timestamps: true automatically tracks when video was **created and updated**.

### **3️⃣ Plugin:** mongooseAggregatePaginate

videoSchema.plugin(mongooseAggregatePaginate);

**Explanation:**

* Adds **aggregate query pagination** capability.
* Example: fetch 10 videos per page, sorted by newest first.
* Helps **scale large datasets** efficiently.

### **4️ Model Creation**

const Video = model("Video", videoSchema);

export default Video;

* model("Video", videoSchema) → creates a Mongoose model named "Video".
* export default Video → allows import in other files (controllers, routes, etc.).

**Usage Example:**

// Fetch videos with pagination

const aggregate = Video.aggregate([{ $match: { isPublished: true } }]);

const options = { page: 1, limit: 10 };

const result = await Video.aggregatePaginate(aggregate, options);

**5️ Key Takeaways**

1. **Mongoose Schema** defines structure, validation, and relationships.
2. **Aggregate Paginate Plugin** allows **efficient pagination** for large datasets.
3. **Timestamps** automatically track creation and updates.
4. **Owner Reference** links videos to the user who uploaded them.
5. **Default Values** prevent missing or undefined fields.
6. **Scalability**: prevents sending all videos at once → improves performance.

### **6️ Analogy for Understanding**

* Videos database = **library with 100,000 books**.
* Without pagination → you ask for all books at once → huge, slow, impractical.
* With pagination → ask for **10 books per shelf** → easy, fast, scalable.
* Owner reference → like “Author” tag on each book → connects books to the author.

# **How to upload file in backend**

## **Cloudinary**

**What it is:**

* Cloudinary is a **cloud-based media management platform**.
* It allows you to **store, manage, transform, and deliver images and videos** from the cloud without managing your own storage server.

**Key Features:**

1. **Storage:** Upload images and videos to the cloud instead of saving them on your server.
2. **Transformation:** Resize, crop, compress, or convert media on the fly.
3. **Delivery/CDN:** Fast global delivery using a CDN (Content Delivery Network).
4. **Management:** Organize media, create folders, and manage versions easily.
5. **Secure:** Supports authentication, private URLs, and secure uploads.

**Why use it:**

* Saves **server storage and bandwidth**.
* Automatically optimizes images/videos for **faster loading**.
* Makes building media-heavy apps like **Vidora (YouTube clone)** easier and scalable.

**Example Usage in Node.js:**

import { v2 as cloudinary } from "cloudinary";

cloudinary.config({

cloud\_name: process.env.CLOUD\_NAME,

api\_key: process.env.CLOUD\_API\_KEY,

api\_secret: process.env.CLOUD\_API\_SECRET,

});

// Uploading a video

const result = await cloudinary.uploader.upload("path/to/video.mp4", {

resource\_type: "video",

});

console.log(result.url); // Get the URL of uploaded video

## **2️ Multer**

**What it is:**

* Multer is a **Node.js middleware** for handling **multipart/form-data**, which is mainly used for **file uploads** (like images or videos).
* It processes incoming files **before they reach your route handler**.

**Key Features:**

1. **File Storage Options:** Can store files on disk (local server) or in memory.
2. **File Filtering:** Validate file types (e.g., only .jpg or .mp4).
3. **Multiple File Uploads:** Upload single or multiple files at once.
4. **Integration:** Works easily with Express routes.

**Why use it:**

* When a user uploads a video or avatar in Vidora, Multer **extracts the file from the request**, then you can either **store it locally** or **send it to Cloudinary**.

**Example Usage in Node.js + Express:**

import multer from "multer";

// Configure storage (memory for Cloudinary)

const storage = multer.memoryStorage();

const upload = multer({ storage: storage });

// Route to upload a video

app.post("/upload", upload.single("video"), async (req, res) => {

const file = req.file;

// You can now send file.buffer to Cloudinary

});

### **Summary Analogy:**

* **Multer** = “Waiter who takes your file from the client request to the kitchen (server)”
* **Cloudinary** = “Restaurant kitchen in the cloud that cooks, stores, and serves your file efficiently”

## **Cloudinary Upload File (**cloudinary.js**)**

**Purpose:**  
This file handles uploading files (images/videos) from your server to Cloudinary, a cloud-based media storage and optimization service.

**Imports:**

import { v2 as cloudinary } from "cloudinary"; // Cloudinary SDK

import fs from "fs"; // Node.js file system module to manage files

**Configuration:**

cloudinary.config({

cloud\_name: process.env.CLOUDINARY\_CLOUD\_NAME,

api\_key: process.env.CLOUDINARY\_API\_KEY,

api\_secret: process.env.CLOUDINARY\_API\_SECRET,

});

* **cloud\_name** → Your unique Cloudinary account identifier.
* **api\_key** → Used to authenticate requests.
* **api\_secret** → Secret key for secure authentication.
* **Using environment variables** is a best practice to keep credentials safe.

**Upload Function:**

const uploadOnCloudinary = async (localFilePath) => {

try {

if (!localFilePath) {

throw new Error("No file path provided for upload.");

}

const response = await cloudinary.v2.uploader.upload(localFilePath, {

resource\_type: "auto", // auto-detects image/video types

});

console.log("File uploaded successfully to Cloudinary.", response.url);

return response;

} catch (error) {

fs.unlinkSync(localFilePath); // Delete temp file if upload fails

}

};

**Step-by-step Explanation:**

1. **Check if file path exists** → prevents uploading nothing.
2. **Upload file** → cloudinary.v2.uploader.upload() sends file to Cloudinary.
   * resource\_type: "auto" automatically handles images, videos, or other media.
3. **Return response** → contains useful info like the file URL (response.url).
4. **Error Handling** → if upload fails, remove the local temporary file to free storage.

**Key Concept:**

* Cloudinary provides scalable, optimized storage for media.
* Uploading to Cloudinary avoids storing large media files directly on your server.

## **Multer File (**multer.middleware.js**)**

**Purpose:**  
This file handles **receiving file uploads from clients** (browser or mobile) and storing them temporarily on your server before uploading to Cloudinary.

**Import:**

import multer from "multer";

**Storage Setup:**

const storage = multer.diskStorage({

destination: function (req, file, cb) {

cb(null, "./public/temp/"); // folder to temporarily save uploaded files

},

filename: function (req, file, cb) {

const uniqueSuffix = Date.now() + "-" + Math.round(Math.random() \* 1e9);

cb(null, file.originalname); // keep original file name

},

});

**Explanation:**

1. **destination** → directory where uploaded files are temporarily saved.
2. **filename** → ensures unique filenames using timestamp and random number.

**Export Multer Middleware:**

export const upload = multer({

storage: storage,

});

**Step-by-step Workflow:**

1. Client sends a file via a POST request.
2. Multer middleware processes the file:
   * Saves it to ./public/temp/ temporarily.
   * Adds a unique filename to avoid conflicts.
3. Backend can then **call uploadOnCloudinary** to move the file to Cloudinary.
4. Optionally, the temp file can be deleted after successful upload.

## **3️⃣ Combined Flow (Using Both Files)**

1. **User uploads a video/image** via frontend.
2. **Multer receives** the file and stores it temporarily.
3. **Backend calls uploadOnCloudinary(localFilePath)** → file moves to Cloudinary.
4. **Cloudinary returns a URL** → you save this in MongoDB (Video model).
5. **Temp file deleted** → server storage stays clean.

**Analogy:**

* **Multer** → acts like a receptionist accepting your package at the front door (server).
* **Cloudinary** → is the warehouse where the package is stored and optimized.

✅ **Summary Notes for Paste:**

* **Cloudinary** → cloud-based media storage & optimization service.
* **Multer** → middleware to handle file uploads, stores files temporarily on server.
* **Workflow:** Client → Multer → Temp storage → Cloudinary → MongoDB → Client gets URL.
* **Security:** Use .env for API credentials.
* **Efficiency:** Cloudinary optimizes images/videos, reduces server load.

# **HTTP Crash Course**

## **HTTP – HyperText Transfer Protocol**

**Definition:**

* HTTP is a **protocol** (a set of rules) used by web browsers and servers to **communicate**.
* Think of it as a “language” that your browser and a web server both understand.

**Key Points:**

* **Text-based** → messages are sent in readable text format.
* **Stateless** → each request is independent; the server doesn’t remember previous requests automatically.
* Used primarily for transferring **web pages, images, videos, JSON, etc.**

**Structure of an HTTP Request:**

GET /index.html HTTP/1.1

Host: example.com

User-Agent: Chrome/117

Accept: text/html

**Structure of an HTTP Response:**

HTTP/1.1 200 OK

Content-Type: text/html

<html> ... </html>

**Example / Analogy:**

* Imagine sending a **letter via postal mail**. You (browser) write a letter (HTTP request) asking for something (like a page). The server receives it and sends a reply letter (HTTP response).

**Limitations:**

* **Not secure** → data is sent in plaintext. Anyone intercepting it (e.g., on public Wi-Fi) can read it.

## **2️ HTTPS – HyperText Transfer Protocol Secure**

**Definition:**

* HTTPS = HTTP + **SSL/TLS encryption**.
* It **secures communication** between your browser and the web server.

**Key Points:**

* Encrypts data using SSL/TLS → nobody can read it if intercepted.
* Uses **certificates** to verify the server’s identity.
* Commonly seen as https:// in URLs, often with a padlock symbol in browsers.

**Example / Analogy:**

* HTTPS is like sending your letter **in a sealed envelope with a lock**. Even if someone intercepts it, they can’t read the content.

**Difference Between HTTP and HTTPS:**

|  |  |  |
| --- | --- | --- |
| Feature | HTTP | HTTPS |
| Encryption | None | SSL/TLS encryption |
| Security | Vulnerable | Secure |
| Port | 80 | 443 |
| Identity | No verification | Verified via certificate |

## **3 URI – Uniform Resource Identifier**

**Definition:**

* URI is a **generic term** for identifying a resource on the web.
* Every URL is a URI, but **not every URI is a URL**.

**Types of URIs:**

1. **URL (Locator)** → specifies **where** the resource is.
2. **URN (Name)** → specifies **what** the resource is, without giving its location.

**Example of URI:**

* URL: https://example.com/videos/123 → tells you where the video is.
* URN: urn:isbn:978-3-16-148410-0 → identifies a book by its ISBN; no location.

**Analogy:**

* URI = a **name tag** on a resource: could either say **where to find it (URL)** or **just what it is (URN)**.

## **4 URL – Uniform Resource Locator**

**Definition:**

* URL is the **full address** used to access resources on the web.
* It **tells your browser exactly where to go** and how to get the resource.

**Structure of a URL:**

https://www.example.com:443/path/to/file.html?query=123#section2

* https → protocol (HTTP or HTTPS)
* www.example.com → domain name / host
* 443 → port (optional; default 443 for HTTPS)
* /path/to/file.html → path on server
* ?query=123 → query parameters (optional)
* #section2 → fragment identifier (optional; scroll to part of page)

**Analogy:**

* URL = your **home address**:
  + Protocol = how to reach the city (car, bike, walk)
  + Domain = city name
  + Path = street and house number
  + Query = specific instructions for delivery
  + Fragment = room inside the house

## **5️ URN – Uniform Resource Name**

**Definition:**

* URN is a **persistent, location-independent identifier**.
* Focuses on **uniquely naming a resource**, not locating it.

**Example:**

urn:isbn:978-3-16-148410-0

* This identifies a book **universally** by ISBN, but doesn’t tell you where to find it.

**Analogy:**

* URN = the **ID card or serial number** of an object: you know exactly what it is, not where it is.

## **6️ Quick Comparison Table**

|  |  |  |
| --- | --- | --- |
| Term | What it does | Example |
| HTTP | Protocol to transfer data | http://example.com |
| HTTPS | Secure protocol (HTTP + SSL/TLS) | https://example.com |
| URL | Complete address of resource | https://example.com/path/file.html |
| URI | Identifier of resource | https://example.com/path/file.html or urn:isbn:123 |
| URN | Name of resource (persistent) | urn:isbn:978-3-16-148410-0 |

## **Key Takeaways / Notes Version**

1. **HTTP:** basic protocol, plaintext, stateless, used for web communication.
2. **HTTPS:** secure HTTP, uses encryption and certificates, prevents eavesdropping.
3. **URL:** tells the **browser where to find a resource**; full address.
4. **URI:** identifies a resource; can be a URL (location) or URN (name).
5. **URN:** location-independent name; uniquely identifies a resource.

**Super Analogy Summary:**

* HTTP = sending a postcard
* HTTPS = sending a postcard in a sealed envelope
* URL = home address with street + house number
* URI = ID tag on your stuff (either location or just name)
* URN = serial number or ID card of the item

# ****Metadata in HTTP****

**Definition:**  
Metadata in HTTP is **additional information sent along with a request or response**, usually in the form of **key-value pairs**, that tells the client or server how to handle the data.

Think of it as a **label on a package**: it’s not the actual content, but it explains what the content is, how it should be treated, or who it’s from.

**Example:**

GET /video/123 HTTP/1.1

Host: vidora.com

Authorization: Bearer abc123

Accept: application/json

* Authorization → who is making the request
* Accept → client wants JSON

# ****2️ Caching****

**Definition:**  
Caching stores HTTP responses **temporarily** so that subsequent requests can be served faster without recomputing or refetching the data.

**Without caching:**

* Every request hits the server → slow, more load.
* Example: Loading the same video thumbnail repeatedly causes server to regenerate it every time.

**With caching:**

* First request: server generates response → stored in cache.
* Subsequent requests: served from cache → fast, less load.

**Key headers:**

* Cache-Control: max-age=3600 → cache for 1 hour
* ETag: "abc123" → unique identifier to check if response has changed

# ****3️ Authentication****

**Definition:**  
Verifying the identity of the user or client.

**How:**

* Usually via **headers** like Authorization.
* Common methods: JWT, Basic Auth, OAuth.

**Example:**

Authorization: Bearer <access\_token>

* Server checks token → if valid → grants access.

# ****4️ Managed States****

HTTP is **stateless**, meaning each request is independent.

**Managed states** allow remembering who the client is:

* **Cookies** → small data stored in the browser
* **Sessions** → server stores session ID → tracks logged-in users
* Prevents asking users to login every single request

# ****5️ X-Prefix Headers (Before 2012)****

* Headers starting with X- were **custom/non-standard headers**.
* Example: X-Requested-With: XMLHttpRequest for AJAX detection.
* Post-2012: use **standard headers** or custom headers without X-.

# ****6️ HTTP Headers Overview****

Headers = metadata sent **with requests and responses**.

## **6a) Request Headers** (Client → Server)

|  |  |  |
| --- | --- | --- |
| Header | Purpose | Example |
| Accept | Data format client wants | Accept: application/json |
| User-Agent | Browser/app info | User-Agent: Mozilla/5.0 |
| Authorization | Identity verification | Authorization: Bearer <token> |
| Content-Type | Type of request body | Content-Type: application/json |
| Cookie | Session info | Cookie: sessionId=abc123 |
| Cache-Control | Cache behavior | Cache-Control: no-cache |

## **6b) Response Headers** (Server → Client)

* Provide info about **the server, response, or caching**.

**Examples:**

* Cache-Control: max-age=3600 → client can cache response 1 hour
* Set-Cookie: sessionId=abc123; HttpOnly → set session cookie

## **6c) Representation Headers**

* Describe **the payload** (body) of the request/response.

**Types:**

1. **Encoding** → character set or byte encoding
2. Content-Encoding: gzip
3. **Compression** → reduces size of payload
4. Content-Encoding: gzip
5. **Payload headers** → size, length, type
6. Content-Type: application/json
7. Content-Length: 345

# ****7️ CORS (Cross-Origin Resource Sharing)****

**Problem:** Browsers block requests from one domain to another (security).

**Solution:** Servers set headers to allow it.

|  |  |  |
| --- | --- | --- |
| Header | Purpose | Example |
| Access-Control-Allow-Origin | Which domains can access | Access-Control-Allow-Origin: https://vidora.com |
| Access-Control-Allow-Credentials | Allow cookies/credentials | true |
| Access-Control-Allow-Methods | Allowed HTTP methods | GET, POST, PUT |

# ****8️ Security Headers****

1. **Cross-Origin-Embedder-Policy (COEP)**
   * Ensures only resources explicitly allowed can be loaded.
2. **Cross-Origin-Opener-Policy (COOP)**
   * Isolates your site from other cross-origin windows → protects against attacks like Spectre.
3. **Content-Security-Policy (CSP)**
   * Controls which scripts, styles, images can load.
   * Example: prevents XSS attacks.
4. **X-XSS-Protection**
   * Built-in browser filter to block some cross-site scripting attacks.

# ****Summary Analogy****

* **Metadata** → label on a package
* **Caching** → store frequently requested items in a warehouse
* **Authentication** → passport check
* **Managed states** → keeping a stamp in passport to remember your last entry
* **Headers** → instructions for delivery service
* **CORS** → “only deliver to allowed addresses”
* **Security headers** → “lock doors, install alarms, monitor entry”

# ****HTTP Methods:****

# ****1️ GET****

**Purpose:** Retrieve data from a server.

* **Safe:** Does not modify data on server.
* **Idempotent:** Calling it multiple times has the same effect as once.

**Example:**

GET /videos/123 HTTP/1.1

Host: vidora.com

* Server responds with the video data for ID 123.

**Analogy:**

* You go to a library and **ask for a book**; the book isn’t changed or deleted, you just read it.

# ****2️ HEAD****

**Purpose:** Same as GET, but **doesn’t return the body**, only headers.

* Useful to check metadata like content length, type, caching info, last modified.

**Example:**

HEAD /videos/123 HTTP/1.1

* Server responds with headers only.

**Analogy:**

* You ask a library **“Do you have this book?”** instead of borrowing it. You just check availability.

# ****3️ OPTIONS****

**Purpose:** Ask server **what HTTP methods are allowed** on a resource.

* Often used in **CORS preflight requests**.

**Example:**

OPTIONS /videos/123 HTTP/1.1

* Server responds:

Allow: GET, POST, OPTIONS

**Analogy:**

* You check **“Can I read, borrow, or buy this book?”** before actually taking any action.

# ****4️ TRACE****

**Purpose:** Echo the request back to the client.

* Rarely used, mainly for debugging.
* Can expose sensitive info, so often disabled in production.

**Example:**

TRACE /videos/123 HTTP/1.1

* Server responds with the request as it received it.

**Analogy:**

* Sending a mirror to the library to **see exactly what message they received from you**.

# ****5️ DELETE****

**Purpose:** Remove a resource from the server.

* **Idempotent:** Deleting the same resource multiple times has the same effect as once.

**Example:**

DELETE /videos/123 HTTP/1.1

* Deletes video with ID 123.

**Analogy:**

* You **return a library book and tell them to remove it from the catalog** permanently.

# ****6️ PUT****

**Purpose:** Update or replace a resource completely.

* **Idempotent:** Sending the same PUT request multiple times results in the same resource state.

**Example:**

PUT /videos/123 HTTP/1.1

Content-Type: application/json

{

"title": "New Title",

"description": "Updated description",

"duration": 300

}

* Replaces video 123’s details entirely.

**Analogy:**

* You **replace a whole book** with a new copy.

# ****7️ POST****

**Purpose:** Create a new resource or submit data to the server.

* **Not idempotent:** Sending multiple times can create multiple resources.

**Example:**

POST /videos HTTP/1.1

Content-Type: application/json

{

"title": "My New Video",

"description": "Awesome video",

"duration": 180

}

* Server creates a new video and returns its ID.

**Analogy:**

* You **donate a new book** to the library. Each donation adds a new book.

# ****8️ PATCH****

**Purpose:** Update **part of a resource**, not the whole thing.

* Used for small changes.

**Example:**

PATCH /videos/123 HTTP/1.1

Content-Type: application/json

{

"title": "Updated Title Only"

}

* Only updates the title field, keeps the rest intact.

**Analogy:**

* You **correct a typo in the library book title** without replacing the whole book.

# ****Summary Table****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method | Purpose | Safe | Idempotent | Typical Use |
| GET | Retrieve resource | ✅ | ✅ | Fetch video, user data |
| HEAD | Retrieve headers only | ✅ | ✅ | Check content length, caching info |
| OPTIONS | Discover allowed methods | ✅ | ✅ | CORS preflight, API exploration |
| TRACE | Echo request | ✅ | ✅ | Debugging |
| DELETE | Delete resource | ❌ | ✅ | Delete video, remove user |
| PUT | Replace resource completely | ❌ | ✅ | Update entire user profile |
| POST | Create new resource | ❌ | ❌ | Upload video, register user |
| PATCH | Update part of resource | ❌ | ❌ | Update single field like title |

💡 **Analogy for all methods:**

* **GET** → read
* **HEAD** → peek
* **OPTIONS** → ask rules
* **TRACE** → mirror request
* **DELETE** → remove
* **PUT** → replace
* **POST** → add new
* **PATCH** → edit

# ****What Are Routes?****

In **backend development**, a **route** is basically the **path + HTTP method** that defines **how your server should respond to a request** from a client.

Think of it as **the address + action** for your API.

**Example:**

GET /videos → Get all videos

POST /videos → Upload a new video

GET /videos/:id → Get details of one video

DELETE /videos/:id → Delete a specific video

Each route:

* **Has a path** (like /videos or /users/login)
* **Has a method** (GET, POST, PUT, DELETE, etc.)
* **Is linked to a controller** that handles the logic.

## **2️⃣ Analogy**

Imagine your **backend** is a restaurant:

* **Route** = The menu item (e.g., “Cheese Pizza”) + the action (“Order” or “Cancel”)
* **Controller** = The chef who actually prepares the pizza
* **Model** = The recipe & ingredients in the kitchen

The customer doesn’t care how the pizza is made — they just ask for it using the **menu path**.

## **3️⃣ Structure of a Route in Express**

import express from "express";

import { getAllVideos, uploadVideo } from "../controllers/video.controller.js";

const router = express.Router();

// Define routes

router.get("/videos", getAllVideos); // GET request to /videos

router.post("/videos", uploadVideo); // POST request to /videos

export default router;

### **Breaking it down**

1. **router.get("/videos", getAllVideos)**
   * Path = /videos
   * Method = GET
   * Controller = getAllVideos
2. **router.post("/videos", uploadVideo)**
   * Path = /videos
   * Method = POST
   * Controller = uploadVideo

## **4️⃣ Why Routes Are Important**

* They **define your API** — clients need to know where to send requests.
* They **separate endpoint definitions** from business logic (keeps code clean).
* They **make your backend organized** for scaling in big projects.

## **5️⃣ Without Routes vs With Routes**

### Without routes:

* All logic might be inside a single file like server.js.
* Messy, hard to maintain.

### With routes:

* Each feature has its own route file (video.routes.js, user.routes.js).
* Easy to understand, debug, and scale.

## **6️⃣ Notes for You**

**Routes (Backend)**

* Definition: **Path + HTTP method** telling the server what to do when a request comes in.
* Always connected to a **controller function**.
* Example:
  + GET /users → Get all users
  + POST /users → Create a user
* Organized via **express.Router()**.
* Analogy: Menu items in a restaurant → Customer chooses → Chef (controller) prepares.

# ****Routes, Routers and Controllers:****

# 🛠 1. ****Routes (the entry points, Road)****

👉 In Express, a **Route** is a combination of:

* **HTTP Method** (GET, POST, PUT, DELETE, PATCH, etc)
* **Path (URL)** (/users, /videos/:id)
* **Handler Function** (what happens when the request comes in)

A basic route looks like this:

app.get("/hello", (req, res) => {

res.send("Hello World");

});

### ⚡ Important points about routes:

1. **HTTP Methods**:
   * GET /users → get a list of users
   * POST /users → create a new user
   * PUT /users/:id → update a user
   * DELETE /users/:id → delete a user

👉 So the same path /users can mean different things based on the method.

1. **Dynamic Routes**:
2. app.get("/users/:id", (req, res) => {
3. res.send(`User ID is ${req.params.id}`);
4. });
   * :id is a **route parameter**.
   * If you visit /users/10, req.params.id = 10.
5. **Middleware Functions**:  
   A route can have multiple handlers (like a pipeline).
6. app.get("/secure", authMiddleware, (req, res) => {
7. res.send("You’re authenticated");
8. });
   * First authMiddleware runs → checks if the user is logged in.
   * If yes → moves to the next handler (res.send(...)).
   * If no → stops and sends back an error.

✅ **So a Route is the smallest unit — it’s like saying: “When a car enters from THIS road, THIS police officer will check it.”**

# 🛠 2. ****Router (organizer of routes, District in a city)****

👉 A **Router** is basically a **mini Express app** that you use to group related routes.  
Why? Because real apps have **dozens or hundreds of routes**, and putting them all inside app.js would make a mess.

Example:

### Without Router (messy app.js)

app.get("/users", ...);

app.post("/users/register", ...);

app.post("/users/login", ...);

app.get("/videos", ...);

app.post("/videos/upload", ...);

app.delete("/videos/:id", ...);

Ugly, right? All mixed.

### With Router (organized)

// user.routes.js

import { Router } from "express";

const router = Router();

router.get("/", (req, res) => res.send("All users"));

router.post("/register", (req, res) => res.send("Register user"));

router.post("/login", (req, res) => res.send("Login user"));

export default router;

// video.routes.js

import { Router } from "express";

const router = Router();

router.get("/", (req, res) => res.send("All videos"));

router.post("/upload", (req, res) => res.send("Upload video"));

export default router;

// app.js

import userRouter from "./routes/user.routes.js";

import videoRouter from "./routes/video.routes.js";

app.use("/api/v1/users", userRouter);

app.use("/api/v1/videos", videoRouter);

### What happens now:

* GET /api/v1/users/ → all users
* POST /api/v1/users/register → register a user
* GET /api/v1/videos/ → all videos
* POST /api/v1/videos/upload → upload a video

✅ **Router = a mini app that handles related endpoints**.  
Think of it like: a mall has multiple sections → clothing section, food court, electronics. Each section has its own entrance (router) but connects to the main mall (app).

# 🛠 3. ****Controllers (the logic, actual building that works on a particular thing)****

👉 A **Controller** is simply a function that contains the **business logic** for a route.

Without controllers:

router.post("/register", async (req, res) => {

const { name, email, password } = req.body;

// validate

// hash password

// save to DB

// send response

});

The problem → Routes get bloated with logic.

With controllers (cleaner):

// user.controller.js

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

const { name, email, password } = req.body;

// validate

// hash password

// save to DB

res.status(201).json({ success: true, message: "User registered" });

};

// user.routes.js

import { registerUser } from "../controllers/user.controller.js";

router.post("/register", registerUser);

### Why use controllers?

* **Separation of concerns**: Routes only define “which URL calls which function”.
* Controllers focus only on “what to do when called”.
* Easier to test controllers individually.
* Cleaner code → easier to maintain as project grows.

✅ **Controllers are the brains, Routes are the triggers.**  
If Router is the **map of the city**, Controller is the **actual worker doing the job inside the building**.

# 🧩 Putting It All Together (Analogy)

Let’s imagine **Vidora** (your YouTube-like app) is a **city**.

* **App.js (Express app)** → The city’s government. It organizes all districts.
* **Router** → A district in the city (Users district, Videos district, Comments district). Each has its own roads.
* **Route** → A specific road in the district (e.g., /register is a road in Users district).
* **Controller** → The building at the end of the road that actually provides the service (e.g., registering a user, uploading a video).

So:

1. A car (client request) enters the city.
2. Government (app.js) sends it to the right district (router).
3. Car drives to the correct road (route).
4. Inside the building at the end of that road, workers (controller) do the actual job.

# ⚡ Key Differences Recap

* **Route**: Defines a single endpoint (method + path + handler).
* **Router**: Groups multiple routes into a mini app. Keeps things organized.
* **Controller**: The actual logic for handling a request.

# 📂 ****File 1:**** src/app.js

This is your **main application setup** (like the foundation of your backend).

import express from "express";

import cors from "cors";

import cookieParser from "cookie-parser";

const app = express();

* **express** → Creates the backend server. It handles HTTP requests and responses.
* **cors** → Middleware to allow requests from other domains (important when frontend and backend run on different ports).
* **cookieParser** → Middleware to parse cookies from the request headers.
* const app = express(); → Initializes your app, similar to creating an object from a class.

### Middlewares (what’s being used before routes run)

app.use(

cors({

origin: process.env.CORS\_ORIGIN || "http://localhost:3000",

credentials: true,

})

);

* Enables **CORS** (Cross-Origin Resource Sharing).
* origin: ... → Defines which frontend URLs can call your backend.
* credentials: true → Allows cookies (like JWT tokens in cookies) to be sent.

app.use(express.json({ limit: "16kb" }));

* Tells Express: “expect JSON in the request body.”
* Limit = **16kb** → prevents huge payloads from crashing your server.

app.use(express.urlencoded({ extended: true, limit: "16kb" }));

* Parses **form data** (like HTML forms application/x-www-form-urlencoded).
* Example: name=John&age=20 → becomes req.body = { name: "John", age: 20 }.

app.use(express.static("public"));

* Serves static files (images, CSS, JS) from public/ folder.
* Example: If you have public/logo.png, it can be accessed at http://localhost:5000/logo.png.

app.use(cookieParser());

* Reads cookies from the client request and makes them available in req.cookies.

### Routes

import userRouter from "./routes/user.routes.js";

app.use("/api/v1/users", userRouter);

* Imports the **User Router** (all user-related routes).
* Mounts them at /api/v1/users.
* Example: If user router has /register, the full path becomes:  
  POST /api/v1/users/register.

✅ **Summary of app.js**:  
This file wires up your entire Express app → adds middleware for security & data parsing → connects routers.

# 📂 ****File 2:**** src/utils/asyncHandler.js

This file solves a very common issue in Express: **handling async errors cleanly.**

### First Approach

const asyncHandler = (requestHandler) => {

return (req, res, next) => {

Promise.resolve(requestHandler(req, res, next)).catch((err) => next(err));

};

};

* asyncHandler is a higher-order function (takes a function and returns a new one).
* requestHandler → your controller function.
* Promise.resolve(...).catch(...) → ensures if the controller throws an error, it is caught and passed to next(err) (Express error handler).

Without this, you’d need **try/catch in every controller**, like:

try {

// do something async

} catch (err) {

next(err);

}

Messy.

With asyncHandler, you just wrap the controller and forget about error handling boilerplate.

### Second Approach (commented out)

const asyncHandler = (fn) => async (req, res, next) => {

try {

await fn(req, res, next);

} catch (error) {

res.status(error.status || 500).json({

success: false,

message: error.message || "Internal Server Error",

});

}

};

* This does **not use next(err)**, instead it **sends the error directly in the response**.
* Advantage: No need for a global error handler.
* Disadvantage: Repeated logic for every controller.

That’s why the **first approach** (with next(err)) is better → central error handling.

✅ **Summary of asyncHandler.js**:  
It’s a helper to make controllers clean and avoid repeating try/catch.

# 📂 ****File 3:**** src/routes/user.routes.js

This defines **all User-related routes** (mini-app).

import { Router } from "express";

import { registerUser } from "../controllers/user.controller.js";

const router = Router();

* Imports Router from Express.
* Creates a router instance (like a mini version of app).
* Imports the registerUser controller.

router.route("/register").post(registerUser);

* router.route("/register") → defines a route for /register.
* .post(registerUser) → when a POST request comes here, call registerUser.
* Because this router is mounted at /api/v1/users (in app.js), the full path is:  
  POST /api/v1/users/register.

export default router;

* Makes this router available for use in app.js.

✅ **Summary of user.routes.js**:  
It organizes all user endpoints. Currently, it has only /register.

# 📂 ****File 4:**** src/controllers/user.controller.js

This file contains **controllers (logic) for User routes.**

import asyncHandler from "../utils/asyncHandler.js";

* Imports asyncHandler to wrap controllers safely.

const registerUser = asyncHandler(async (req, res) => {

res.status(200).json({

message: "ok",

});

});

### Step-by-step:

1. registerUser is wrapped with asyncHandler.  
   → This means if something goes wrong, errors are automatically caught and forwarded.
2. Inside the controller:
   * Right now, it just responds with status **200 OK** and { message: "ok" }.
   * Later, you’ll add logic like:
     + Read data from req.body (like name, email, password).
     + Validate data.
     + Save new user in the DB.
     + Return a success response.

export { registerUser };

* Exports the controller to be used in routes.

✅ **Summary of user.controller.js**:  
Currently, it’s a placeholder. It proves the structure works. Later, you’ll fill it with real business logic.

# 🧩 How Everything Connects

1. app.js → Mounts routers (/api/v1/users).
2. user.routes.js → Defines routes (/register).
3. user.controller.js → Has logic for /register.
4. asyncHandler.js → Ensures async errors don’t crash your app.

So when a client makes:  
👉 POST /api/v1/users/register

The flow is:

* **Express app** (app.js) → sends to userRouter.
* **user.routes.js** → finds /register with method POST → calls registerUser.
* **user.controller.js** → runs the function → sends response { "message": "ok" }.
* **asyncHandler.js** → catches any errors (if there are).

# 🚀 Big Picture

Right now, your backend is **well-structured but minimal**.

* You already have **clean architecture**:
  + app.js → setup
  + routes → endpoints
  + controllers → logic
  + utils → helpers

This is the **professional way** to structure Node.js projects.