Backend

Node.js (V8)

Node JS → JS run time environment → converts JS code into 0's & 1's
that provides JS to run, not just on browser only → V8 engine converts JS code into 0's & 1's

In node terminal if we want to make a file in the folder then
we'll run the command to make the JS file.

eg. touch script.JS → eg. folder → after coming to the folder [redacted] JS.
(touch — In terminal; script.JS — filename)

for running we can run
node script.JS → In terminal.
(script.JS — filename)

Process → object provides info about and controls over, the current node.JS process

Process.argv → returns an array containing the command-line arguments passed when the node.JS process was launched.

Process.cwd() → show the current working directory

Generally we run it over this or node repl, instead of that use JS file

eg. output

[user/ ... → shows path where is node.exe
user/ ... → shows folder path

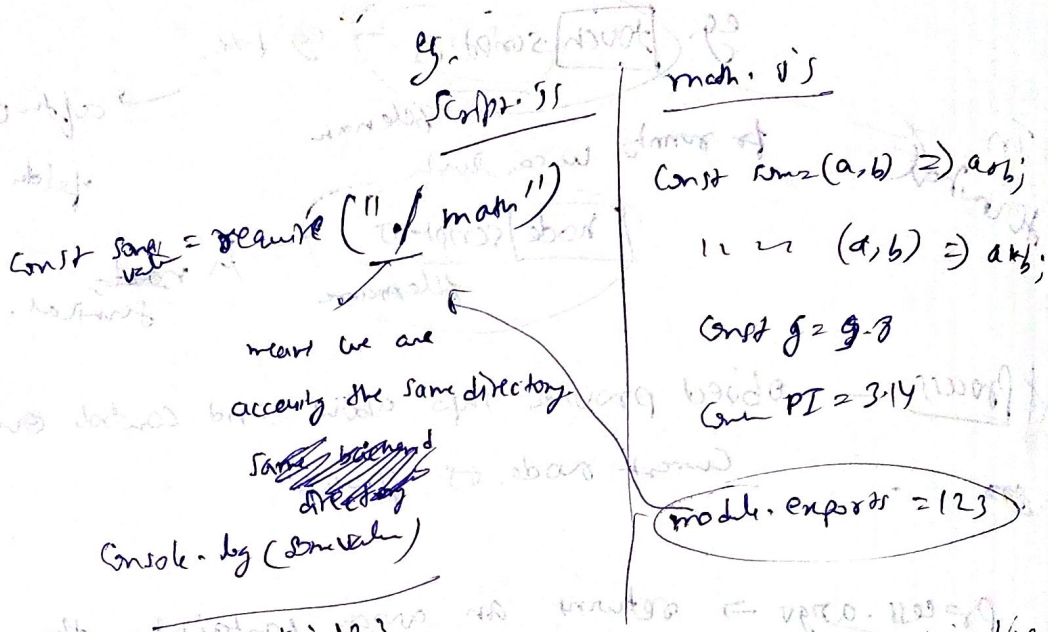
eg. node script.JS hello bye ..
[user/ ... /
/ ... / ... /
hello,
bye,]

module.exports

finds only index.JS file

require() a built-in fn to include external modules that exist in seperate files

module.exports - a special object



Note. by default: if we don't export & require, then it gets empty object

Overview of Backend

2 major component in Backend development

A programming language

es Java, JS, PHP, Py. etc

↓

choose one framework

~~Java~~, we're gonna use

JS → ~~Express~~ using nodeJS, express, ~~next~~

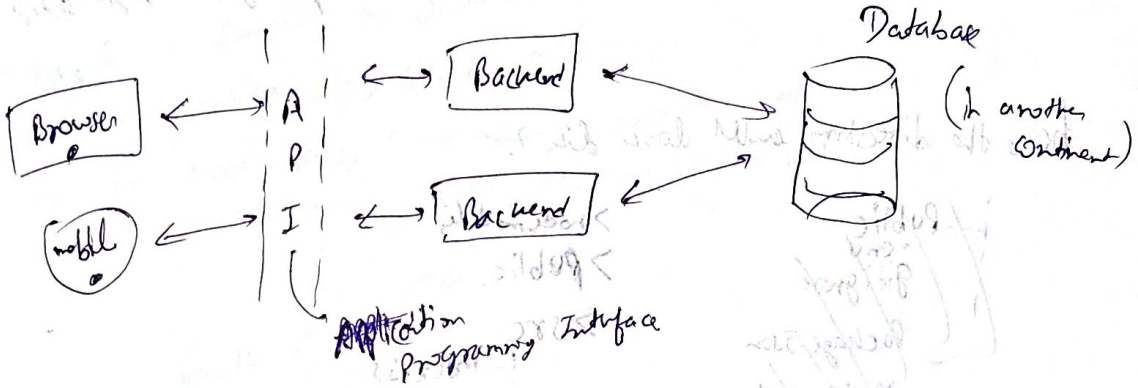
A Database

es Mongodb, MySQL, Postgres, sqlite

↓

ORM, ODM

we're gonna use mongoose for Mongodb



→ a JS based Backend

Data

File

Third parts (API)

→ JS runtime : Node.JS / Deno / bun

↓

we're gonna use this

Important files:

→ Package.JSON

→ .env

→ README, gitignore, etc.

→ src

→ index.js file

↳ entry point

↳ db connects

APP.JS

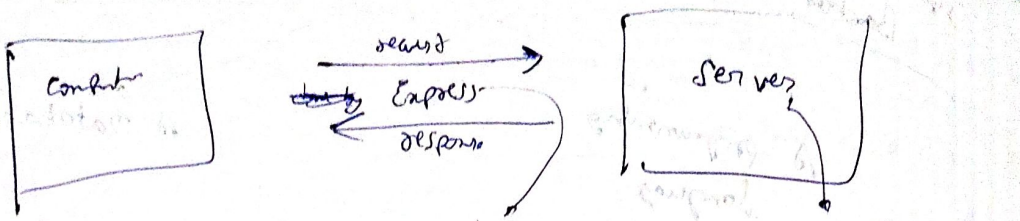
↳ Cookies, configuration

Constants

↳ enum, DB name

directory structure

- DB
- models
- controllers
- Routing
- middlewares
- utils
- more (depends)



it's a package
helps to make servers

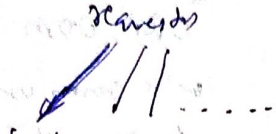
a software, system, special purpose which serves request.

eg.

/ = home route

if login : login setup

: etc.



how the directory will look like ?

- Public
- .env
- .gitignore
- package.json
- readme.md

> node_modules
> public
> src
- index.js
- app.js
- constant.js
- controller
- db
- middlewares
- models
- routes
- utils

- .env
- .env.sample
- .gitignore
- Package.json
- Readme.md

HTTP Server

(hypertext transfer protocol)

- a protocol that's defined for machines to communicate
- specifically for website, it's most common way for our website's frontend to talk with its backend

Node.js → JS runtime environment
- helps to run JS codes.
- can create a game
- can create clis

- In the end, it's the client throwing some information at a server
- server doing something with that info and responding back with the final result.

Think http as a function, where,
- arguments are something the client sends
- rather than calling a fn, client uses a URL
- " " function body, server does something with the request
- " " " ", returning a value, the server responds with some data

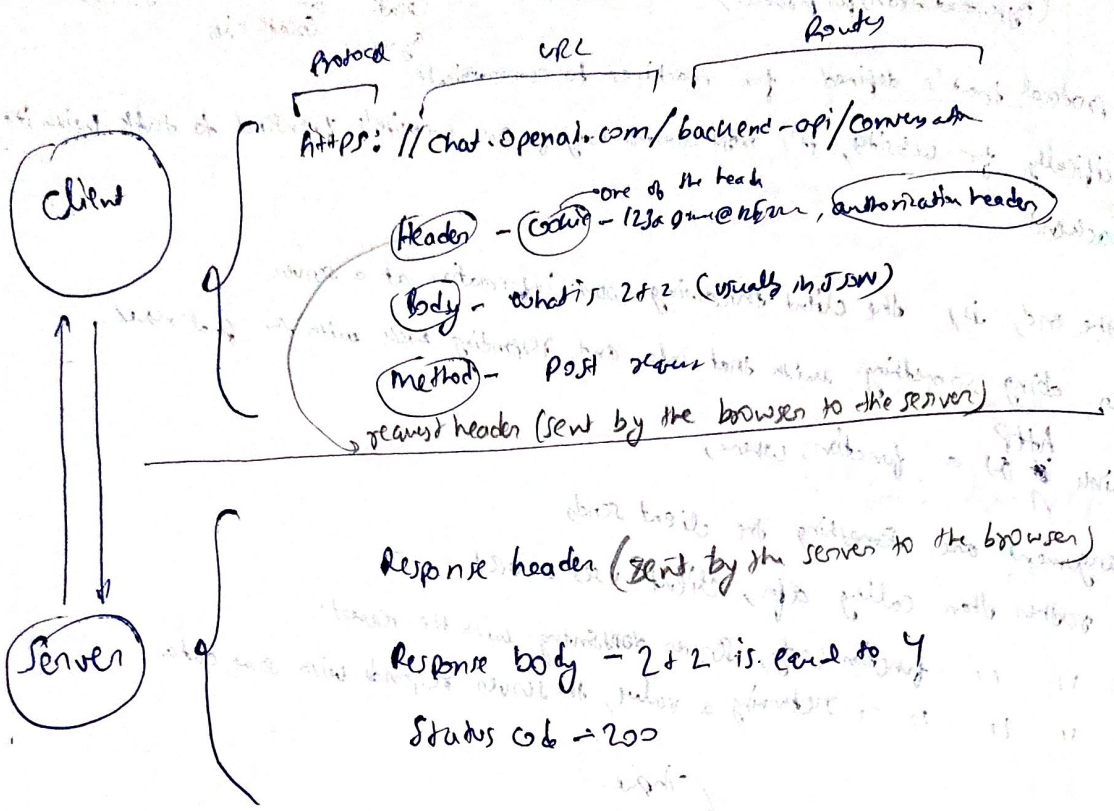
```
function (input) {
  return
}
```

things we should worry abt http

HTTP (Client side)
- Protocol (HTTP/HTTPS)
- Address (URL / IP / Port)
- Route
- Header, body, query parameters
- Method

HTTP (server side)
- Response header
- Response body
- Status codes

Usually communication would happen like this



Things that happen in browser after you fire any request.

1. Browser parses the URL
2. Does a DNS lookup (converts google.com to an IP)
3. Establishes a connection to the IP

DNS resolution

URLs are just like Contacts in your phone

In the end, they map to an ip

Imp point

Headers: → Key-value pairs sent b/w the client (browsers) & the server.
→ They provide the metadata (info abt the request or response, not the actual content)

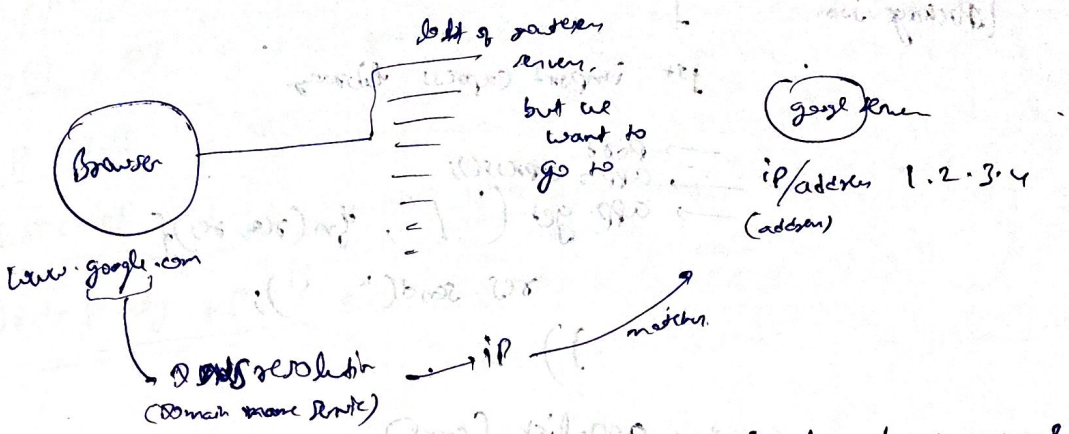
Body: - Unlike metadata in headers, It's the actual data (main content of req or res).
- eg → login details, form, data, JSON (usually in JSON)

Method: → define the type of operation the client wants to perform on the server's resource.
(or verbs)

Cookies: These are stored data within headers, They are delivered & transmitted using headers.
Purpose → Session management, Personalization, Tracking & Analytics.

eg. ip → 237.1.22.5 (like an address to a server)

eg.



we (can't remember) the ip's of the server, so, by domain name it gets mapped to its ip.
(can't remember)

we it's hard to remember the ip of the server

Things that happen on Server after the request is received

- you get the inputs (route, body, headers)
- you do some logic on the input, calculate the output
- you return the output body, headers and status code.

* Common methods we can send to our Backend server →
  1. GET → Retrieves data, eg - get user details.
  2. POST → Send new data, eg - Register a new user.
  3. PUT → Update existing data/replace, eg - Update user profile
  4. DELETE → Delete data, eg - delete user account
  5. PATCH → Update part of data, eg - change user's email only.

* Status Code the backend responds with →
  1. 200 - Everything is OK
  2. 404 - page/route not found
  3. 403 - Authentication issues
  4. 500 - Internal Server Error.

Express Library → lets us create HTTP servers.

Directory:
- index.js
- Package.json

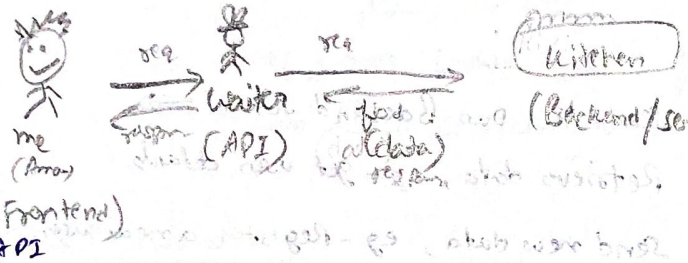
1st import express library

```
const port
app = express();
app.get('/', fn(req, res){
    res.send("  ");
})
app.listen(port);
```

API (Application Programming Interface) :→

→ allows two system (like frontend & backend, or two servers) to talk to each other.

→ Think API as a waiter in the kitchen,



HTTP API or Web API → works over web using HTTP protocol.

So, the frontend (browser) can talk to backend using HTTP requests (GET, POST etc)

⇒ Backend creates APIs so
- Frontend can fetch or send data, Apps connect to our system
- Other services (like Payment gateway, google login) can integrate with the app.
- To let frontend, mobile apps, and other services interact with your backend logic & database.

Types of API: →

- REST / RESTful API ★ (most common in web dev)
- GraphQL API (more flexible querying)
- SOAP API (older, XML-based)
- WebSocket API (for real-time communication, like chat)
- Third-party API (eg. Google Books API, Spotify API)

REST API: (Representational State Transfer): →

⇒ A standard style to design web APIs so they are clean, predictable & easy to use. / Properly designed API uses HTTP methods, clean url, & JSON to make communication smooth b/w frontend & backend

uses:
- uses HTTP methods like GET, POST, PUT, DELETE.
- uses URLs to represent resources (eg. /users/Book/22)
- uses JSON format to send/receive data.

Note: ★ we build APIs on top of HTTP server.
- Then a node.js server using Express.js handles routes (/in/login, /book, etc)
- Each route handles an API endpoint using HTTP methods (GET, POST, PUT)
- Rest & Restful are used interchangeably in web dev.
  - Restful follows REST principles strictly, (the name itself its full (a more proper).

Rest API = RESTful API = web API that follow Rest architecture/principles (uses GET, POST, PUT, Delete + clean URLs + JSON + stateless)

Third Party APIs :

→ An API created and provided by someone else (a third party company) that we can use in our own application to add features without building them from scratch.

→ We use these APIs :-
- → to save time
- → Add powerful features easily
- → Access real time data.

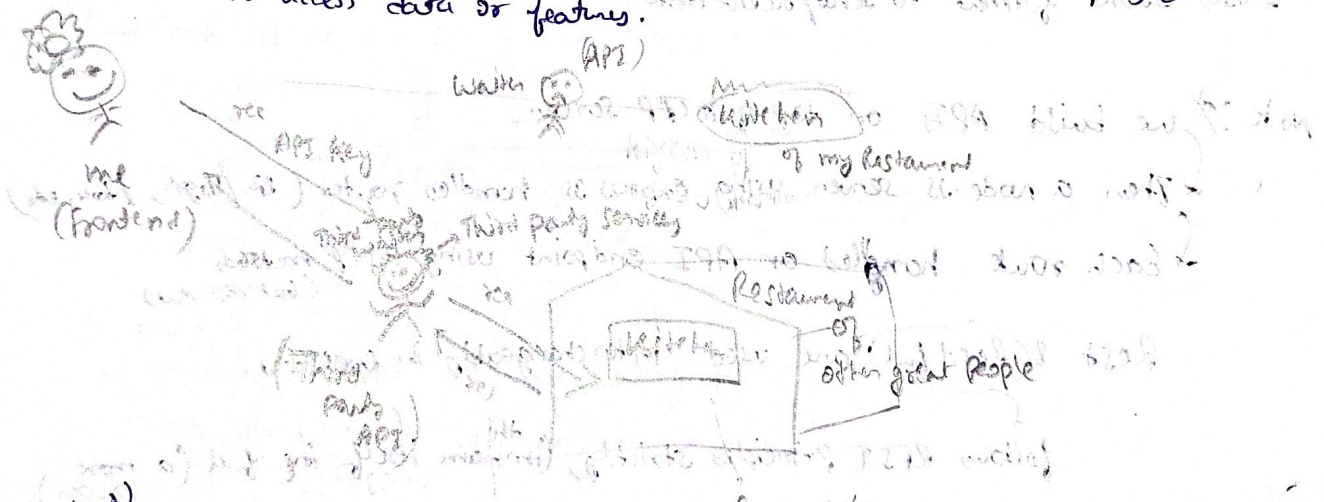
or, we say

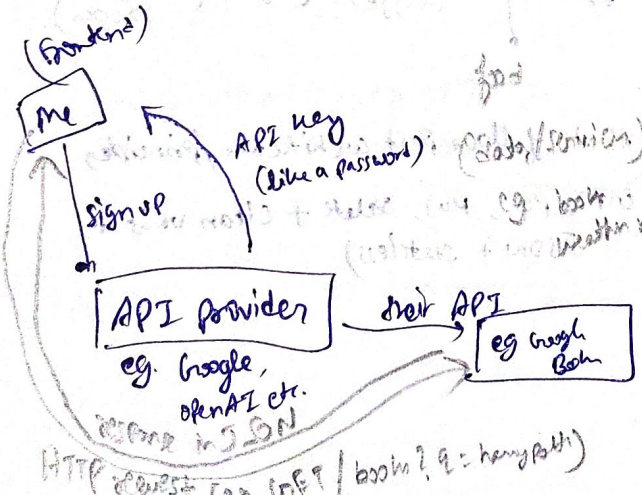
→ eg. Google Book API, Stripe API etc.

→ ~~Most of them APIs are Rest APIs.~~

→ Most of these APIs are Restful. (i.e. they use REST principles, support GET, POST etc. return JSON, access them via HTTP URLs)

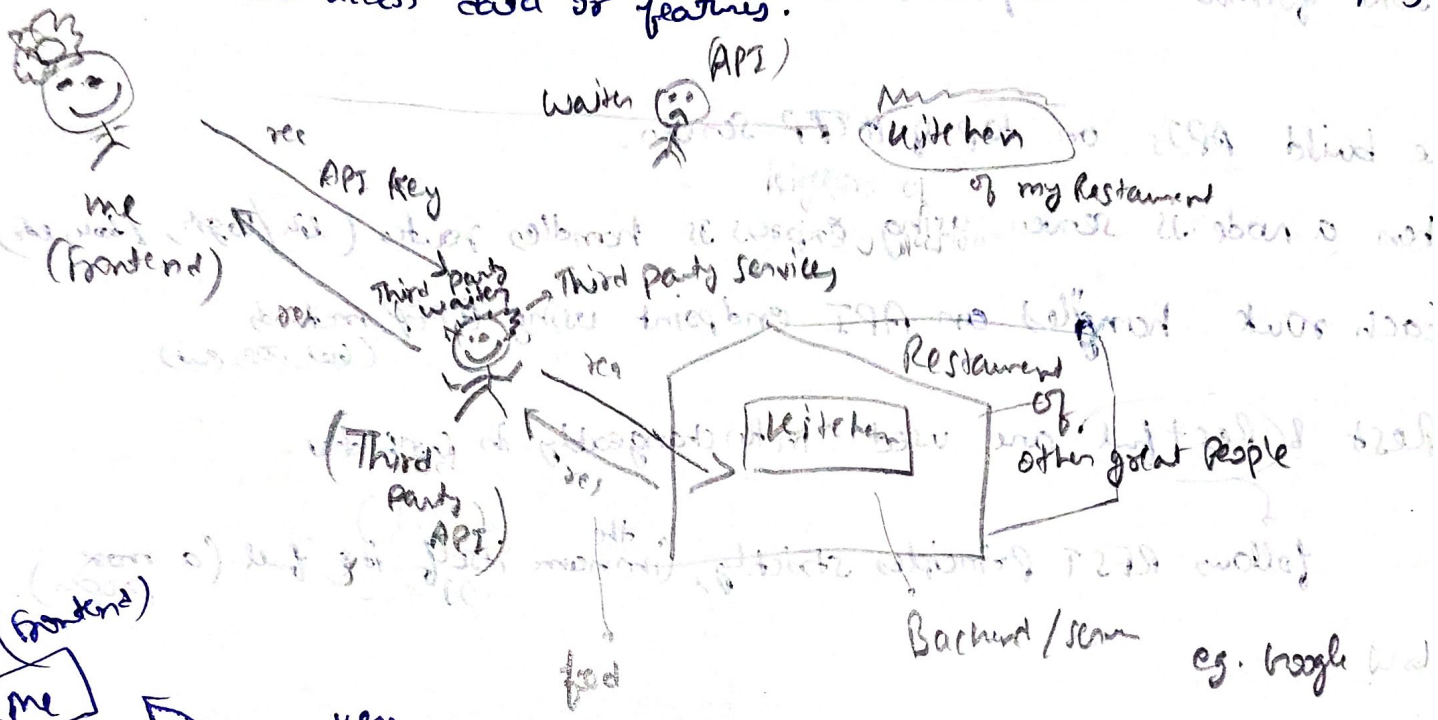
→ It's someone else's REST API that we use in our project to access data or features.

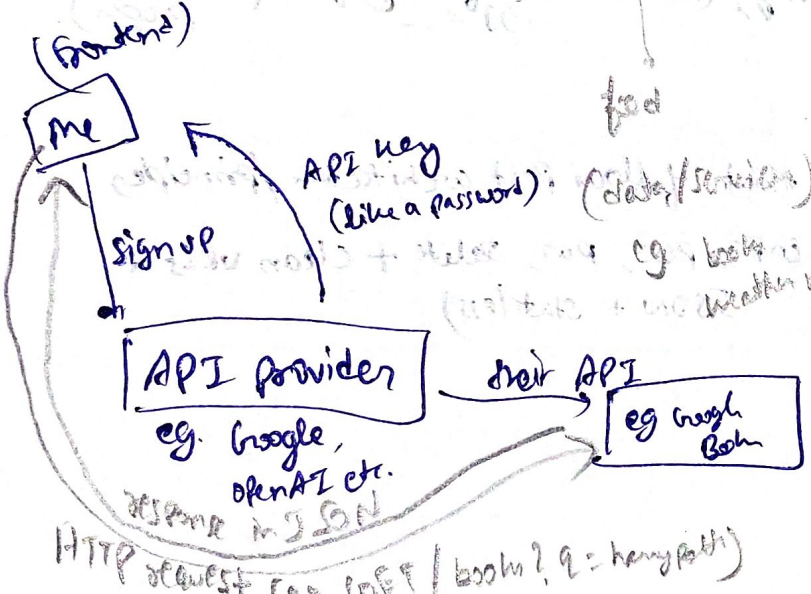




I access them via HTTP

It's someone else's REST API that we use in our proje
to access data or features.





Few Important Third-party APIs:

1. Book & Reading APIs

- Google Books API – (search books, get book details)
- Open Library API – (free book info, cover image, authors)

2. NLP & AI APIs

- OpenAI API (Chat GPT, DALL-E) – (Add AI chatbots, text generation, image generation)
- Cohere API – (Text classification, sentiment analysis)
- Hugging face API – (NLP models, like translation, summarization)

3. Maps and Location APIs

- Google Maps API – (embed maps, find place, calculate distance)
- Mapbox API – (custom maps with detailed location services)
- Geopify / Positionstack – (Convert address to location & vice versa)

4. Weather & Time

- OpenWeatherMap API – (Get current weather, forecasts)
- Weather API – (Weather, astronomy, and historical data)
- World Time API – (Get time info for any location/time zone)

5. User Authentication & Login

- Auth0 – (Secure user login/signup)
- Firebase Authentication – (Easy auth with google, GitHub, email etc.)
- OAuth 2.0 – (Login using Google, GitHub, fb etc.)

6. Payments & E-commerce

- Stripe API – (Accept payments, subscriptions, invoices)
- Razorpay API – (Indian Payment gateway with UPI, cards etc.)
- PayPal API – (Online payment integration)

7. Media & Entertainment

- Spotify API – (Access music tracks, albums, artists)
- Youtube Data API – (Search vids, get vid/channel info)
- TMDb API – (Get movie, show info, rating, posters (like IMDb))

8. Productivity tools

- Twilio API – (Send SMS, WP, voice calls)
- SendGrid API – (Send transactional emails)
- Notion API – (Access pages, databases in your Notion workspace)

9. Tools for Devs:

- GitHub API – (manage repos, Pull req, user data)
- GIPHY API – (Add. fun GIFs to app)
- Unsplash API – (Fetch high-quality images (great for blogs, portfolios).

Note:

related but not same

- Creating an API = Building routes (endpoints) that respond to client requests.
- a Server = Starting a program that listens to req.

Eg.

- Everything with app.get(), app.post() etc. is part of our API
- app.listen() part is the server, which is necessary to make APIs

The Server

```
app.listen(3000, () => {
  console.log("Server is running on port 3000");
});
```

The API

```
app.get('/books', (req, res) => {
  res.send([1, 2, 3]);
});
```

Fetch API :

→ not an api but a method 😊

→ used in frontend to call any API.

eg. `fetch("https://someapi.com/data")` (may be 3rd party API or other)

(It means:

"Hey browser, pls send a GET req to this API and gimme the data" 😊

Since it's asynchronous in nature,
So, It always returns a promise, because network calls are slow and happen in the background.

&

we handle the promise using

→ .then() & .catch()

OR

→ async/await with try/catch } syntactical sugar 😎

eg.

```
fetch("https://fakerapi.it/api/v1/persons").then(
    function(response){
        return response.json();
    }).then(function(finalData){
        console.log(finalData);
    });
```

→ fetch returned a promise, handled by .then, it resolves a response object.
→ Now "response.json()" reads the response body & converts it to json, again it returns a promise resolved by another ".then" (chaining)

Cleaner way:

8 } Using Async Await.

Note: To connect Frontend with Backend we use API.

9

1st Backend : Expose API Endpoints

means It creates HTTP endpoints (URLs) using HTTP methods

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

⇓

2nd Frontend : Make HTTP requests to those endpoints using

- fetch()
or
- Axios
or
- other HTTP clients

3rd CORS (It's a third party middleware) — If needed.

→ If frontend & Backend running on different ports we'll use CORS

4th Sending data

frontend :

eg. fetch('http://. . . /books', {
method = 'post'
headers =
body =
});

Backend.

eg. app.post('/books', (req, res)=>{
Get book. req.body
set status
}

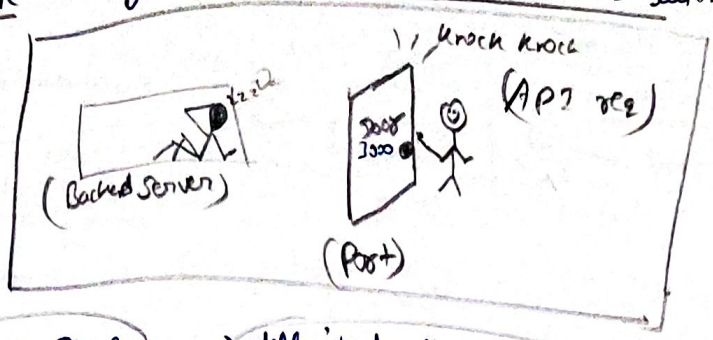
5th Authentication

use JWT

Port → Unlike physical ports (like USB, charger ports), It's a virtual/software concept.

→ A number used by OS to route incoming or outgoing data.

→ It's like a gate through which a server communicates with the outside world.



We create a server → tell it to listen on a port → and handle API requests.

→ ports like 3000, 5000 etc are **local ports** used by an application (like backend server) on our own machine (i.e local host), only we can access it.

→ When we deploy it on internet, the code still contains port = 3000 etc. (cuz we write this), the server still runs on a port (like 3000) but the cloud provider maps it to a public URL.

On Local Machine

eg. http://localhost:3000/api/...

this is private, only our own device can access it.

After Deploying

Now we deploy it on cloud etc.

- the hosting provider gives us a public server to run our code on.
- app still uses app.listen(3000) inside the code.
- hosting provider maps that port (3000) to public URL

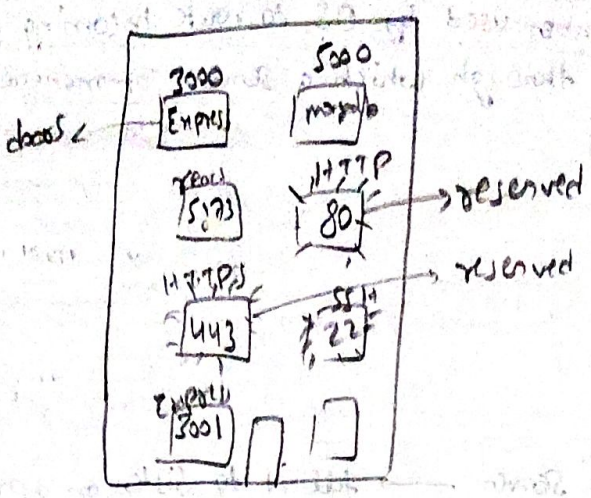
  eg: https://bookify-api.onrender.com

  this is public, anyone can access it.
- the hosting service forwards the requests to port 3000 in the backend server

→ We can run different servers on different ports on the same machine. but only one app can use one port at a time.

Port	Usage
3000	common for Node.js/Express apps
8000	Alternative for servers
5173	vite-based React dev server
27017	Default mongodb port
443	HTTPS (secure web)
80	HTTP (default web)

We say servers listening at (Port)(.....) here server waits for API req.



Building (machine)
eg: IPadd (127.0.0.1)

Note: Can we always use Port 3000, 5000, 3001, Why not 3173, 0000, 2173, 435624 any random no?

Answer: Few reasons are there, Why we usually choose 3000, 5000, 8000 ...

Valid port number: 0 to 65535

0 - 1023
reserved for system services like HTTP (80), HTTPS (443), SSH (22), FTP etc..

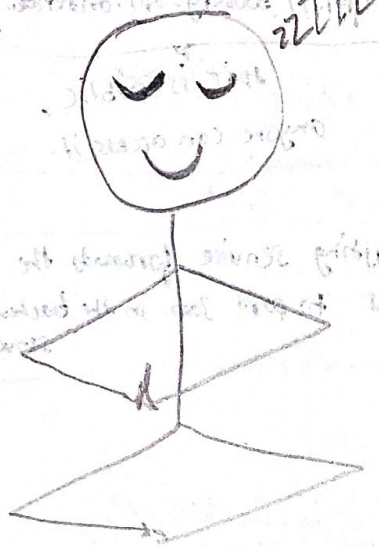
1024 - 65535
Devs choose ports like 3000, 5000 while writing servers by convention (feels good)

Port: 435624 32 X (42 out of range of port no)

Port: 1234 (wrong way) (but confusing to others & possibly clashing with OS process)
- less common

Port: 3000 ✓ (more common)
- by convention perfect

We usually pick 3000, 5000 etc for clarity & convention



Example

eg code: This is how a basic server setup look like, using express in node.js

Load .env variable → Setup Express → Connect to Db → Define home route → Start the server

```
import express from 'express';
```
→ "We're importing express framework."
→ helps to create HTTP servers easily. etc

```
import dotenv from 'dotenv';
```
→ loads dotenv library,

```
import connectDb from './db/index.js';
```
→ importing a custom function named "connectDb" from db/index.js file

```
dotenv.config();
```
→ initializes dotenv, so .env variables are loaded into process.env.

```
const app = express();
```
→ Creating an Express app instance named as "app".

```
const port = process.env.PORT || 3000;
```
→ we set port in .env, it'll use that. Otherwise, fallback to 3000
defining port.

```
connectDb();
```
→ Calling the function to connect to our database before our server starts accepting requests.

```
app.get('/', (req, res) => {
    res.send('<h1>Aman, This is API!</h1>');
});
```
API

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

Starting a server & telling it to listen for request on defined port. If server starts successfully, we'll see the log message "Server is running on port 3000". on our node terminal.

→ Defining a basic route for the homepage '/'. When someone visits our API like http://localhost:3000/, they'll see "Aman, This is API"

Note, req → what user sends (request), res → what server sends back (response)

Query Parameters : ⟶

↳ key-value pairs added to the end of a URL to send optional data to server.

eg. https://localhost:3000/books?genre=fiction&limit=5

QP come after "?" and seperated by "&"

both are query parameters

⇒ Used for :

eg.
* Filtering (?.category = history)
* Sorting (?.sort = price)
* Pagination (?.page = 2 & limit = 10)
* Search (?.q = harry + potter)
* Helpful for middlewares (check for auth i.e ?.token = abc123)

→ We can access them in express as :

eg: req.query.genre → qp
req.query.limit → qp

↳ Example

```
app.get('/search', (req, res) => {
    const q = req.query.q;
    res.send('you searched for: ${q}');
});
```

↳ When we run our localhost

http://localhost:3000/search?q=book

qp

We'll get response.

(you searched for : book)

Middlewares :

→ just a function that runs b/w request and response. They are like checkpoints for handling log, auth etc.

- It can modify req or res objects
- End req-res cycle
- pass control to next middleware or route handler.

Middleware ≡ fn that run before reaching the final route.

route handler → fn that runs when a specific route is hit in express server

fn that handles the request

eg app.get('/', (req, res) => { res.send('Hello Aman'); });

eg HTTP method, route

this arrow function is the route handler.

app.method(path, handler)

It's important bcuz' Every API we create will use Route handler to
→ Recieve data /request
→ Process/validate it
→ Send data back/response back.

Types of Middleware : →

1. Application-level Middleware :

- These apply to the entire app using app.use or app.method();
- eg:

```
app.use((req, res, next) => {
    console.log('This runs for every req');
    next();
});
```

- use case: Logging, request time tracking, request validation

2. Router-Level Middleware :

- Similar to application-level, but bound to an instance of express.Router();
- eg.

```
const router = express.Router();
router.use((req, res, next) => { console.log('Router middlw'); next(); });
router.get('/books', (req, res) => { res.send('Book route'); });
app.use('/api', router); → prefix all routes with /api
```

- Use case; organizing middlewares by route modules (eg booksRouter, usersRouter)

---

* Common use cases of Middleware : Logging, Authentication, validation, Error handling, custom logic.

3. Built-in Middleware

Before we move page, there's the detail explanation

We don't need to install them separately

- Express provides a few built-in middleware:

  - `express.json()` → Parses JSON payloads
  - `express.urlencoded()` → Parses URL-encoded form data
  - `express.static()` → Serves static files.

few confusing terms:

- Parse → Convert from one format to a usable Javascript object
- Payloads → The actual data being carried in the request.
- Static files → files that don't change, eg html, css, JS, Font, Image (PNG, JPG)

→ eg. `app.use(express.json());` → parse JSON req bodies.

4. Third-Party Middlewares

→ installed from npm packages & added via `app.use()`

- eg
  - cors → Enable cross-origin resource sharing (helps server handle request from other domains)
  - morgan → HTTP request logger (logs incoming requests) great for debugging eg. GET /200 15ms
  - helmet → Security headers (protect app)
  - cookie-parser → Parse cookies.

eg.
```
import cors from 'cors';
// morgan : 'morgan';
app.use(cors());
app.use(morgan('dev'));
```

5. Error-handling Middleware

- Catch errors in routes or other middleware
- must have 4 arguments ie (err, req, res, next)
- eg.
```
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).send('Something broke');
});
```
- use case: central error handling, logging errors, sending formatted error responses.

6. Custom middleware

- user-defined middleware functions.

- eg import ..... basic server setup...

classic eg of Authentication middleware

custom middleware that checks for a user in the query and only allow 'Aman' to access the route.

Custom middleware that check for Aman

```
const checkUser = (req, res, next) => {
    const user = req.query.user;
    if (user === 'Aman') {
        console.log("Access granted");
        next();
    } else {
        res.status(403).send('Acces denied');
    }
};
```

next(); → Go to the next route handler.

Home Route (No middleware here)

```
app.get('/', (req, res) => {
    res.send('<h2> ... — API home');
});
```

using the middleware here.

Protected Route using middleware

```
app.get('/secret', checkUser, (req, res) => {
    res.send('<h1> This is secret message only for Aman');
});
```

Starting server.

```
app.listen(3000, () => {
    console.log('... running on port 3000');
});
```

- Use cases: Auth checks, Rate Limiting, Input validation

Built-in middleware

(continued)

Suppose, A client sends this JSON Payload `{"name": "Aman"}` as a raw string → actual data carried in req.

built in middleware → `app.use(express.json());` → It Parses the JSON string into JS object

`{ name: "Aman" }` JS object → Now we can access `req.body.name`

without express.json(), req.body is undefined.

Another example:

```
{
  "email" : "aman@gmail",
  "password" : "1234"
}
```
JSON

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

`{ email: "aman@gmail", pass: "1234" }` JS object

now `req.body.email` → aman@gmail

without express.json, req.body will be undefined

format used by HTML forms

Same as

URL-encoded data

Parses form data

express.urlencoded()

eg.

html form

browser sends data like:

username=aman&Password=1234

Another built-in Middleware.

app.use(express.urlencoded({extended: true})) → Parses this to JS object

Now we can access req.body

eg req.body.username ("aman")

" " .password ("1234")

Input validation (server side)

→ Checking whether the data received from the clients (user/browser)er is valid, correct and safe before processing it.

→ we need it to: prevent bugs – cuz' user might send wrong or missing data

Prevent Security issues: some people can send harmful input (SQL injection, XSS)

Keep the Database clean: only valid data should be saved

Ensures data Integrity: (eg. email is valid, password is strong etc)

Ways of doing it in Backend:

Manual checks:
(use if/else)

eg.

```
if(!req.body.email.includes("@"))
{
  return res.status(400).send("
      Invalid email");
}
```

ugly way

Validation Libraries
(eg. zod, joi, yup etc)
Express-validator → middleware.

zod using:

```
eg import {z} from "zod";
const loginSchema = z.object({
  email: z.string().email(),
  password: z.string().min(6)
});
const result = loginSchema.
    safeParse(req.body);
if (!result.success){
  return
  res.status(400).json
```

We'll use zod

Database-level Validation

- Done using schema constraints at DB level
- Ensures even if Backend validation fails, DB won't accept wrong data.

Zod :

- A TypeScript-first schema validation library.
- not a middleware, but a validation tool/library
- We define the shape of our data (schema) and zod checks if the input matches it.
- Simple & readable
- Can be used both in React (Frontend) & Node/Express (Backend)

eg

```
import { z } from "zod";

const userSchema = z.object({
    name : z.string(),
    age : z.number().min(18),
    email : z.string().email(),
});

const result =
userSchema.safeParse({
    name: "Aman",
    age : 20,
    email: "aman@example.com",
});
```

→ zod will check if it matches the schema or not

```
if (result.success) {
    console.log("data:", result.data);
} else {
    console.log("Error:", result.error.errors);
}
```

} if it matches the schema

} if not

token i.e. JWT

ess, server doesn't store session

Note: When the user logs out, it asks the user to forget it token (or revoke it from the backend

// Primitive value

z.string();
z.number();
z.bigint();
z.boolean();
z.date();
z.symbol();

} primitive values

z.undefined();
z.null();
z.void();

} empty types

z.void(); → accepts undefined

z.any();
z.unknown();

} catch-all type
allows any value

z.never();

} never type
allow no value

Authentication:

↳ Verifying who the user is. "Is this person really who they claim to be?" It answers the question

eg. When we log in with our email & password, the system checks if we're real, valid user.

- It's important because:
  - protects private data
  - Restricts access to authorized users
  - Essential for login systems, user dashboards, admin panel, etc.

It's working →

1. User sends login info (email/password) to server
2. Server verifies credentials
3. If valid, server creates a token. (usually JWT).
4. client stores the token (in localstorage or cookie)
5. For all future requests, client sends token in headers.
6. Server checks the token to authenticate the user.

Authentication : Who are you? (Identity check)

Authorization : What can you do? (Permission check)

Popular Methods for Authentication

Session-based Auth (Traditional)

- Stores session info on server
- Uses cookies

Token-based Auth (modern)

- Uses token like JWT
- Stateless; server doesn't store session

JWT: JSON web token

Note: When the user logs out, It asks the user to forget the token (or revoke it from the backend

Before diving deep into Authentication we should know some topics :
: related to cryptography

1 → Token / (JWT) — JSON web Token

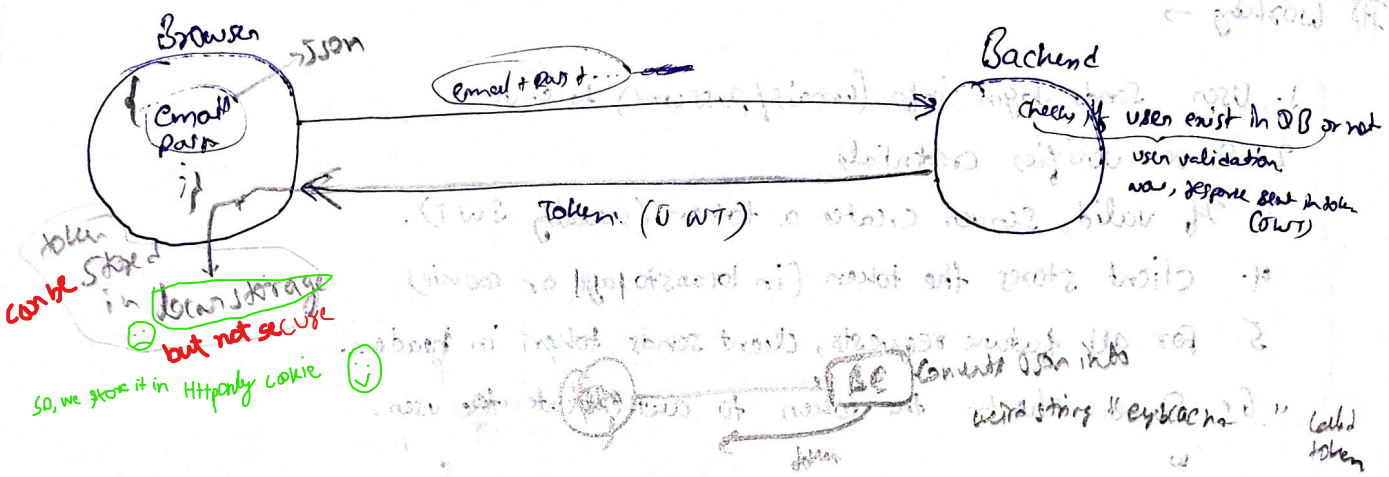
↳ like a digital id card given to us when we log in.
↳ It proves the server : "Hey ! I'm a valid loggin user".

eg eyJhbGciOiJIUzI1NiIsInR...

Contains 3 parts

- Header : info abt the token type and algorithm
- Payload : User data (like user id)
- Signature : to make sure it's not tampered.

→ It takes only JSON as input.



Token is important cuz :

- we don't need to login again & again
- Token make our app stateless & faster
- we can protect routes like /dashboard, /order etc. using token

2 → Hashing :

↳ A process of converting any input (like a string, file or password) into a fixed-length string of characters, usually called a hash.

"Aman" → hash → 9f1a6b....

→ fixed length output → no matter how long input is, hash is of always fixed length.

→ one-way → we can't reverse the hash back to orginal input back

"Aman" → hash 9f1a6b.... ← X

→ Same input = Same hash → Everytime we hash the same data, we get the exact same hash.

→ Tiny change = big difference → Even if we change 1 character in input, the whole hash changes.

Hashing used in :

- Password security
- Data integrity → to check if data is tampered with
- Hash tables (DSA) - Fast data access using keys
- Blockchain - For linking blocks securely.
- File Fingerprinting - To detect duplicates or verify files.

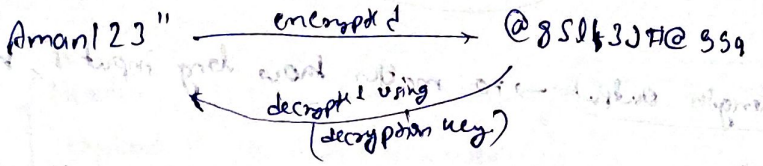
There are lots of hashing algo like MD5, SHA-1, SHA-256, bcrypt

SHA-256 → Secure & widely used

bcrypt → Slow but more secure

3 → Encryption :

↳ Converting readable data (plaintext) into unreadable data (ciphertext) using a key. Only the person with the correct key can decrypt it.



↳ Two-way → we can encrypt & then decrypt back

→ Uses key → secret keys are required to encrypt/decrypt

→ Secure → keeps data safe during transfer or storage

→ used for → messages, passwords (during transit), files, tokens

Hashing	Encryption
One way	two way
Output length fixed	No
used for verification purpose	

4 ⟶ Local Storage

↳ place inside the browser where we can permanently store small amount of data, even if we restart comp, close tab, close browser, it still stays.

eg - JWT (for login) - stored in local storage → but not secure

(can be stored in local storage of browser) but ✗ not secure

Token

Theme

Username

localStorage.setItem("token", "abc123"); ⟶ saving

" .getItem("token"); ⟶ reading

const token = localStorage.getItem("jwt"); ↳ getting token

Local Storage is used in frontend, not in Backend Coz

↳ It's not so secure for sensitive authentication, use HttpOnly Cookie.

We use HttpOnly cookie

↳ When we store JWT (token)

Note

Session storage → like local but temporarily, data clears automatically on closing tab, etc; eg, OTP.

↳ Stored in session storage, not in Local storage. (remember)

↳ Cookies → JS can't access it, It's safe place for tokens, automatically sent in every request, stored in cookie storage of browser.

Note

JWT can be stored in

1) Local storage (not safe)
2) Cookie stored
   - ↳ normal cookie
   - ↳ HttpOnly Cookie — Safe for tokens

Now, we'll see the code & much more abt JWT (we'll use this for Authentication)

Authentication includes few phases :-

Login / Sign → Signup → Login → Access protected Routes → Log Out

(Create account) → Save user in db

(verify credentials and give them a token) → Store in "HttpOnly" cookie

(Use the token to access APIs like (Profile etc) → Only logged-in can access

(Remove the token) → Clean cookie

We'll need :

libraries like
- ~~Postman~~, database
- → EJ modules (node + Express)
- → mongoDB (data base)
- → JWT → bcrypt (for hashing the password)
- → cookie-parser → a third-party middleware → used to read cookies.

As we know
→ Our .env file will have our Port, mongoURI, here also we'll store our JWT_secret

JWT consists of header, Payload, Signature

→ A random string (like password) that our server use to sign and verify tokens.
→ This is our own secret key, that stays same unless we change it
→ Our server uses this key, everytime, signing every token created and verifying every token it receives.
→ Used while signing in :

```
eg. const token = jwt.sign(
    {userId: id, email: ...},   → Payload → from user/db
    → process.env.JWT_secret,   → Secret key, → from our server
    {expiresIn: '7d'}
);                              → options
```

→ Used while verifying token :

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

If someone tampered with the token (changed the payload), the signature (our key) won't match.

→ Now we'll see how it is done in our entry point (index.js)

- first we'll import important libraries.

imported libraries

```
import express from...
  " mongoose...
  " dotenv...
  " cookieParser... from "cookie-parser"
  " bcrypt... from "bcryptjs"
  " jwt from "jsonwebtoken"

dotenv.config();
```

third-party middlewares

```
app.use(express.json());   // to read json body
app.use(cookieParser());   // " " cookies from req.cookies
```

mongodb connection

```
const User = mongoose.model("User", UserSchema);
```

→ this is our model, we generally store this in our model folder, from there we can export this and import it here, but just for eg, I'm using here

Signup Route :

```
// -------------------- 1) SIGNUP ROUTE --------------------
app.post("/signup", async (req, res) => {
  try {
    const { name, email, password } = req.body;

    // Basic checks
    if (!name || !email || !password) {
      return res.status(400).json({ message: "All fields are required" });
    }

    // Check if user already exists
    const existing = await User.findOne({ email });
    if (existing) {
      return res.status(400).json({ message: "Email already in use" });
    }

    // Hash password
    const passwordHash = await bcrypt.hash(password, 10);

    // Create user
    const user = await User.create({ name, email, passwordHash });

    res.status(201).json({
      message: "User created successfully",
      user: { id: user._id, name: user.name, email: user.email },
    });
  } catch (err) {
    console.error("Signup error:", err);
    res.status(500).json({ message: "Server error" });
  }
});
```

Copy code

Login Route

```
// -------------------- 2) LOGIN ROUTE --------------------
app.post("/login", async (req, res) => {
  try {
    const { email, password } = req.body;

    // Check input
    if (!email || !password) {
      return res.status(400).json({ message: "Email and password required" });
    }

    // Find user
    const user = await User.findOne({ email });
    if (!user) {
      return res.status(401).json({ message: "Invalid credentials" });
    }

    // Compare password
    const isMatch = await bcrypt.compare(password, user.passwordHash);
    if (!isMatch) {
      return res.status(401).json({ message: "Invalid credentials" });
    }

    // Create JWT
    const token = jwt.sign(
      { id: user._id, email: user.email },          // payload
      process.env.JWT_SECRET,                       // secret
      { expiresIn: "7d" }                           // expiry
    );

    // Send JWT in HttpOnly cookie
    res.cookie("token", token, {
      httpOnly: true,
      secure: false, // set true in production (HTTPS)
      sameSite: "lax",
      maxAge: 7 * 24 * 60 * 60 * 1000, // 7 days
    });

    res.json({
      message: "Logged in successfully",
      user: { id: user._id, name: user.name, email: user.email },
    });
  } catch (err) {
    console.error("Login error:", err);
    res.status(500).json({ message: "Server error" });
  }
});
```

Copy code

Auth middleware for protects routes

```
// ---- 3) AUTH MIDDLEWARE (for protected routes) ----
const auth = (req, res, next) => {
  try {
    const token = req.cookies.token; // read token from cookie

    if (!token) {
      return res.status(401).json({ message: "Not authenticated" });
    }

    const decoded = jwt.verify(token, process.env.JWT_SECRET); // verify token
    req.user = decoded; // { id, email, iat, exp }
    next();
  } catch (err) {
    console.error("Auth error:", err);
    return res.status(401).json({ message: "Invalid or expired token" });
  }
};

// -------------------- 3) PROTECTED ROUTE --------------------
app.get("/me", auth, async (req, res) => {
  try {
    // req.user.id was set in auth middleware
    const user = await User.findById(req.user.id).select("-passwordHash");
    if (!user) return res.status(404).json({ message: "User not found" });

    res.json({
      message: "Protected route accessed",
      user,
    });
  } catch (err) {
    console.error("Get me error:", err);
    res.status(500).json({ message: "Server error" });
  }
});
```

Copy code

it's a 'get' request

Logout

```
// -------------------- 4) LOGOUT ROUTE --------------------
app.post("/logout", (req, res) => {
  // Clear cookie
  res.clearCookie("token", {
    httpOnly: true,
    secure: false, // true in production
    sameSite: "lax",
  });
  res.json({ message: "Logged out successfully" });
});
```

Now, we'll see the folder structure and arrangement of codes in files->

1 Folder / File Structure -→

```
project/
  package.json
  .env
  server.js
  src/
    app.js
    models/
      User.js
    middleware/
      auth.js
    routes/
      authRoutes.js
```

2 .env

```
PORT=3000
MONGO_URI=mongodb://1[redacted]
JWT_SECRET=supersecretaman
```

3 src/models/User.model.js

```
import mongoose from "mongoose";

const userSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  passwordHash: { type: String, required: true },
});

const User = mongoose.model("User", userSchema);

export default User;
```

4 src/middleware/auth.js (for protected routes)

```
import jwt from "jsonwebtoken";

export const auth = (req, res, next) => {
  try {
    const token = req.cookies?.token;

    if (!token) {
      return res.status(401).json({ message: "Not authenticated" });
    }

    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded; // { id, email, iat, exp }
    next();
  } catch (err) {
    console.error("Auth error:", err);
    return res.status(401).json({ message: "Invalid or expired token" });
  }
};
```

5 src/routes/authRoutes.js

```
import express from "express";
import bcrypt from "bcryptjs";
import jwt from "jsonwebtoken";
import User from "../models/User.js";
import { auth } from "../middleware/auth.js";

const router = express.Router();

// ---- SIGNUP ----
router.post("/signup", async (req, res) => {
  try {
    const { name, email, password } = req.body;

    if (!name || !email || !password) {
      return res.status(400).json({ message: "All fields are required" });
    }

    const existing = await User.findOne({ email });
    if (existing) {
      return res.status(400).json({ message: "Email already in use" });
    }

    const passwordHash = await bcrypt.hash(password, 10);

    const user = await User.create({ name, email, passwordHash });
```

Copy code

```
    res.status(201).json({
      message: "User created successfully",
      user: { id: user._id, name: user.name, email: user.email },
    });
  } catch (err) {
    console.error("Signup error:", err);
    res.status(500).json({ message: "Server error" });
  }
});

// ---- LOGIN ----
router.post("/login", async (req, res) => {
  try {
    const { email, password } = req.body;

    if (!email || !password) {
      return res.status(400).json({ message: "Email and password required" });
    }

    const user = await User.findOne({ email });
    if (!user) {
      return res.status(401).json({ message: "Invalid credentials" });
    }

    const isMatch = await bcrypt.compare(password, user.passwordHash);
    if (!isMatch) {
      return res.status(401).json({ message: "Invalid credentials" });
    }
```

Copy code

```
    const token = jwt.sign(
      { id: user._id, email: user.email },
      process.env.JWT_SECRET,
      { expiresIn: "7d" }
    );

    res.cookie("token", token, {
      httpOnly: true,
      secure: false, // true in production (HTTPS)
      sameSite: "lax",
      maxAge: 7 * 24 * 60 * 60 * 1000,
    });

    res.json({
      message: "Logged in successfully",
      user: { id: user._id, name: user.name, email: user.email },
    });
  } catch (err) {
    console.error("Login error:", err);
    res.status(500).json({ message: "Server error" });
  }
});
```

Copy code

```
// ---- PROTECTED "ME" ROUTE ----
router.get("/me", auth, async (req, res) => {
  try {
    const user = await User.findById(req.user.id).select("-passwordHash");
    if (!user) return res.status(404).json({ message: "User not found" });

    res.json({
      message: "Protected route accessed",
      user,
    });
  } catch (err) {
    console.error("Get me error:", err);
    res.status(500).json({ message: "Server error" });
  }
});

// ---- LOGOUT ----
router.post("/logout", (req, res) => {
  res.clearCookie("token", {
    httpOnly: true,
    secure: false, // true in production
    sameSite: "lax",
  });
  res.json({ message: "Logged out successfully" });
});

export default router;
```

Copy code

6 src/app.js – Create Express app & mount routes

```js
import express from "express";
import cookieParser from "cookie-parser";
import authRoutes from "./routes/authRoutes.js";

const app = express();

// middlewares
app.use(express.json());
app.use(cookieParser());

// routes
app.use("/api/auth", authRoutes);

export default app;
```

7 server.js – Connect DB + start server

```
import dotenv from "dotenv";
import mongoose from "mongoose";
import app from "./src/app.js";

dotenv.config();

const port = process.env.PORT || 3000;

mongoose
  .connect(process.env.MONGO_URI)
  .then(() => {
    console.log("✅ MongoDB connected");
    app.listen(port, () => {
      console.log(`✅ Server running on http://localhost:${port}`);
    });
  })
  .catch((err) => {
    console.error("❌ DB connection error:", err);
  });
```

Now we'll see how our React frontend would call these→

Signup:

js Copy code

```
await fetch("http://localhost:3000/api/auth/signup", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ name, email, password }),
});
```

Login (important: send cookies):

js Copy code

```
await fetch("http://localhost:3000/api/auth/login", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  credentials: "include", // so browser stores cookie
  body: JSON.stringify({ email, password }),
});
```

Get current user (`/me`):

js Copy code

```
const res = await fetch("http://localhost:3000/api/auth/me", {
  method: "GET",
  credentials: "include", // send cookie
});

const data = await res.json();
console.log(data);
```

Logout:

js Copy code

```
await fetch("http://localhost:3000/api/auth/logout", {
  method: "POST",
  credentials: "include",
});
```