Introduction to NodeJS

Chapter-3

What is Node JS?

- Node.js is an open-source, cross-platform JavaScript runtime environment that allows developers to execute JavaScript code outside of a web browser, typically on a server.
- It enables server-side development using JavaScript, a language traditionally used for front-end development. Node.js is built on Chrome's V8 JavaScript engine, which compiles JavaScript into machine code, contributing to its speed and efficiency.

Why Do We Have Node.js If Java, PHP, Python Already Exist?

Before Node.js:

- JavaScript → could run only on the client side (browser)
- Backend could only use: Java, PHP, Python, C#, Ruby, etc.

You needed to learn two languages:

- JavaScript for frontend
- A different language for backend (Java/Python/PHP)

This made development harder.

Node.js solved this problem by allowing JavaScript to run outside the browser, on the server.

Node.js is extremely fast (because of Google V8 engine)

Node.js uses the same engine as Chrome → V8 engine.

It's known for:

- Very high speed
- Non-blocking I/O
- Handles thousands of requests at the same time

Node.js handles real-time applications better than Java/PHP/Python

- Java, PHP, Python → use multi-threaded request handling.
  Node.js → uses event-driven, single-threaded, non-blocking model

This makes Node.js perfect for:

- Chat apps (WhatsApp Web, Messenger)
- Live streaming (YouTube live chat)

1. How Java/PHP/Python Handle Requests (Traditional Multi-Threaded Model)

These languages (Java Servlet, PHP Apache, Python Django/Flask) use a **thread-per-request** model.

How it works:

- Each user request - creates a new thread
- A thread has memory, CPU scheduling overhead
- If many users come - many threads are created
- Thousands of threads = CPU overloaded

Problem:

- Real-time systems require:
- Thousands of connections always open
- Instant data transfer
- Many tiny read/write operations
- But threads are expensive.

In PHP/Java/Python:

If 10,000 users connect → 10,000 threads! - Your server will struggle or crash.

2. How Node.js Handles Requests (Event-driven, Non-blocking Model)

- **How it works:**
- Node.js uses ONE thread to handle ALL users
- It never blocks or waits for tasks
- Long operations run in the background (asynchronously)
- When the task finishes, Node.js processes the callback

Why this works:

- Opening a connection (like chat/streaming) doesn't require much CPU.

It mostly needs:

- Reading small messages
- Sending small responses
- Keeping the connection open

Node.js can hold 50,000+ open connections on a single thread without crashing.

Real-Time Example

WhatsApp Web / Messenger Chat

What happens when user sends a message?

- Node.js keeps a WebSocket connection open
- When user types a message, Node.js receives it instantly
- It doesn’t create a thread
- It forwards message to other users via event callbacks
- Response is sent back instantly
- No thread creation, no blocking, no waiting

If the same chat app was built with Java/PHP:

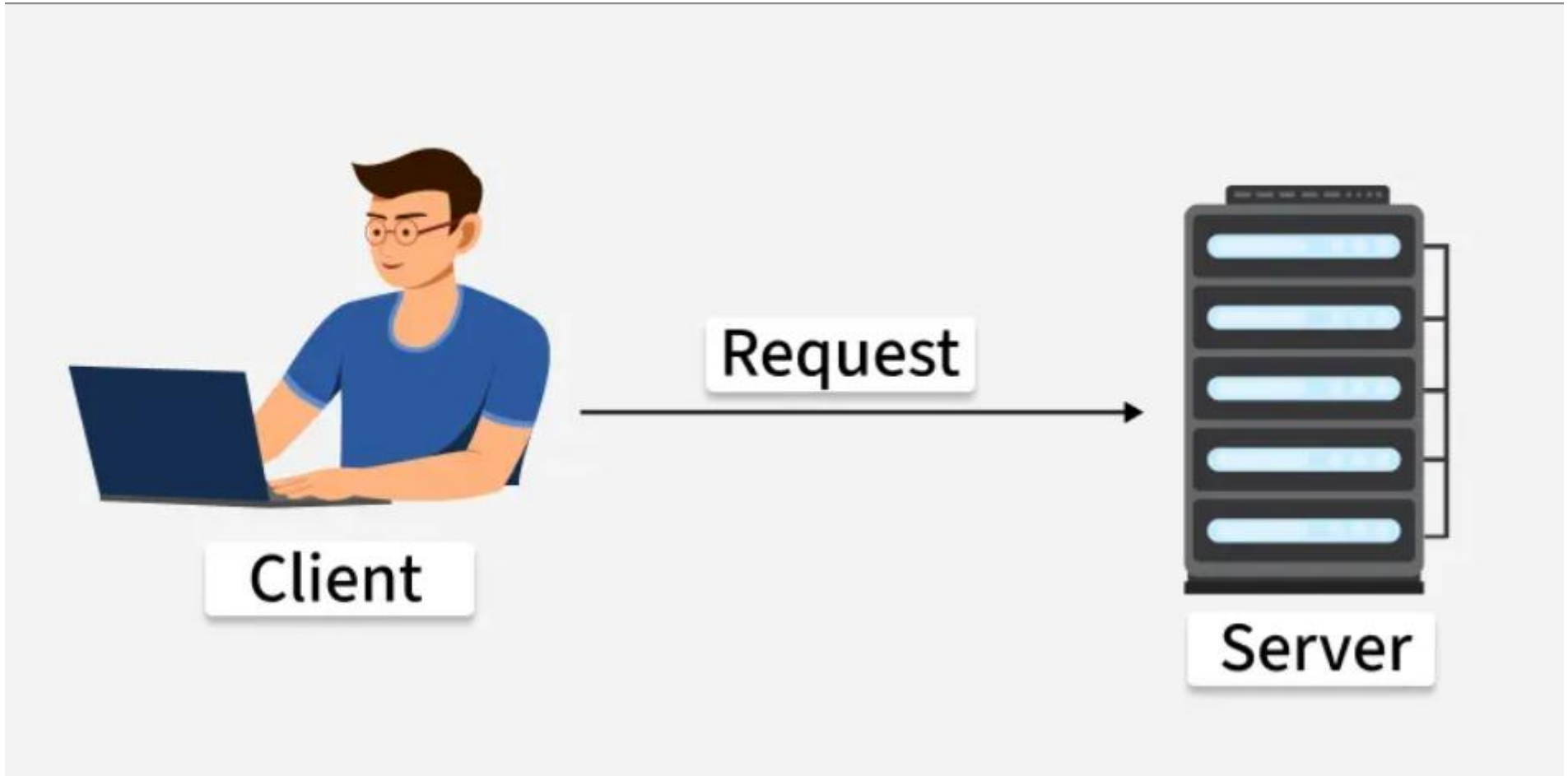
- Every message = new request
- Opens thread, processes message
- Sends response
- Closes connection
- NOT real-time
- Slower
- Heavy on server

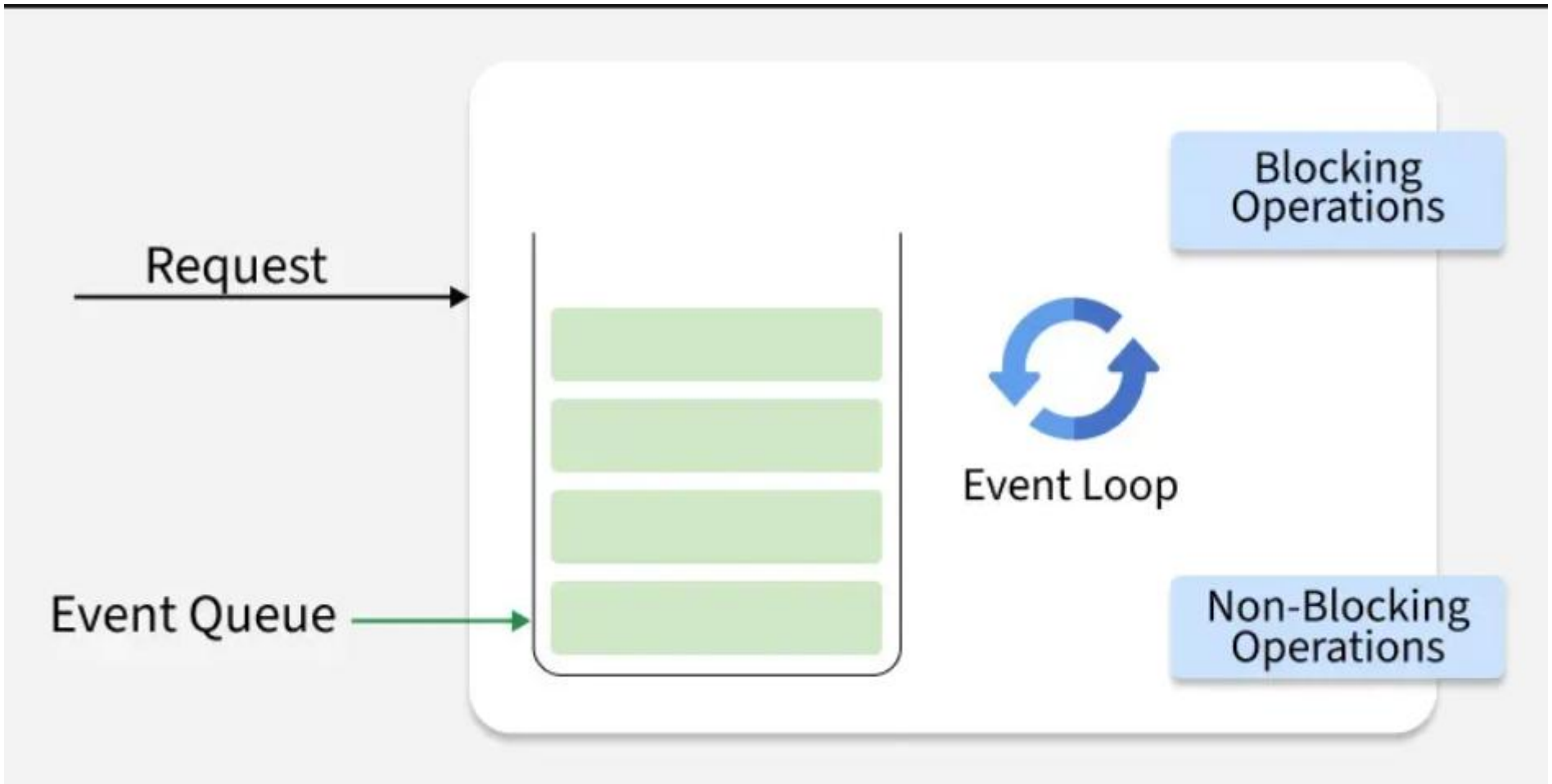
Advantages of NodeJS

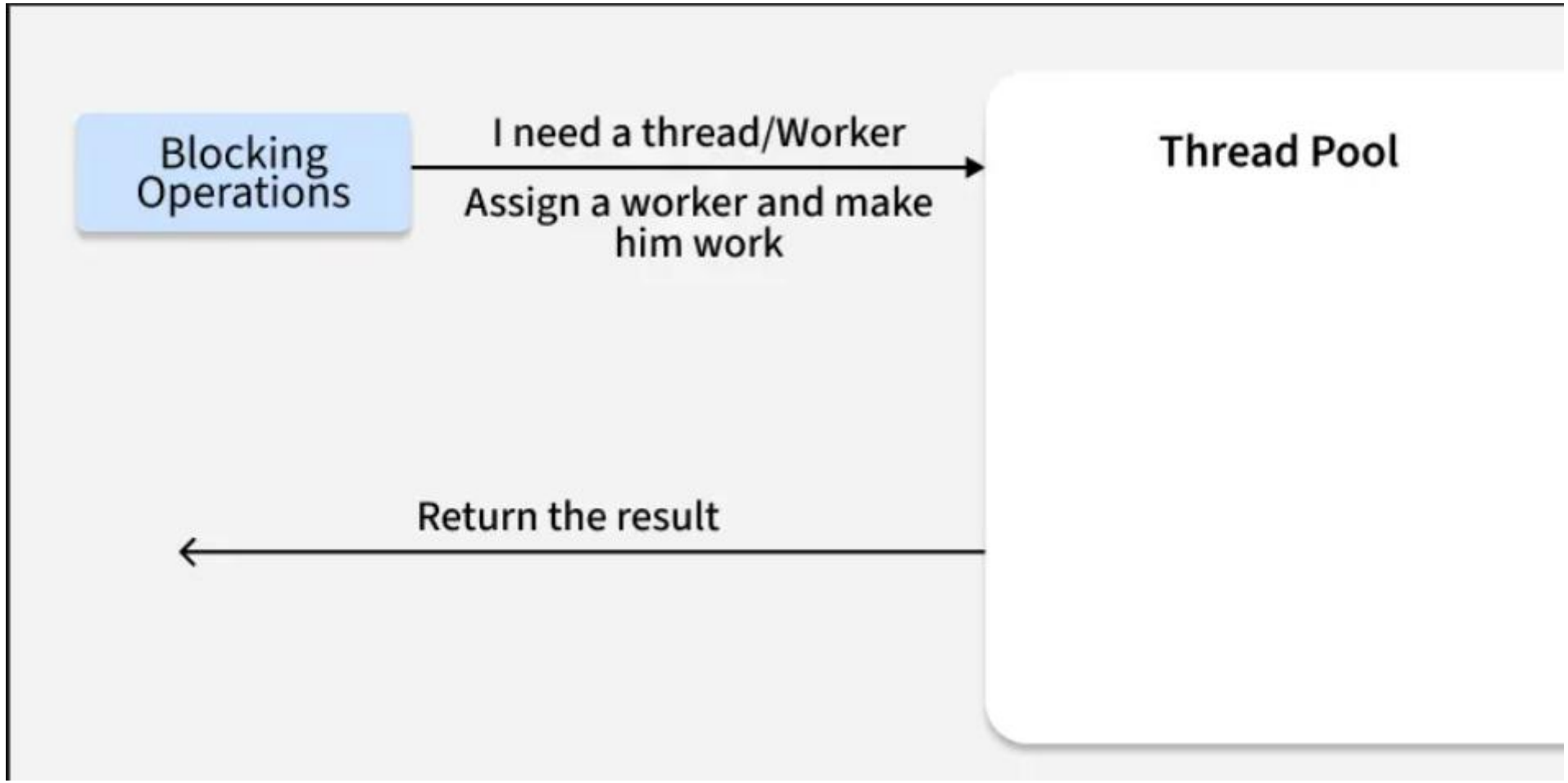
- High performance with V8 engine.
- Handles many clients concurrently.
- JavaScript everywhere- Frontend + backend.
- Large community and npm modules.
- Excellent for real-time apps(chat, gaming, streaming).

How NodeJS works?

- NodeJS is a runtime environment that allows JavaScript to run outside the browser. It is asynchronous, event-driven, and built on the V8 JavaScript engine, making it ideal for scalable network applications.
- NodeJS operates on a single thread but efficiently handles multiple concurrent requests using an event loop.
    - **Client Sends a Request:** The request can be for data retrieval, file access, or database queries.
    - **NodeJS Places the Request in the Event Loop:** If the request is non-blocking (e.g., database fetch), it is sent to a worker thread without blocking execution.
    - **Asynchronous Operations Continue in Background:** While waiting for a response, NodeJS processes other tasks.
    - **Callback Execution:** Once the operation completes, the callback function executes, and the response is sent back to the client.


Request
Client
Server


Request
Event Queue
Event Loop
Blocking Operations
Non-Blocking Operations


Blocking Operations
I need a thread/Worker
Assign a worker and make him work
Thread Pool
Return the result

Node.js basically works on two concept

- Non-blocking I/O
- Asynchronous

Asynchronous

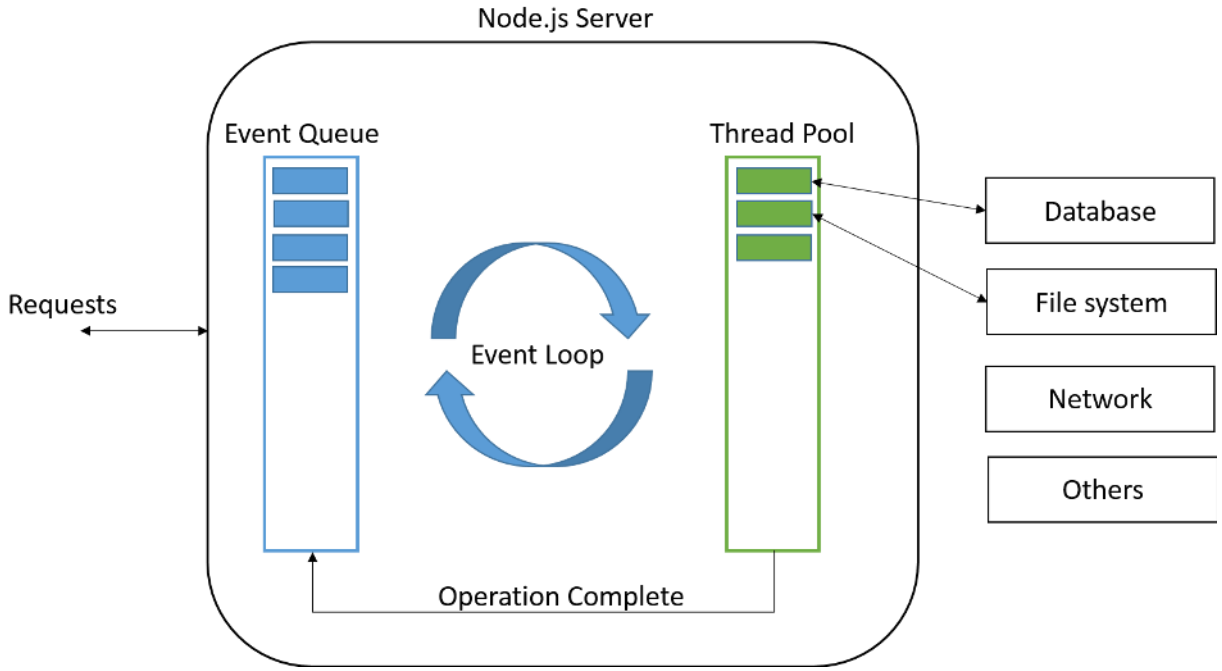
- Asynchronous programming allows Node.js to handle multiple operations without waiting for each task to finish.
- When a request is made (e.g., to a server or database), Node.js sends it off and continues executing other code.
- Once the operation is completed, the callback function (or promise) is executed through the event loop.
- This is possible because Node.js uses an **event-driven, non-blocking I/O model**.
- The single-threaded nature of Node.js doesn't block on I/O tasks; instead, it delegates them to the **libuv** library.
- To implement the concept of the system to handle the request *node.js* uses the concept of ***Libuv***.
- Libuv is an open-source library built-in C. It has a strong focus on asynchronous and I/O, this gives node access to the underlying computer operating system, file system, and networking.
- Libuv implements two extremely important features of node.js
    1. Event loop
    2. Thread loop

Non-blocking I/O

- Non-blocking I/O means Node.js can handle many requests at the same time without waiting for one request to finish before starting the next.
- For example, when Node.js reads a file or fetches data from a database, it doesn’t stop everything else - it continues working on other tasks while waiting for the file or data to be ready.
- I/O stands for Input/Output, which includes things like reading files, sending data to a server, or connecting to a database.

Event Loop and Thread pool

- Node.js receives incoming requests and places them into the Event Queue. The Event Loop continuously checks this queue and decides how to handle each task. If the task is simple, the Event Loop executes it directly. If the task involves heavy I/O work like database queries, file system access, or network calls, it sends that task to the Thread Pool. The Thread Pool performs these operations in the background without blocking the main thread. When the operation is completed, the result is sent back to the Event Loop, which then returns the final response to the user. This process allows Node.js to handle many requests efficiently using a non-blocking, asynchronous model.



How to create a server?

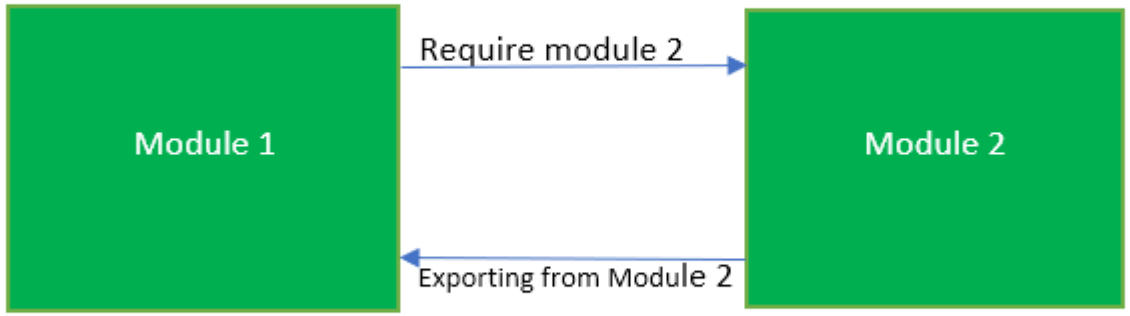
```
1  var http = require("http");
2  var server = http.createServer((req, res) => {
3    res.end("{ name: 'sagar',city:'hello' }");
4  });
5  server.listen(5656, () => {
6    console.log("server started");
7  });
```

Node.Js Modules

What is Module?

In NodeJS, modules play an important role in organizing, structuring, and reusing code efficiently. A module is a self-contained block of code that can be exported and imported into different parts of an application. This modular approach helps developers manage large projects, making them more scalable and maintainable.

1. Variables
2. Functions
3. Classes
4. Objects



Core Built-in Modules

Node.js provides several built-in modules that are compiled into the binary.
Here are some of the most commonly used ones:

fs - File system operations
http - HTTP server and client
os - Operating system utilities
events - Event handling
url - URL parsing

1. NodeJS File System Module

- The Node.js File System module (fs) provides a comprehensive set of methods for working with the file system on your computer.
- It allows you to perform file I/O operations in both synchronous and asynchronous ways.

Importing a module

CommonJS (Default in Node.js)

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

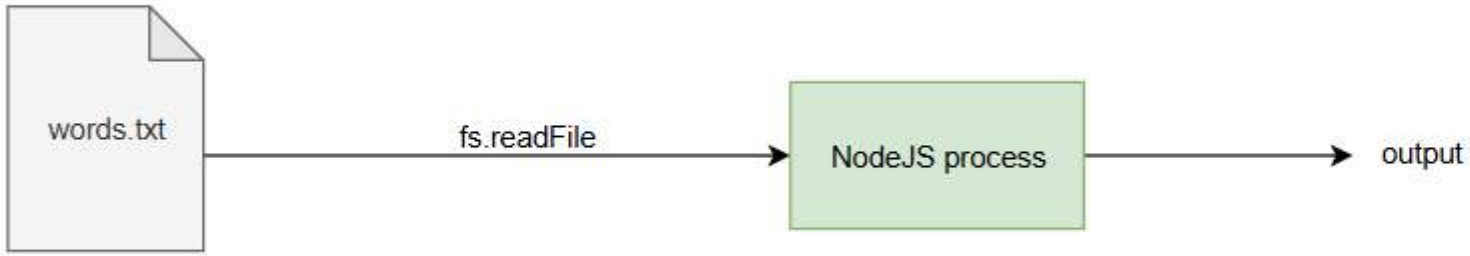
ES Modules (Node.js 14+ with "type": "module" in package.json)

```
import fs from 'fs';
```

Streaming Files

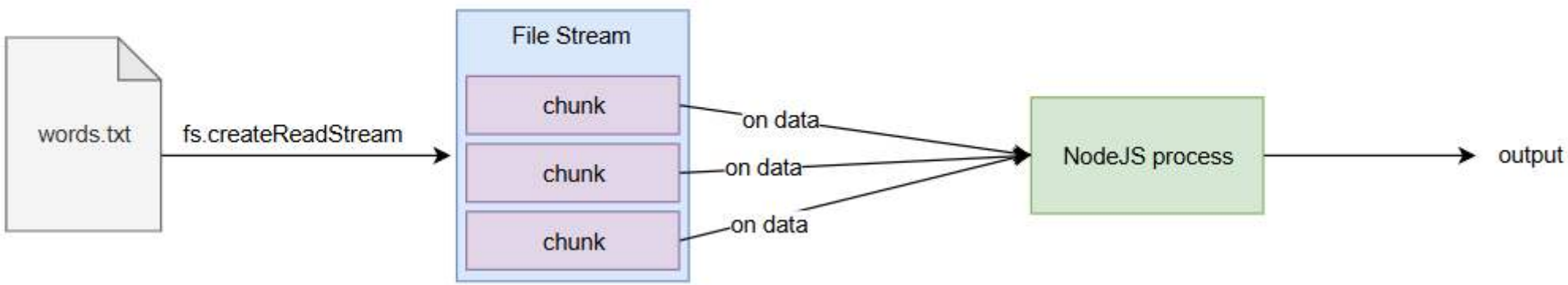
- When we use the readFile and writeFile methods of the fs module, we treated the file as one chunk of data that we can read from or write to.
- While this approach works for small files, it won't scale for larger files. In this case, we need to think of each file as a stream of data, rather than a single large chunk.
- Data streams allow us to work with large data without compromising the limited memory or CPU of our system. The fs module allows us to make use of streams for this purpose.

Non Streaming API



The regular readFile API loads the entire contents onto the Node process

Streaming API



Example for Reading File

Suppose your file README.md contains:

```
Welcome to Node.js
```

Then your code:

```
fs.readFile('README.md', (err, data) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log(data.toString());
});

```

Output:

```
Welcome to Node.js
```

Example for Writing to a File

Suppose your file README.md contains:

Welcome to Node.js

Then your code:

```
const fs = require("fs");
fs.writeFile('README.md',"welcome to NodeJS" ,(err) => {
  if (err) {
    console.error(err);
    return;
  }
  console.log("Wrote to file successfully.");
});

```

Output:

Welcome to Node.js

2. Event Emitters

- If you worked with JavaScript in the browser, you know how much of the interaction of the user is handled through events: mouse clicks, keyboard button presses, reacting to mouse movements, and so on.

- On the backend side, Node.js offers us the option to build a similar system using the events module.

- This module, in particular, offers the EventEmitter class, which we'll use to handle our events.

You initialize that using

```
1  const EventEmitter = require('node:events');
2
3  const eventEmitter = new EventEmitter();
```

- This object exposes, among many others, the on and emit methods.
  - emit is used to trigger an event.
  - on is used to add a callback function that's going to be executed when the event is triggered.

For example, let's create a start event, and as a matter of providing a sample, we react to that by just logging to the console:

```
1  eventEmitter.on('start', () => {
2    console.log('started');
3  });
```

When we run

```
1  eventEmitter.emit('start');
```

the event handler function is triggered, and we get the console log.

Example

```
1  // Import the events module
2  const EventEmitter = require('events');
3
4  // Create an event emitter instance
5  const myEmitter = new EventEmitter();
6
7  // Register an event listener
8  myEmitter.on('greet', () => {
9    console.log('Hello there!');
10 });
11
12 // Emit the event
13 myEmitter.emit('greet'); // Outputs: Hello there!
```

3. HTTP Module

The HTTP interfaces in Node.js, such as the http module, are designed to handle complex web communications efficiently. When large amounts of data, like files or videos, are sent or received over the internet, Node.js doesn't store the entire request or response in memory at once. Instead, it processes the data in small chunks as it arrives — a technique known as streaming. This approach allows Node.js to manage large or continuous data transfers smoothly without consuming too much memory, making it fast and efficient for handling web requests.

HTTP message

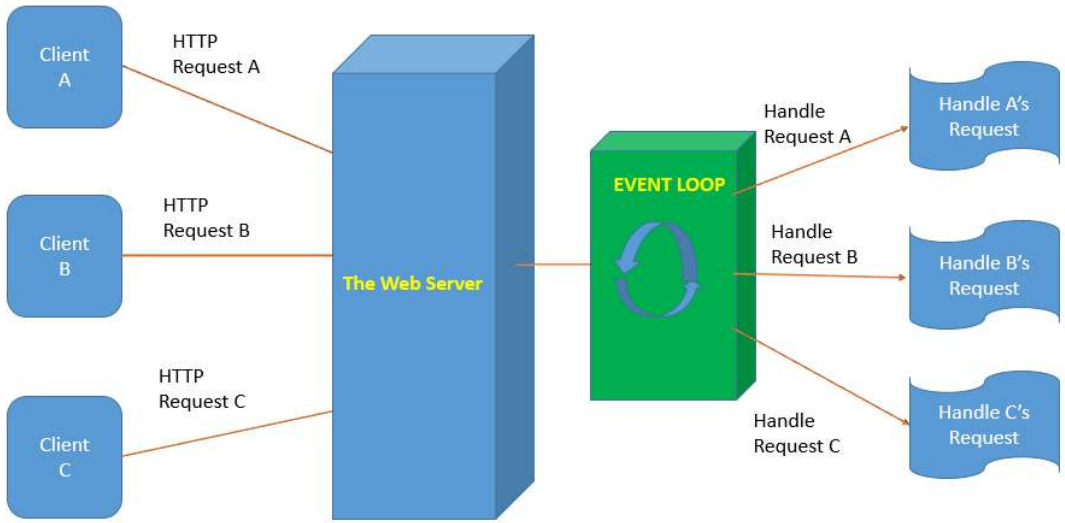
```
{ "content-length": "123",
  "content-type": "text/plain",
  "connection": "keep-alive",
  "host": "example.com",
  "accept": "*/*" }
```

Keys are lowercased. Values are not modified.

The HTTP API in Node.js is designed to be low-level, meaning it gives developers more control over how HTTP requests and responses are handled.

It mainly focuses on stream handling (sending and receiving data in chunks) and message parsing (separating the request into headers and body). However, it doesn't go further to interpret or modify what's inside those headers or the body — that part is left for the developer to handle manually.

The property message.headers shows the headers in a simple object form, while message.rawHeaders keeps the headers exactly as they were received, in an array format like [key, value, key2, value2, ...].



Basic http server example

```
// Import the HTTP module
const http = require('http');

// Create a server object
const server = http.createServer((req, res) => {
  // Set the response HTTP header with HTTP status and Content type
  res.writeHead(200, { 'Content-Type': 'text/plain' });

  // Send the response body as 'Hello, World!'
  res.end('Hello, World!\n');
});

// Define the port to listen on const PORT = 3000;

// Start the server and listen on the specified port
server.listen(PORT, 'localhost', () => {
  console.log(`Server running at http://localhost:${PORT}/`);
});
```

Understanding the Code

http.createServer() - Creates a new HTTP server instance
The callback function is executed for each request with two parameters:
req - The request object (http.IncomingMessage)
res - The response object (http.ServerResponse)
res.writeHead() - Sets the response status code and headers
res.end() - Sends the response and ends the connection
server.listen() - Starts the server on the specified port

Key Features

- Create HTTP servers to handle requests and send responses
- Make HTTP requests to other servers
- Handle different HTTP methods (GET, POST, PUT, DELETE, etc.)
- Work with request and response headers
- Handle streaming data for large payloads

4. OS Module

- The os module in Node.js provides information about the computer's operating system.
- It allows developers to access system-level details such as CPU, memory, hostname, and platform.
- It's a built-in (core) module — no need to install it separately.

Importing the module:

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

Example Code

```
1  const os = require('os');
2
3  console.log("Operating System:", os.type());
4  console.log("Platform:", os.platform());
5  console.log("Architecture:", os.arch());
6  console.log("Hostname:", os.hostname());
7  console.log("Uptime (seconds):", os.uptime());
8  console.log("Total Memory:", os.totalmem());
9  console.log("Free Memory:", os.freemem());
```

Output

```
Operating System: Windows_NT
Platform: win32
Architecture: x64
Hostname: LAPTOP-123
Uptime (seconds): 10540
Total Memory: 17071063040
Free Memory: 8961230848
```

Key Features

- Operating System type and version
- CPU and memory usage
- System uptime
- User information
- Network interfaces

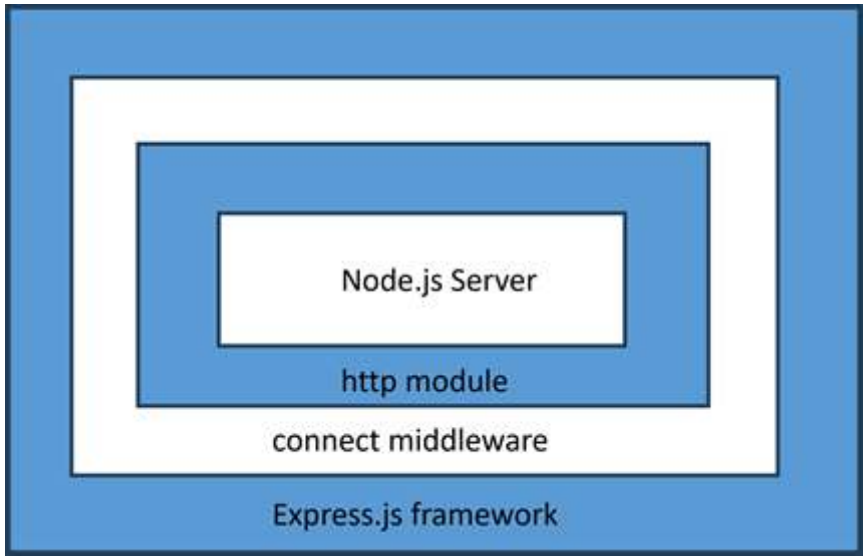
Useful for monitoring, logging, and debugging applications.

Express.js

Building web servers & APIs with Node.js

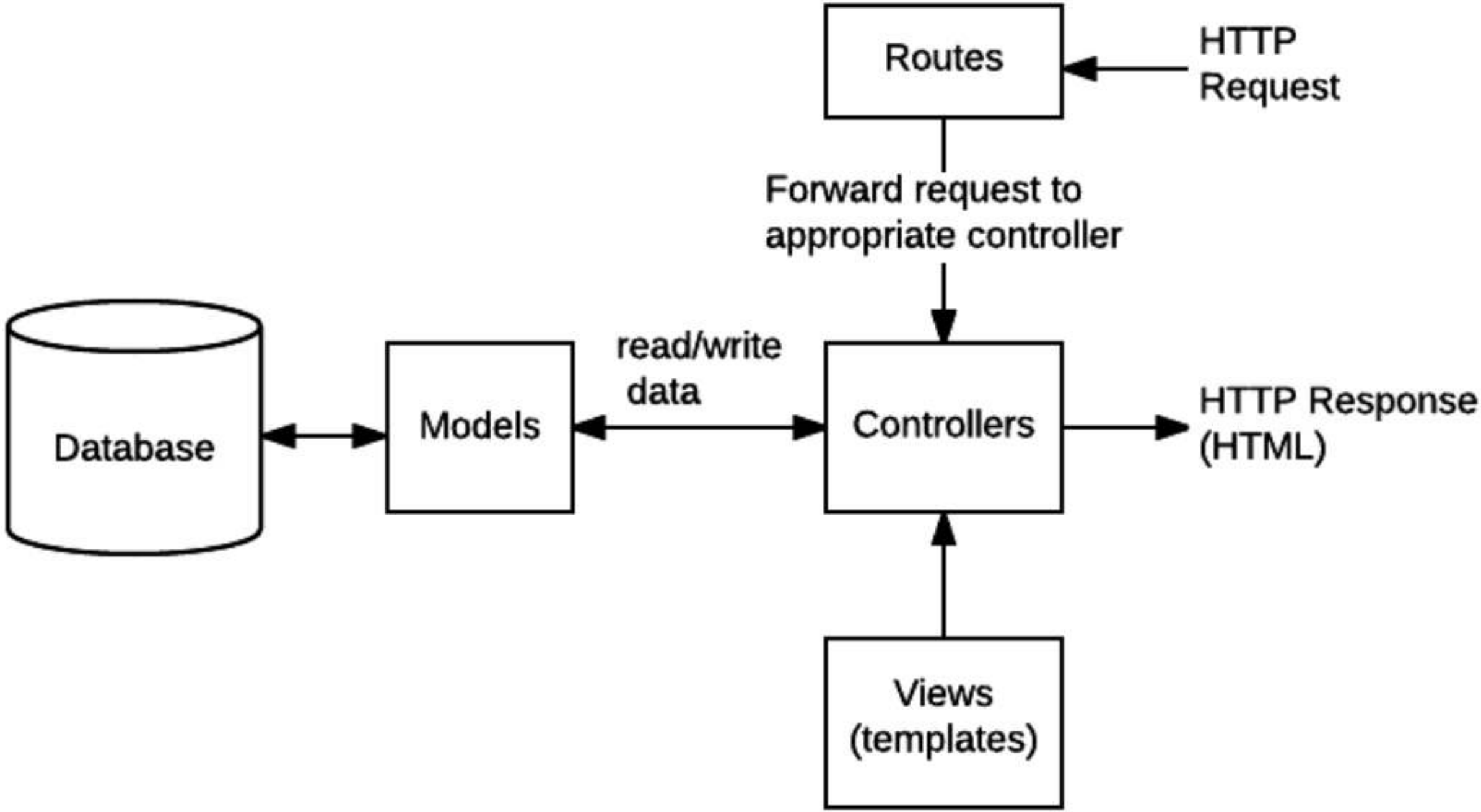
What is Express.js?

- Express.js (or simply Express) is the most popular Node.js web application framework, designed for building web applications and APIs.
- Express is a fast, unopinionated, minimalist web framework for Node.js
- It provides a thin layer over Node's HTTP module, to help you define routes, middleware, views, etc.
- It's often called the de-facto standard server framework for Node.js



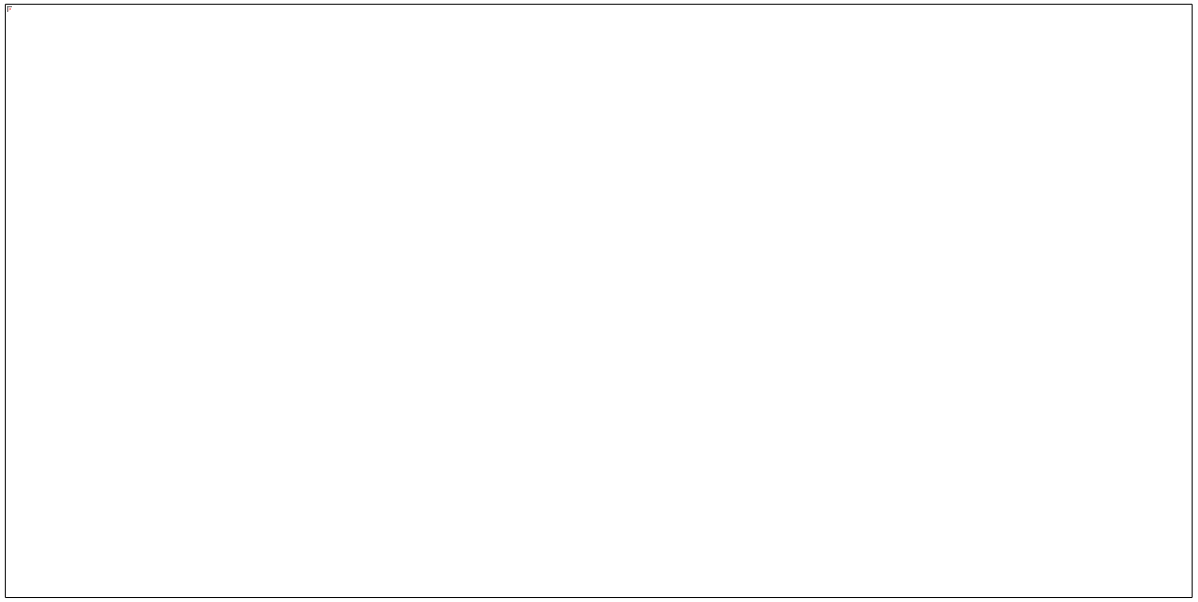
Key Characteristics:

- Minimal and flexible
- Unopinionated (you decide how to structure your app)
- Lightweight and fast
- Extensible through middleware
- Huge ecosystem of plugins and extensions


Routes
HTTP
Request
Forward request to
appropriate controller
Database
Models
read/write
data
Controllers
HTTP Response
(HTML)
Views
(templates)

Why use Express.js?

- Simplifies HTTP server handling: routing, requests, responses.
- Huge ecosystem of middleware packages (for things like cookies, sessions, body parsing) thanks to being lightweight and flexible.
- Works seamlessly with front-end frameworks & APIs (e.g., SPAs, mobile backends)
- JavaScript all the way: both server-side and client-side can use it (for full-stack developers)



What happens under the hood?

- A request comes in to the server (via HTTP).
- Express uses routing and middleware to handle it.
- app.get() defines route handlers.
- res.send() sends the response.
- app.listen() starts listening for connections.


Req Obj
Res Obj
Request
// Middleware
.....
next();
// Middleware
.....
next();
// Middleware
.....
next();
// Middleware
.....
res.send()
Response

When to Use Express (and When Not)

Use Express when:

- You need to build web API servers, REST APIs.
- You want fast iteration, JS full-stack.
- You need middleware support and flexibility.

Limitations / when to evaluate alternatives:

- For heavy structured applications, you'll need disciplined architecture (because Express itself is unopinionated).
- If you need serverless / microservice frameworks tailored for specific scale, you might evaluate frameworks built on Express or other ecosystems.

Basic Express Application

Prerequisites & Setup

You should have:

- Node.js installed (v14.0.0 or later recommended)
- npm (comes with Node.js) or
- yarnA code editor (VS Code, WebStorm, etc.)

Steps:

1. To use Express in your Node.js application, you first need to install it:

```
npm install express
```

2. To install Express and save it in your package.json dependencies:

```
npm install express --save
```

Hello World Example

```
1  var express = require('express');
2  var app = express();
3
4  app.get('/', function (req, res) {
5     res.send('Hello World');
6  })
7
8  var server = app.listen(5000, function () {
9     console.log("Express App running at http://127.0.0.1:5000/");
10 })
```

Save the above code as index.js and run it from the command-line.

```
D:\expressApp> node index.js
Express App running at http://127.0.0.1:5000/
```

Output

CODETANTRA

Express.js Deep Dive:

Middleware, Security & Error Handling

Overview

Middleware & Routing

Understanding the request-response pipeline, custom middleware creation, and advanced routing patterns

Authentication & Security

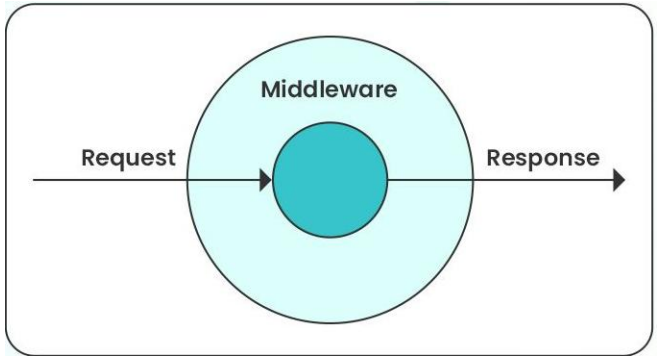
Implementing robust authentication with Passport.js, session management, and security best practices

Error Handling & Logging

Building resilient applications through proper error management and comprehensive logging strategies

What is Middleware?

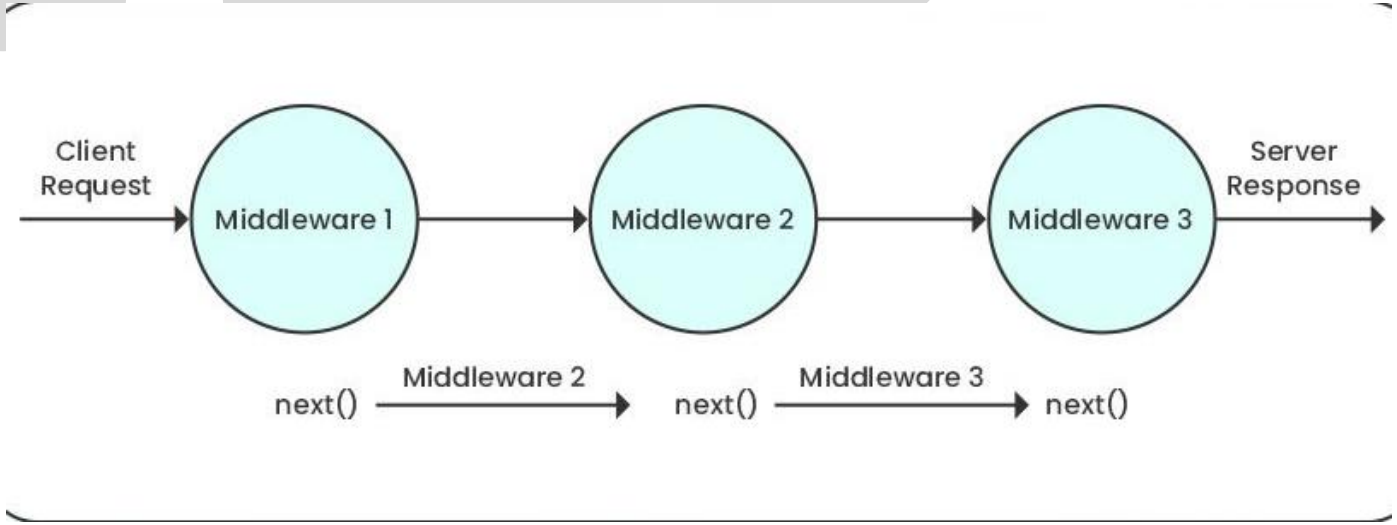
- Middleware is a request handler that has access to the application's request-response cycle. It is a function which contains request object (req), response object (res), and the next middleware function.
- It allow you to run code, modify requests and responses, end the request-response cycle, or pass control to the next function in the stack.
- It was essential for adding features like logging, authentication, data parsing, and error handling to your Express applications.
- Middleware functions are executed sequentially in the order they are defined.

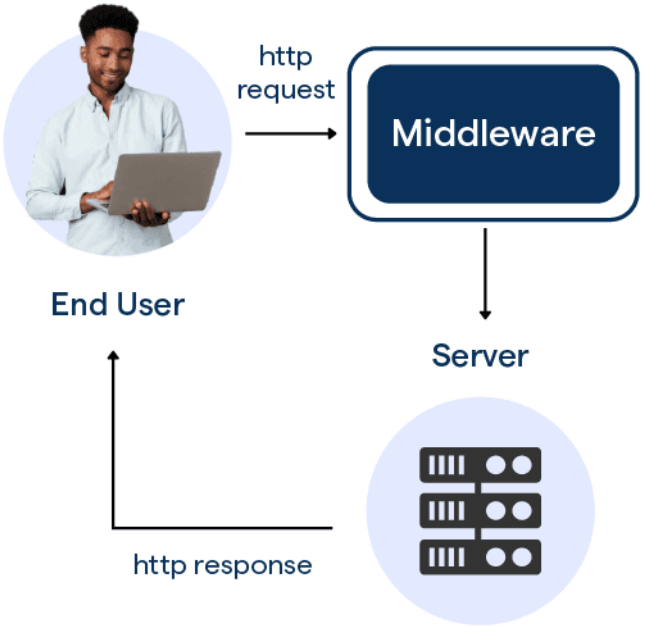


Middleware works like airport security checkpoints. Your request (passenger) passes through multiple stations—ticket validation, security screening, boarding pass check—before reaching the destination (route handler).

CODETANTRA

Types of Middleware





- **Application-level Middleware**
- **Router-level Middleware**
- **Error-handling Middleware**
- **Built-in Middleware**

Application-level Middleware

- This middleware is bound to an entire Express application and runs for every request to the app.
- used for tasks like logging, parsing JSON bodies, authentication, and setting headers globally across routes.
- To Defined the middleware we using
  - **app.use();**
  - **app.METHOD(); // Where METHOD is Http Method**

```
1  const app = express();
2
3  app.use((req, res, next) => {
4      console.log("Time : ", Date.now());
5      next();
6  })
```

This Function will be executed every time the application receives the request

req (Request Object):

- Represents the HTTP request from the client. Contains information such as URL, HTTP method, headers, query parameters, and the body of the request.
- Used to access data sent by the client.
- Example properties: **req.url**, **req.method**, **req.headers**, **req.body**

res (Response Object):

- Represents the HTTP response that the server sends back to the client. Provides methods to send data, set headers, status codes, and control the response lifecycle.
- Common methods: **res.send()**, **res.json()**, **res.status()**, **res.redirect()**.

next (Next Middleware Function):

- Helps to passes control to the next middleware in line.
- Essential for chaining multiple middleware functions.
- Without calling **next()**, the request-response cycle ends, potentially causing the client to hang.

Router-level Middleware

- This middleware is bound to an instance of **express.Router();** and applies only to routes defined in that router.
- useful for modularizing middleware logic for specific route groups (e.g., user management routes).
- To Defined the middleware we using
  - **router.use();**
  - **router.METHOD(); // Where METHOD is Http Method**
  - and mounted on the main app with **app.use();**

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

// Router-level middleware executed for all routes in this router
router.use((req, res, next) => {
  console.log('Router middleware - Time:', Date.now());
  next();
});
```

```
import {router} from './user'

// Mount router on a path in the main app
app.use('/user', router);
```

Error-handling Middleware

- Specialized middleware designed to handle errors occurring in the application.
- Takes four arguments - (err, req, res, next). This identifies it as an error-handling middleware.
- Catches errors from previous middleware or route handlers, centralizing error management and preventing server crashes.
- If an error occurs, it is passed to this middleware via next(err). And send custom error responses

```
1  app.use((err, req, res, next) => {
2      console.error(err.stack); // Log the error stack trace
3      res.status(500).send('Something went wrong!');
4  });
```

Place error-handling middleware after all other middleware and route handlers.

Built-in Middleware

- Middleware functions provided by Express itself that handle common web application tasks

Middleware Function	Purpose
express.json()	Parses incoming requests with JSON payloads and populates **req.body** with the data.
express.urlencoded()	Parses **application/x-www-form-urlencoded** data submitted by HTML forms into **req.body.**
express.static()	Serves static files such as images, CSS, and JavaScript from a specified directory.
express.text()	Parses incoming requests with text/plain payloads.
express.raw()	Parses incoming requests with Buffer payloads, typically used for binary data.

Example :

```
1   import express from 'express';
2   const app = express();
3
4   // Parse incoming JSON data
5   app.use(express.json());
6
7   // Parse URL-encoded form data
8   app.use(express.urlencoded({ extended: true }));
9
10  // Serve static files from the "public" directory
11  app.use(express.static('public'));
```

Third-party Middleware

- Middleware developed and maintained by the community to add various functionalities to Express applications.

Middleware	Purpose
cors	Enables Cross-Origin Resource Sharing (CORS), controlling resource access from different domains.
cookie-parser	Parses cookie header and populates **req.cookies** for easy cookie management.
helmet	Secures apps by setting HTTP headers to protect against common vulnerabilities.
passport	Provides authentication strategies like **OAuth** and **OpenID** for user login management.
morgan	Logs HTTP requests for debugging and monitoring server activity.
compression	Compresses HTTP responses to improve performance.

Advanced Routing Patterns

Route Parameters

- Extract dynamic values directly from URLs for flexible endpoint design.
- Access using **req.params**.

```
1  // Fetch user by ID
2  app.get('/users/:id', (req, res) => {
3    const userId = req.params.id;
4    res.send(`User ID requested: ${userId}`);
5  });
```

The ':' before a id in the URL path defines it as a **dynamic route parameter**.

Route Handlers

- Chain multiple callback functions for modular and reusable request processing.
- Each function runs sequentially, calling **next()** to pass control.

```
app.get('/data', validateInput, processData, sendResponse);

function validateInput(req, res, next) {
  // Validate request data
  next();
}

function processData(req, res, next) {
  // Process data
  next();
}

function sendResponse(req, res) {
  res.send('Data processed successfully');
}

```

Optional Parameters (?):

- Allow parts of a URL path to be optional.
- Defined by appending a question mark **?** after a route parameter name.
- If value not provided, the parameter value is **undefined**.

```
1  app.get('/products/:Id?', (req, res) => {
2    const productId = req.params.Id || 'default';
3    res.send(`Product ID: ${productId}`);
4  });
5
```

- This route matches both **/products** and **/products/123**.

Optional parameters streamline route handling when certain data may or may not be present.

Wildcards (* or +):

- Used for pattern matching in routes to match any part of the URL.
- '*' matches zero or more characters; '+' matches one or more.
- Useful for matching multiple or unknown path segments.

```
1  app.get('/files/*', (req, res) => {
2    res.send('Wildcard route matched');
3  });
```

- This route matches any path starting with **/files/**,
  e.g., **/files/documents/report.pdf**

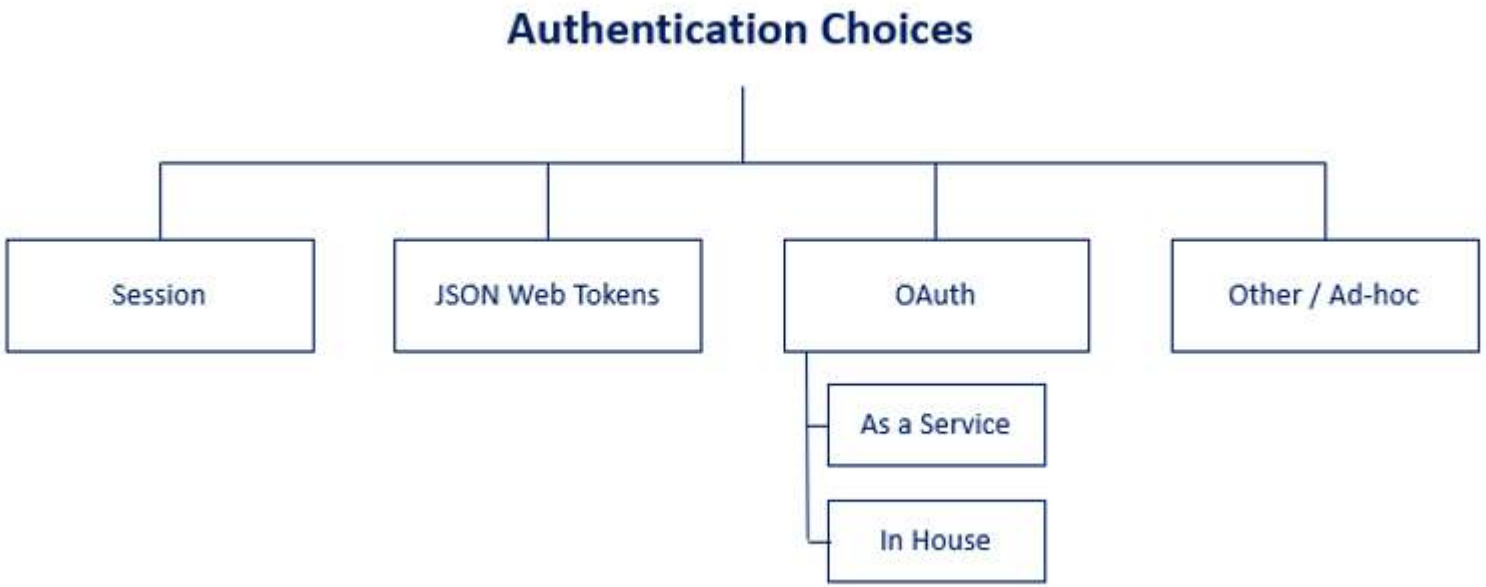
Wildcards provide flexibility for catch-all or undefined paths.

Authentication with Passport.js

- Passport.js is the de facto authentication middleware for Node.js.
- Provides a flexible, modular framework supporting over 500 authentication strategies (OAuth, JWT, local login, etc.).

Key Benefits

- Strategy-based architecture enables easy integration of multiple authentication methods.
- Built-in session management via serialization/deserialization.
- Strong community support and comprehensive documentation.
- Works seamlessly with Express.js middleware.
- **https://www.passportjs.org/**




```
1  import passport from 'passport';
2
3  // Initialize Passport in your Express app
4  app.use(passport.initialize());
5
6  // Then Implement strategies
7
```

Security Best Practices

Helmet.js Protection

- Secure HTTP headers to prevent common vulnerabilities like XSS and click jacking

```
import helmet from 'helmet'
app.use(helmet());
```

Rate Limiting

- Prevent brute-force attacks and API abuse with request throttling

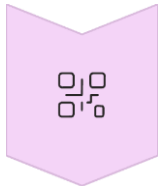
```
import rateLimit from 'express-rate-limit'

// limit each IP to 100 requests per 15 minutes
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 100
});
app.use('/api/', limiter);

```

Input Validation

- Sanitize and validate all user input to prevent injection attacks

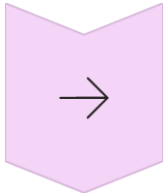
```
import { body, validationResult } from 'express-validator'

app.post('/user', [

  body('email').isEmail(),
  body('password').isLength({ min: 8 })

], (req, res) => {

  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400)
      .json({ errors: errors.array() });
  }

});

```

Error Handling Architecture



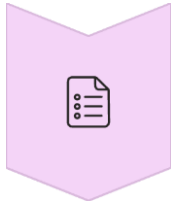
Try-Catch Blocks

Wrap async operations to catch synchronous errors



Error Middleware

Centralized error handling with custom middleware



Logging Service

Record errors for monitoring and debugging

Client Response

Return appropriate status codes and messages

Common Mistake:

Never expose stack traces or internal error details to clients in production. Always use generic error messages and log full details server-side.

Key Takeaways

- Use structured logging with consistent formats across your application
- Implement different log levels (error, warn, info, debug) for granular control
- Rotate log files to prevent disk space issues
- Integrate with monitoring services like Datadog or New Relic
- Log contextual information (user ID, request ID, timestamps) for better debugging


Thank You
CODETANTRA