CipherSchools

Class 3 NodeJS

http Module

The http module in Node.js is a built-in module that allows you to create web servers and handle HTTP requests and responses without any third-party library like Express.

It supports:

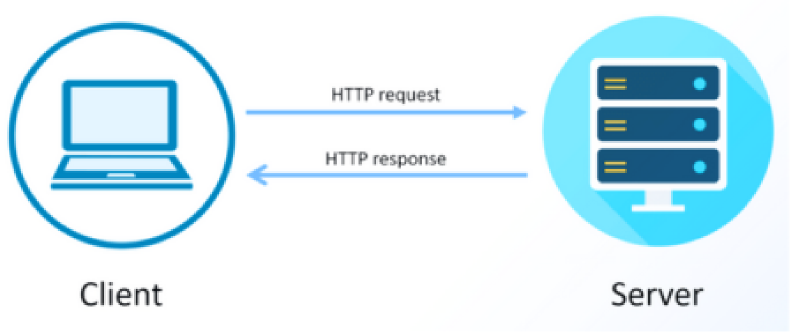
- Creating a server
- Listening on a port
- Handling HTTP methods (GET, POST, etc.)
- Sending and receiving headers, cookies, etc.

You import it like this:

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

Client-Server Model (Simplified)

How the Web Works:



Step-by-step:

1. A client (like a browser) sends an HTTP request (GET, POST, etc.) to a server.

2. The server (your Node.js code) listens for the request and responds with data (HTML, JSON, etc.)
3. The client receives the response and displays or processes it.

Create http server

```
// 1. Import the http module
const http = require('http');

// 2. Create the server using createServer
const server = http.createServer((req, res) => {

  // 4. Set response headers
  res.writeHead(200, { 'Content-Type': 'text/plain' });

  // 5. Send the response body
  res.end('Hello from Node.js HTTP server!');
});

// 6. Server listens on port 3000
server.listen(3000, () => {
  console.log('Server is running on http://localhost:3000');
});
```

Http Server sending response of JSON

```
const http = require("http");

const server = http.createServer((req, res) => {
  console.log("Server is available");
  const student = {
    name: "Aditya",
    college: "LPU"
  };
  res.writeHead(200, { "Content-Type": "application/json" });
  res.end(JSON.stringify(student));
});
```

```
server.listen(3001, () ⇒ {
  console.log("Server is live on port 3001");
});
```

Http Server sending response of HTML

1. Returning Single tag in response

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

// Create server
const server = http.createServer((req, res) ⇒ {
  res.writeHead(200, { 'Content-Type': 'text/html' }); // Set content type to HTML
  res.end('<h1>Hello, LPU!</h1>'); // Send an HTML tag as response
});

// Listen on port 3000
server.listen(3000, () ⇒ {
  console.log('Server running at http://localhost:3000');
});
```

2. Returning Html file in response using stream

   a. index.html (your HTML file)

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Node HTML Response</title>
</head>
<body>
  <h1>Hello from an HTML file!</h1>
</body>
</html>
```

   b. index.js (Server file)

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

// Create the server
const server = http.createServer((req, res) => {
    const filePath = path.join(__dirname, 'index.html');
    fs.readFile(filePath, 'utf8', (err, data) => {
      if (err) {
        res.writeHead(500, { 'Content-Type': 'text/plain' });
        res.end('Internal Server Error');
        return;
      }

      res.writeHead(200, { 'Content-Type': 'text/html' });
      res.end(data);
    });
  } else {
    // Handle unknown routes
    res.writeHead(404, { 'Content-Type': 'text/plain' });
    res.end('404 Not Found');
  }
});

// Start the server
server.listen(3000, () => {
  console.log('Server is running at http://localhost:3000');
});
```

3. Returning Html template with name variable and String replacement
    a. index.html — HTML Template with Placeholder

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Welcome</title>
</head>
```

```
<body>
  <h1>Hello, {{name}}!</h1>
</body>
</html>
```

b. index.js

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

// Sample dynamic variable
const username = 'LPU';

const server = http.createServer((req, res) => {
  const filePath = path.join(__dirname, 'index.html');

  fs.readFile(filePath, 'utf8', (err, htmlContent) => {
    if (err) {
      res.writeHead(500, { 'Content-Type': 'text/plain' });
      return res.end('Internal Server Error');
    }

    // Replace placeholder {{name}} with actual username
    const renderedHtml = htmlContent.replace('{{name}}', username);

    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end(renderedHtml);
  });
});

server.listen(3000, () => {
  console.log('Server running at http://localhost:3000');
});
```

Http Routing

HTTP Routing is the process of defining how your server should respond to different HTTP requests made to specific URLs or paths.

Simple Analogy:
Think of a routing system like a receptionist at an office:

- If you say "I want to meet HR", the receptionist sends you to HR.
- If you say "I want to meet Sales", they send you to Sales.

Similarly, in a Node.js server, if a user hits /home, you return a homepage. If they hit /about, return the about page.

Example Without Express (Core http module)

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

const server = http.createServer((req, res) => {
  const url = req.url;

  if (url === '/') {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end('<h1>Home Page</h1>');
  } else if (url === '/about') {
    res.writeHead(200, { 'Content-Type': 'text/html' });
    res.end('<h1>About Page</h1>');
  } else {
    res.writeHead(404, { 'Content-Type': 'text/html' });
    res.end('<h1>404 - Page Not Found</h1>');
  }
});

server.listen(3000, () => {
  console.log('Server running on http://localhost:3000');
});
```

💡 What's Happening?

Part	Meaning
req.url	Gives the path the user requested (like /about)

res.writeHead(...)	Sets response type and status
res.end(...)	Sends the response to the client

Browser Runtime vs Node.js Runtime

- JS itself doesn’t run on its own — it needs a runtime.
- V8 is the engine (from Chrome) used by both browsers and Node.js.
- In Node.js, the runtime includes V8 + libuv + Node APIs.

Component	Browser	Node.js
JS Engine	V8 (Chrome)	V8
APIs	Web APIs (DOM, fetch)	Node APIs (fs, http)
Async System	Web APIs + Event Loop	libuv + Event Loop

NodeJS Runtime Environment

A JavaScript runtime is the environment in which JavaScript code executes.

It consists of:

- *Call Stack* - Where functions are executed.
- *Heap* - Where memory (variables, objects) is stored.
- *libuv (in Node.js)* - Handles things like timers, HTTP, file system.
- *Callback & MicroTask Queue* - Stores tasks waiting to be executed.
- *Event Loop* - The traffic controller that coordinates all the above.

How JavaScript Executes Code (Sync)

Synchronous JavaScript

Example:

```
console.log('A');
console.log('B');
console.log('C');
```

Stack behavior:

- Push console.log('A')
- Push console.log('B')
- Push console.log('C')
- Pop, execute in order

All synchronous - executed line by line.

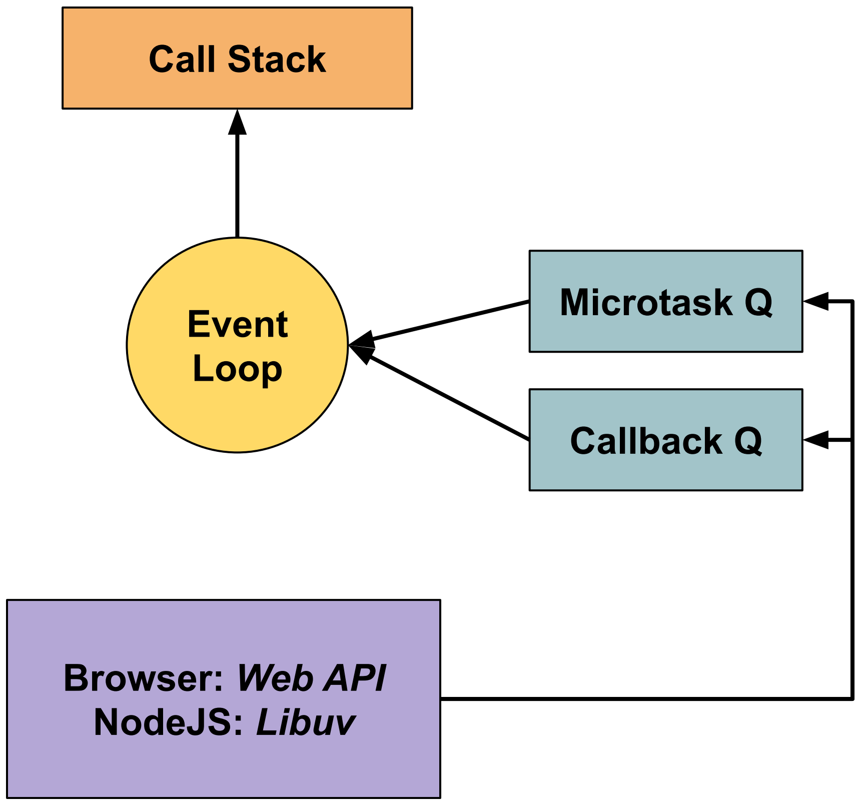
Asynchronous JavaScript: Why We Need It

JS runs on a single thread, so blocking operations (e.g., file read, DB call, HTTP request) would freeze the app.
Instead, async operations let the **Main Thread** keep working, while long tasks **Run in the background**.

The Magic: Event Loop (Core Concept)

The event loop is the system that:

- Picks up async tasks when they’re ready (e.g., file done reading)
- Places their callbacks on the callback queue
- Pushes them to call stack when it's empty



How Node.js Does Async Work

Node.js uses libuv, a C++ library, to handle:

- File I/O
- Networking
- Timers
- DNS
- Thread pool

Steps (Example: fs.readFile):

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

fs.readFile('file.txt', 'utf8', (err, data) => {
  console.log(data);
});
```

Behind the scenes:

- fs.readFile is offloaded to **libuv**.
- It runs on a thread from the **thread pool**.

- When done, the result is added to the **callback queue**.
- The **event loop** puts it on the **call stack**.
- Your callback runs and logs data.

This is how Node handles asynchronous I/O without blocking the main thread.

Thread Pool

What is a Thread Pool?

- A thread pool is a group of worker threads maintained by a system to perform expensive or blocking tasks in parallel without blocking the main thread.
- Instead of creating a new thread every time a task is run (which is expensive), a pool of threads is created once and reused.

Why Node.js Needs a Thread Pool

Node.js is single-threaded for JavaScript execution, but:

- It offloads expensive/blocking tasks (like file I/O, DNS lookup, encryption, compression) to a thread pool.
- This pool is managed by libuv, a C/C++ library bundled with Node.js.

So, thread pool enables concurrency and parallelism for non-JS tasks.

How libuv Uses the OS Kernel

- libuv is written in C/C++, and it's a layer between Node.js and the OS.
- It uses OS-level APIs like *pthreads* (Linux/Unix), *CreateThread* (Windows) to manage actual threads.
- It creates and manages a thread pool of software threads.
- These threads are scheduled and executed by the OS kernel scheduler.

Node.js → libuv → OS Kernel → CPU Core Schedulers

Demonstration: Password Hashing with crypto.pbkdf2()

Synchronous Code:

```
const crypto = require("crypto");

console.log("Total Time");

const MAX_TIME = 4;
const now = Date.now();
for (let i = 0; i < MAX_TIME; i++) {
  crypto.pbkdf2Sync("password", "salt", 100000, 64, "sha512");
  console.log(`Task ${i}: ${Date.now() - now}`);
}
```

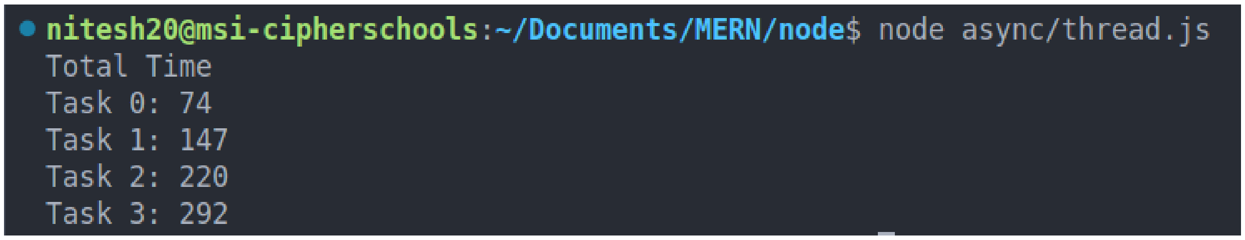
crypto.pbkdf2Sync(...)

- A synchronous function for hashing passwords.
- CPU-intensive because it runs 100,000 iterations to derive a secure key.
- Uses the main (JavaScript) thread - no thread pool, no offloading.

pbkdf2Sync blocks the thread:

- Each task must be completed before the next starts.
- No concurrency.
- The main thread is blocked until the function returns

Output:



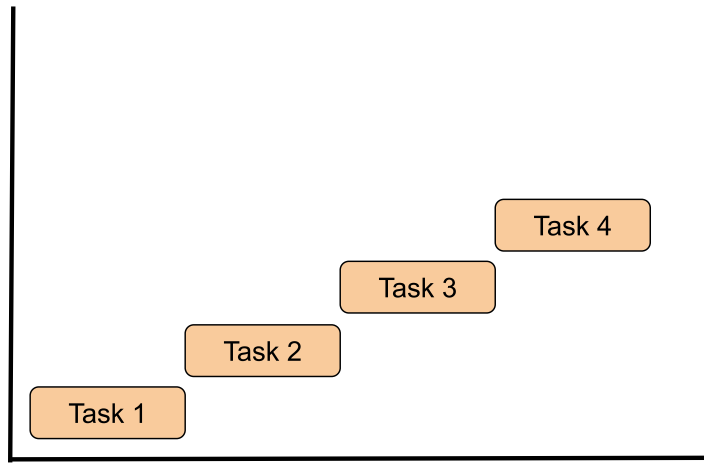
```
● nitesh20@msi-cipherschools:~/Documents/MERN/node$ node async/thread.js
  Total Time
  Task 0: 74
  Task 1: 147
  Task 2: 220
  Task 3: 292
```

What It Shows:

- Task 0 took ~74 ms.
- Each subsequent task took about the same time in addition.
- Time accumulates: 74 + ~73 + ~73 + ~72 ≈ 292 ms total.

This is expected for synchronous, blocking behavior - no overlap in work.

Visual Flow (Timeline)



(Only one runs at a time)

Real-World Implication

- In real servers (e.g., Express apps), using pbkdf2Sync means:
- The main thread can't serve other requests while hashing.
- This leads to poor performance and delays.

Conclusion: Synchronous code is simple but dangerous in a Node.js server for heavy tasks like password hashing.

Asynchronous Code:

```
const crypto = require("crypto");

console.log("Total Time");

const MAX_TIME = 4;
const now = Date.now();
for (let i = 0; i < MAX_TIME; i++) {
   crypto.pbkdf2("password", "salt", 100000, 64, "sha512", () => {
     console.log(`Task ${i}: ${Date.now() - now}`);
   });
}
```

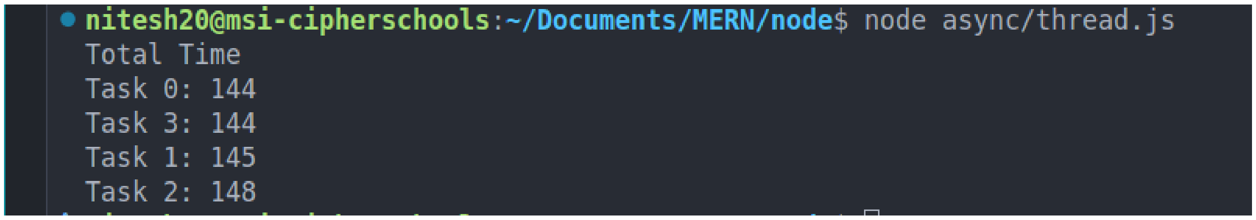
What's Happening Here?

- crypto.pbkdf2() is Asynchronous
- Non-blocking function.
- The hashing work is offloaded to the thread pool managed by libuv.
- The main thread is free to continue, and callbacks are executed when threads finish.

Default Thread Pool Size: 4

Since Node.js's libuv thread pool size is 4 by default, and we have 4 tasks (MAX_TIME = 4), all 4 run concurrently.

Expected Output



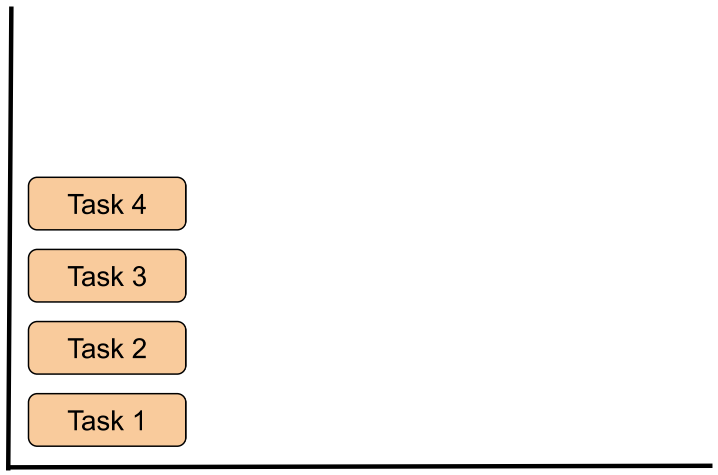
```
nitesh20@msi-cipherschools:~/Documents/MERN/node$ node async/thread.js
Total Time
Task 0: 144
Task 3: 144
Task 1: 145
Task 2: 148
```

Note: Timing may vary slightly on different machines

What It Shows:

- All tasks start at the same time.
- Each task completes around the same time.
- Total execution time is roughly equal to the slowest individual task — not the sum.

Visual Flow (Async/Parallel Execution)



Results return in ~145ms for all

How libuv Handles This

- JavaScript (main thread) calls pbkdf2(...) 4 times.
- libuv puts each task in its work queue.
- Each task is picked up by one of the 4 threads in the thread pool.
- When a thread completes the hash, it adds the callback to the event loop queue.
- The event loop runs the callback - prints output.
- No blocking on the main thread!

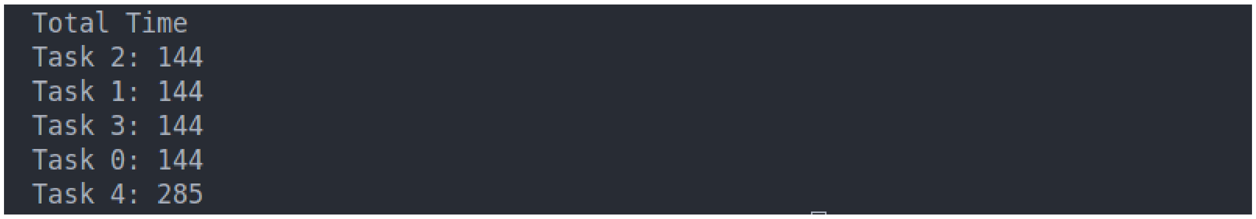
What happens if we increase the process beyond 4.

```
const crypto = require("crypto");

console.log("Total Time");

const MAX_TIME = 5;
const now = Date.now();
for (let i = 0; i < MAX_TIME; i++) {
   crypto.pbkdf2("password", "salt", 100000, 64, "sha512", () => {
     console.log(`Task ${i}: ${Date.now() - now}`);
   });
}
```

Expected Output



```
Total Time
Task 2: 144
Task 1: 144
Task 3: 144
Task 0: 144
Task 4: 285
```

Explanation:

- First 4 tasks start immediately
- 5th task waits (queues) until one thread is free
- It adds ~140ms more total time

How to Increase Thread Pool Size

The environment variable *UV_THREADPOOL_SIZE* controls the libuv thread pool size (max 128).

1. Linux/macOS Terminal:

```
UV_THREADPOOL_SIZE=8 node async/thread.js
```

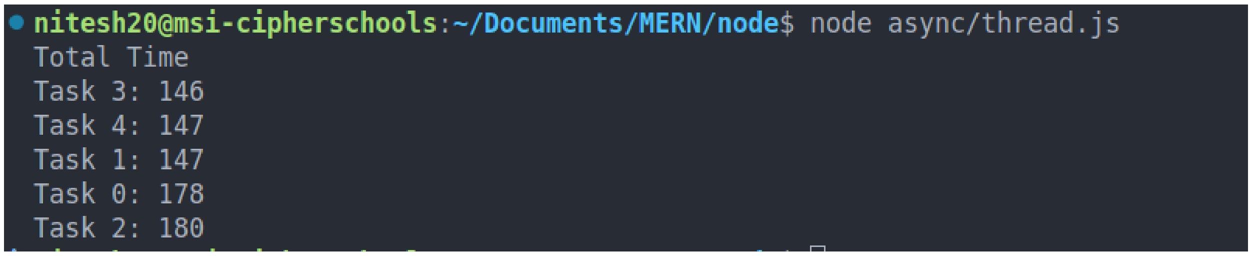
2. Windows CMD:

```
set UV_THREADPOOL_SIZE=8 && node pbkdf2-async.js
```

3. In Code (must be first!):

```
process.env.UV_THREADPOOL_SIZE = 8;
```

Expected Output



```
nitesh20@msi-cipherschools:~/Documents/MERN/node$ node async/thread.js
Total Time
Task 3: 146
Task 4: 147
Task 1: 147
Task 0: 178
Task 2: 180
```

How Thread Pool Works with CPU Cores

Thread ≠ Core

- A CPU core can run multiple threads via time-slicing
- Node.js thread pool uses OS threads, scheduled on CPU cores by the OS kernel

Thread Pool Internals:

- When a thread pool task is triggered (like pbkdf2()), libuv puts it into its work queue
- Threads pick up jobs from the queue
- OS schedules threads on available CPU cores
- Once complete, the thread hands the result back to the Node.js event loop via a callback

What is Time Slicing?

- Time slicing is a technique used by operating systems to simulate parallelism on a single CPU core by rapidly switching between threads or processes.

In simple terms:

- Time slicing gives each process/thread a small "slice" of CPU time, then switches to the next, giving the illusion of simultaneous execution.

Example with 1 CPU Core, 3 Threads(*Thread pool)*

Time (ms)	CPU running
0–10	Thread A
10–20	Thread B
20–30	Thread C
30–40	Thread A (again)

This switching is so fast, it feels like threads are running in parallel - even though there's only one core. Due to which, even on a dual-core CPU, 8 threads from UV_THREADPOOL_SIZE=8 can run efficiently.

Worker Thread

Worker Threads in Node.js provide a way to run JavaScript in parallel on multiple threads.

Unlike the libuv thread pool (used for I/O), Worker Threads can run JS code (like loops, computation, logic) in separate threads.

They are useful for:

- CPU-intensive tasks
- Background processing
- Isolated script execution

Why Worker Threads?

Node.js is single-threaded for JavaScript by default.

This means:

- Long-running tasks (e.g., big loop, image processing) block the event loop.
- The server becomes unresponsive during such operations.

Worker Threads solve this by moving such tasks to separate threads, leaving the main thread free.

Basic Example of Worker Thread

Step 1: Create worker.js (the worker file)

```
// worker.js
const { parentPort } = require('worker_threads');

let sum = 0;
for (let i = 0; i < 1e9; i++) {
  sum += i;
}

parentPort.postMessage(sum); // send result back
```

Step 2: Create main.js

```
const { Worker } = require('worker_threads');
console.log("Main thread starts");
const worker = new Worker('./worker.js');

worker.on('message', (result) => {
  console.log('Sum is', result);
});

worker.on('error', (err) => console.error('Worker error:', err));
worker.on('exit', (code) => console.log('Worker exited with code',
code));

console.log("Main thread is still running...");
```

Expected Output:

```
Main thread starts
Main thread is still running...
Sum is 4999999995000000000
Worker exited with code 0
```

Even while the worker computes the sum, the main thread keeps running. No blocking!

Important Note:

This is an advanced topic.

- You just need to understand what Worker Threads are and how they help Node.js achieve **parallelism**.
- While the Thread Pool is used for achieving **concurrency**, Worker Threads are used for parallel execution of JavaScript code.
- You should also have a clear understanding of the difference between the Thread Pool and Worker Threads, and how they serve different purposes in Node.js.

External Module

In Node.js, external modules are third-party packages that are:

- Not built into Node.js core (like fs, http)
- Installed via npm (Node Package Manager)
- Reside in the node_modules/ directory
- Can be reused across multiple projects

Why Use External Modules?

- To avoid reinventing the wheel
- To add new functionality easily (e.g., HTTP clients, validators, loggers)
- To build scalable, modular apps

Examples of Common External Modules

Package	Purpose
express	Web server / routing framework
lodash	Utility functions
dotenv	Load environment variables
axios	HTTP client for APIs
mongoose	MongoDB object modeling
chalk	Colored console output
cors	Enable cross-origin requests

How to Use an External Module (Step-by-Step)

1. Initialize npm project

```
npm init -y
```

This creates a package.json file to manage your project dependencies.

2. Install a package (e.g., chalk)

```
npm install chalk
```

Now node_modules/ is created, and chalk is added to package.json.

3. Import and use it in your JS file

```
const chalk = require('chalk');
console.log(chalk.green('Hello from chalk!'));
```

Task:

1. Build a colorful terminal logger that logs messages in different colors based on type (info, success, error)
2. Build a module to convert a csv data from *data.csv* to the JSON data and store it in *data.json* using *csvtojson* package

What is Express.js?

Express.js is a minimal, flexible Node.js web application framework that provides tools to:

- Handle routing (URLs like /, /about)
- Manage HTTP requests/responses
- Use middlewares for processing requests
- Serve HTML, JSON, static files
- Build REST APIs and full apps easily
- It simplifies building servers in Node.js.

What is res in Express?

- res stands for response.
- It's an object that represents the HTTP response that your server will send back to the client (browser, Postman, etc).

Commonly used res methods:

Method	Description
res.send(data)	Sends a response (auto detects string, buffer, or object)
res.json(obj)	Sends a JSON response
res.status(code)	Sets HTTP status (e.g. 200, 404, 500)

res.sendFile(path)	Sends a file as response
res.redirect(url)	Redirects to another URL
res.end()	Ends the response process manually

Task: Create First Server with Express

Step-by-Step

1. Initialize project

```
mkdir my-express-app
cd my-express-app
npm init -y
npm install express
```

2. Create server.js

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

// Home route
app.get('/', (req, res) => {
  res.send('Hello from Express!');
});

// About route
app.get('/about', (req, res) => {
  res.send('This is the About page.');
});

// JSON response
app.get('/api/data', (req, res) => {
  res.json({ name: "Express Server", status: "Running" });
});

// Start server
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server running on http://localhost:${PORT}`);
});
```

3. Run the server

```
node server.js
```

Open browser and try:

- http://localhost:3000/ → "Hello from Express!"
- http://localhost:3000/about → "This is the About page."
- http://localhost:3000/api/data → JSON response

Key Concepts Explained

1. express():
    - Initializes your app.
    - Sets up methods like .get(), .post(), etc.

2. app.get(route, handler):
    - Defines a GET route.
    - route is the URL path.
    - handler is a function with (req, res).

3. res.send():
    - Sends plain text or HTML.
    - Ends the response automatically.

4. res.json():
    - Sends JSON data.
    - Useful for APIs.

5. app.listen(port):
    - Starts the server and listens for incoming requests.

Tasks:

1. Create a server which has 3 routes
    a. / ⇒ Root Page (Plain text response)
    b. /contact ⇒ Contact Us Page(HTML file response)
    c. /about ⇒ About(HTML Tag response)
    d. /data ⇒ JSON response

//CODE:

```javascript
const express = require("express");
const path = require("path");

const app = express();

const student = {
  name: "Aditya",
  college: "LPU"
};
// / => Root Page (Plain text response)
app.get("/", (req, res) => {
  res.status(200).header({ "Content-Type": "plain/text"
}).send("Hello LPU").end();
});

// /contact => Contact Us Page(HTML file response)
app.get("/about", (req, res) => {
  res.status(200).header({ "Content-Type": "text/html"
}).send("<h1>About</h1>").end();
});

// /about => About(HTML Tag response)
app.get("/contact", (req, res) => {
  res
    .status(200)
    .header({ "Content-Type": "text/html" })
    .sendFile(path.join(__dirname, "contact.html"));
});

// /data => JSON response
app.get("/data", (req, res) => {
  res.status(200).header({ "Content-Type": "application/json"
}).json(student).end();
});

app.listen(3001, () => console.log("Server is running on 3001"));
```