BACKEND

1) .ENV (Environment Variables)
2) Serve Files From Server
3) Mongoose for MongoDB
4) express.json()
5) CORS(Cross Origin Resource Sharing)
6) JWT (Json Web Token)
7) Nodemailer
8) Nodejs
9) libuv
10) Thread Pool
11) Modules
12) CommonJS
13) ES Modules
14) Import Export Patterns (CommonJS)
15) Import Export Patterns (ECMAScript Module aka ESModules)
16) Module Scope
17) Module Wrapper & Module-scoped variables
18) Module Caching
19) Watch Mode
20) Build in Modules
21) Path Modules
22) Events Modules
23) Streams
24) Pipes
25) Buffers
26) HTTP Module
27) View [MVC -> V(view)]
28) Web Framework
29) Event Loop
30) I/O Polling
31) NPM
32) package.json
33) Versioning
34) Scripts
35) Building CLI Tools
36) Cluster Module
37) PM2 (Process Manager 2)
38) Worker Thread

39) Libuv Threads vs Worker Threads
40) V8 Engine
41) ExpressJs
42) ExpressJS Response-Methods
43) ExpressJs Middlewares

Backend Guide : https://github.com/goldbergyoni/nodebestpractices

🧭 **Backend concepts you should absolutely learn (beginner → advanced):**

🔹 1. HTTP & REST basics Methods (GET, POST, PUT, DELETE) Status codes (200, 404, 401, 500) Headers, query params, body	🔹 2. Authentication & Authorization JWT (JSON Web Tokens) Cookie-based sessions Role-based access control (RBAC)	🔹 3. Middleware & Routing (Express, NestJS, etc.) app.use(), req, res Route-level and global middleware
🔹 4. Database integration MongoDB / PostgreSQL / MySQL ORMs like Prisma or Mongoose Query optimization	🔹 5. CORS & Cross-origin requests	🔹 6. Error Handling Centralized error middleware Logging (Winston, Pino) Custom error classes
🔹 7. File Uploads Multer (Express) Cloud storage (S3, Cloudinary)	🔹 8. Security Best Practices Input validation Rate limiting Helmet / CSRF	🔹 9. Testing APIs Postman Jest / Supertest
	🔹 10. API Design Principles Versioning Pagination REST vs GraphQL	

ENV (Environment Variables)

The .env file is a simple text file that stores environment variables for an application. It's commonly used to store sensitive configuration settings, such as API keys, database credentials, and other values that might vary across different environments

(development, staging, production). By keeping these settings in a separate file, they can be managed more easily and securely, and the code itself remains cleaner.

1. **Key-Value Pairs**: The file contains key-value pairs, where the key is the name of the environment variable, and the value is its corresponding value.
2. **Security**: .env files are typically excluded from version control systems (e.g., Git) to prevent accidental exposure of sensitive information.
3. **Flexibility**: They allow for easy adaptation to different environments by simply modifying the values in the .env file.

Ex :

```
1. DB_HOST=“localhost”
2. DB_USER=“myuser”
3. DB_PASSWORD=“mypassword”
4. API Keys: API_KEY=“your_api_key”
```

Nodejs Implementation :

1. npm i dotenv
2. in the main server.js file (where the port is running), import dotenv package require('dotenv').config(); This command will help access the env variables in all other files because all files are run through server.js
3. Now whereever you want to use the variables defined in the .env file, just use process.env.VARIABLE

Note : Sometimes, the file doesn't recognize what process.env.VARIABLE is, for that import the dotenv package in that specific file again

What is process ?

The process object is a Node.js object. It provides information about, and control over, the current Node.js process. It is a global object, meaning it is always available in Node.js applications without requiring an import statement. While Node.js uses JavaScript as its language, the process object itself is not a standard JavaScript object available in browser environments. It's specific to the Node.js runtime environment.

more about process object : https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/dotenv-and-global-access.png

Serve Files (Assets) From Server

https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/serve-files-from-server.png

By default, no one can access files on a server just by using a link to the server. To allow public access to files stored on the server, we can use middleware provided by Express that serves static files via their file paths.

(Note: In real-world applications, files are often stored on cloud platforms like AWS S3 instead of directly on the server.)

Imagine the following file structure on the server:

└── Public/

  └── image.jpg

To give public access to image.jpg via a URL, we can use the following Express middleware:

```
1. app.use(‘/files’, express.static(path.join(process.cwd(), ‘Public’)))
```

Now, image.jpg will be accessible through this URL: https//your-domain.com/files/image.jpg

Mongoose for MongoDB

```
>> npm i mongoose
```

Mongoose is an Object Data Modeling (ODM) library for MongoDB, a NoSQL database. It provides a higher-level abstraction for working with MongoDB, making it easier for developers to interact with the database using JavaScript. Mongoose simplifies data modeling, schema creation, and validation, allowing developers to work with MongoDB in a more structured and intuitive way.

Here's a more detailed breakdown:

ODM (Object Data Modeling):

Mongoose allows developers to model data using JavaScript objects, which are then mapped to MongoDB documents.

Schema-based:

Mongoose uses schemas to define the structure of documents in MongoDB collections, ensuring data integrity and consistency.

Simplifies database interaction:

Mongoose provides a more developer-friendly way to interact with MongoDB compared to using the native MongoDB driver, which requires more manual query building and schema management.

Benefits of using Mongoose:

Easier data modeling: Mongoose makes it easier to create and manage data structures, especially for complex applications.

Built-in features:

Mongoose includes features like type casting, validation, and query building, reducing the need for manual coding.

Improved code organization: Using Mongoose can lead to cleaner and more organized code, as it handles many of the low-level database operations.

MongoDB v/s Mongoose

Without Mongoose package, we have to use mongodb package(npm i mongodb) to insert data in DB, but mongodb package doesn't provide schemas or validations at the application level.

Mongoose provide more facilities and security then mongodb.

If you don't want to create schema and just want to insert data for small project, use mongodb. But if you want more restriction and validations, use mongoose.

mongodb :

- No schemas or validations at the application level.
- No middleware/hooks (like pre-save, post-remove).
- You must manually handle: Validation, Relationships, Data transformations, Timestamps, defaults, etc.

mongoose :

- Built on top of the mongodb driver.
- Adds schema definitions, model-based data access, and validation.
- Offers middlewares, virtuals, population (joins), plugins, etc.

Guide on how to use Mongoose to connect to MongoDB, define a schema, and perform basic queries :

➢ **1. Install Mongoose**

```
1. npm install mongoose
```

➢ **2. Connect to MongoDB**

```
 1. const mongoose = require('mongoose');
 2.
 3. const url = 'mongodb+srv://jubernowawave:JUBERkhan%40123@cluster0.n4tvq.mongodb.net/Eportal'; // connection url from mongodb atlas
 4.
 5. mongoose.connect(url, {
 6.   useNewUrlParser: true,
 7.   useUnifiedTopology: true
 8. })
 9. .then(() => console.log('MongoDB connected'))
10. .catch(err => console.error('MongoDB connection error:', err));
```

➢ 3. Define a Schema & Model

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

const userSchema = new mongoose.Schema({
  first_name: { type: String, required: true },
  last_name : String,
  age: Number,           // Number
  isActive: Boolean,     // Boolean
  birthday: Date,        // Date
  profilePic: Buffer,    // Binary Buffer. Used for storing binary data like files, images, or encrypted content.
  sport: {
    type: mongoose.Schema.Types.ObjectId, // ObjectId reference
    ref: 'Sports' // ObjectId from Sports collection/table
  },
  tags: [String], // Array of Strings
  preferences: mongoose.Schema.Types.Mixed, // Mixed. A flexible type that can hold any kind of value — object, array, string, etc.
  meta: {         // Map of dynamic keys
    type: Map,
    of: String
  },
  price: mongoose.Schema.Types.Decimal128, //  Used for storing high-precision decimal numbers (like currency). eg: 999.99
}, {
  timestamps: true // adds createdAt and updatedAt
});

const User = mongoose.model('User', userSchema);
module.exports = User;
```

4. Use the Model to Perform Queries

```
const User = require('./models/User');

// Create a new user
const newUser = await User.create({ name: 'Alice', age: 25, email: 'alice@example.com' });

// Find one user
const foundUser = await User.findOne({ name: 'Alice' });

// Update a user
await User.updateOne({ name: 'Alice' }, { age: 26 });

// Delete a user
await User.deleteOne({ name: 'Alice' });

// Find all users
const allUsers = await User.find();
```

More on MongoDB in MongoDB notes :

express.json()

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

express.json() is a built-in middleware function in Express.js that parses incoming requests with JSON payloads. It handles JSON data sent from clients and makes it accessible within your server-side application. This allows you to easily work with data sent by clients, especially when building RESTful APIs that use JSON.

Here's a more detailed explanation:

Parsing JSON:

When a client sends data to your server in the form of JSON, express.json() parses this JSON data and attaches it to the req.body property of the request object.

Accessibility:

This means you can access the parsed JSON data within your route handlers using req.body. For example, req.body.name would give you access to the 'name' field in the JSON data sent from the client.

Middleware Function:

express.json() is a middleware function, meaning it's a piece of code that can modify the request object or respond to a request. It's typically used with app.use() to apply it globally or with individual routes for more targeted parsing.

body-parser Dependency:
Historically, Express.js relied on the body-parser library for parsing request bodies. However, in newer versions of Express, express.json() is now a direct part of the framework and essentially replaces the need for body-parser when dealing with JSON data.

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

// Use express.json() as middleware
app.use(express.json());

app.post('/api/data', (req, res) => {
  // req.body now contains the parsed JSON data
  const jsonData = req.body;
  console.log(jsonData);  // Output: { name: 'John', age: 30 }
  res.json({ message: 'Data received successfully' });
});

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

In this example, any request sent to /api/data with a JSON payload will have its data parsed and made available in req.body. This makes it easy to access and work with the data within the route handler.

CORS (Cross Origin Resource Sharing)

It is a security feature built into web browsers that controls how resources on a server can be requested from another domain (or "origin") outside the one the resource originated from.
CORS is a protocol that uses HTTP headers to tell browsers whether a specific web application running at one origin (eg. domain.com) is allowed to access resources from a different origin (eg. domain1.com).
Example scenario:
1. Your frontend is hosted at https://example-frontend.com.
2. Your backend API is hosted at https://api.example-backend.com.
3. The browser blocks API requests from frontend to backend unless the backend includes proper CORS headers (e.g., Access-Control-Allow-Origin).

Why do we need CORS in the backend?
Security: Prevent unauthorized websites from reading sensitive data from your server.
Controlled Sharing: Only allow access to known and trusted frontend domains.
Avoid Same-Origin Policy issues: Browsers enforce the Same-Origin Policy, which blocks cross-origin requests by default for security. CORS allows safe and secure controlled exceptions to this rule.

```
app.use(cors());
```

in Express, This line tells your Express.js app to enable CORS (Cross-Origin Resource Sharing) with default settings.

What it does exactly:
Allows requests from any origin (Access-Control-Allow-Origin: *)
Does not allow credentials (like cookies, sessions, or Authorization headers)

Tip : to restrict public usage of your api and only allow defined frontend domain to access your backend APIs :

```
app.use(cors({
  origin: 'http://your-frontend.com',
  methods: ['GET', 'HEAD', 'PUT'], // only these methods will be allowed, all the rest methods PATCH, POST, DELETE will be blocked
}));
```

When you use:

```
app.use(cors());
```

It's shorthand for:

```
app.use(cors({
  origin: '*',
  methods: ['GET', 'HEAD', 'PUT', 'PATCH', 'POST', 'DELETE'],
}));
```

full detailed discussion : (https://chatgpt.com/c/6821960c-80d8-8012-8d48-0b5dc3da0684)
https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/CORS.png
https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/CORS1.png
https://www.geeksforgeeks.org/cross-origin-resource-sharing-cors/
https://www.geeksforgeeks.org/http-access-control-cors/
https://developer.mozilla.org/en-US/docs/Web
https://www.youtube.com/watch?v=E6jgEtj-UjI

JWT (Json Web Token)

JSON Web Tokens (JWTs) are used primarily for authentication and authorization in web applications and APIs. They provide a secure way to share information between two parties, usually a client and a server, in a compact and self-contained format. JWTs allow for stateless authentication, meaning the server doesn't need to store session information, which improves scalability.
JWTs (JSON Web Tokens) can be securely stored on the client side, typically in cookies, allowing user sessions to persist without requiring frequent re-authentication. As long as the JWT remains valid (i.e., until it expires), users can revisit the site without needing to log in again, since the token can be used to verify their identity on subsequent requests.

JWT Authentication Process :

1. user login with his credentials
2. authenticate his credentials(email-password) from backend
3. if authenticated, fetch user data from db and convert that user data to jwt token and send to frontend.
   converting data to jwt token :

```
npm i jsonwebtoken
```

```
const jwt = require('jsonwebtoken')
jwt.sign(dataToEncrypt, process.env.JWT_KEY, { expiresIn: '7d' },(error, token)=>{
    if(error) return res.status(500).json({message:'try after some time!'});
    res.status(200).json({message:'success',token})
})
```

   ExpiresIn Eg: 60, "2 days", "10h", "7d". A numeric value is interpreted as a seconds count. If you use a string be sure you provide the time units (days, hours, etc), otherwise milliseconds unit is used by default ("120" is equal to "120ms").
   https://www.npmjs.com/package/jsonwebtoken#:~:text=numeric%20value
4. frontend get the token and store it in cookies.
5. backend create a verifyToken api to verify the token
6. frontend send token in verifyToken api in the headers

```
const res = await axios.post('https://localhost:4000/verifyToken',{},{
headers:{
  'Authorization':jwtToken
}})
```

7. Backend verify the token and send the actual user data.

```
jwt.verify(req.headers.authorization, process.env.JWT_KEY,(error, decoded)=>{
if(error) res.status(401).json({message:'please login again!'})
   res.status(200).json({message:'success',data:decoded})
})

```

   Error : After receiving the error(invalid token, expired token), the frontend should log out the user and redirect to login page and generate the jwt again (from step 1)
   Success : store encrypted userdata in context that was send from verifyToken api
8. The actual user data(decoded) will be saved in the context.
9. The jwt token will stay till its expiry date in the cookie.
10. On Every refresh, the verifyToken api is called with the jwt token (to convert token to userdata)

Check authentication on every protected API route in backend with auth middleware, jwt will be send with header/cookie from frontend :

```
// middleware/auth.js
const jwt = require('jsonwebtoken');

function authenticateToken(req, res, next) {
  const authHeader = req.headers['authorization'];
  const token = authHeader?.split(' ')[1]; // Bearer <token>

  if (!token) return res.status(401).json({ message: 'No token provided' });

  jwt.verify(token, process.env.JWT_SECRET, (err, user) => {
    if (err) return res.status(403).json({ message: 'Invalid token' });

    req.user = user; // Store user info in req for use in routes
    next();
  });
}

module.exports = authenticateToken;
```

Use it in routes:

```
const authenticateToken = require('./middleware/auth');

app.get('/api/profile', authenticateToken, (req, res) => {
  res.json({ message: `Hello ${req.user.name}` });
});

app.post('/api/update', authenticateToken, (req, res) => {
  // Only logged-in users can update data
});
```

Nodemailer

Nodemailer is a popular Node.js module that simplifies sending emails from your server. It provides a user-friendly API for composing and sending email messages, including support for various email protocols like SMTP, and features like HTML content, attachments, and inline images.

https://nodemailer.com/

"To use Nodemailer with a Gmail account, you must enable 2-Step Verification on the Gmail account that will be used to send emails. This is a security requirement from Google. Once 2-Step Verification is enabled, you can generate an App Password, which Nodemailer will use to authenticate and send emails securely through the Gmail SMTP server."

```
npm install nodemailer
const nodemailer = require('nodemailer');
  const transporter = nodemailer.createTransport({
    service : 'GMAIL',
    auth : {
      user : process.env.EMAIL,
      pass : process.env.PASS // generated App Password
      }
   })
 const options = {
   tfrom : process.env.EMAIL,
   to : req.body.email,
   subject : 'OTP for update email'
   text : `Your OTP is ${OTP}`
   }
 transporter.sendMail(options,(error, success)=>{
   if(error) return res.status(500).json(message:'try after some time!');
   return res.status(200).json(message:'OTP send to you mail');
  });
```

How to get the App Password:

Enable 2-Step Verification on your Gmail account: https://myaccount.google.com/security

Once enabled, go to: https://myaccount.google.com/apppasswords

Select App: Mail

Device: Other (Custom name) — you can write "Nodemailer"
Google will generate a 16-character password. Use that in your code as process.env.PASS

How to store OTP in backend for multiple user :

```
1. const otpMap = new Map();
2. otpMap.set('user@example.com', '123456');
```

remove OTP after certain time :

```
1. setTimeout(() => otpMap.delete('user@example.com'), 5 * 60 * 1000); // expires in 5 mins
```

A Map is more reliable and better performance for frequent insertion/deletion than an objects

Nodejs

Nodejs is the code written in c++ and javascript. Nodejs runtime unlike the browser runtime does not have access to the web APIs, there is no window or document when working with Nodejs
Node.js is an open-source, cross-platform Javascript runtime environment.
It is not a language, it is not a framework. It is a runtime environment.
Capable of executing JS code outside a browser.
It can execute not only the standard ECMAScript language but also new features that are made available through C++ bindings using the V8 engine
It consists of C++ files which form the core features and javascript files which expose common utilities and some of the C++ features for easier consumption

With Nodejs, we can create :
1. Traditional websites
2. Backend services like APIs
3. Real-time applications
4. Streaming services
5. CLI tools
6. Multiplayer games

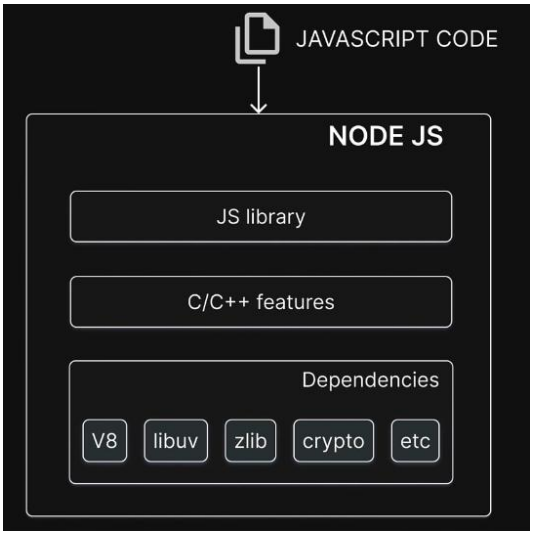
lets take file system as an example.
JavaScript, when run in the browser, cannot access the file system for security reasons(Access to the file system is blocked to protect user privacy and security.). However, Node.js extends JavaScript with modules like fs that allow file system operations, using underlying C++ code to interact with the OS.

this (https://github.com/nodejs/node/blob/main/lib/fs.js) is the nodejs code of fs module that handles file system manipulation with the help of C++, so we don't have to write it from scratch. We just import fs module.
Check node version :

```
1. node -v
```

Executing Javascript with Node (REPL)
1. Read
2. Evaluate
3. Print
4. Loop

Nodejs Architecture :



Browser vs Node.js

In the browser, most of the time what you are doing is interacting with the DOM, or other Web Platform APIs like Cookies. In case of Node, we don't have the document, window and all the other objects that are provided by the browser.
And In the browser, we don't have all the nice APIs that Node.js provides through its modules. For example the filesystem access functionality.
With Node.js, you control the environment. But with a browser, you are at the mercy of what the users choose

libuv

libuv is a cross platform open source library written in C language. handles asynchronous non-blocking operations in Node.js
How does libuv do that : using "Thread pool" & "Event loop"

libuv is the C++ library used by Node.js to provide:

1. Asynchronous I/O
2. Event loop
3. Thread pool
4. Timers
5. TCP, UDP, Pipes
6. File system access

Node.js sits on top of libuv. The thread pool is implemented and managed by libuv, not JavaScript directly.

Libuv is an open-source, cross-platform C library that provides core infrastructure for Node.js's asynchronous, non-blocking I/O operations. It was originally developed specifically for Node.js but is now used in other projects as well.
Here's a breakdown of its key roles in Node.js:
Event Loop Management:
Libuv implements the event loop, which is the central mechanism for handling asynchronous operations in Node.js. It continuously monitors for events (like I/O completion, timers, or network activity) and dispatches them to their corresponding callback functions.

1. Asynchronous I/O:

It provides a consistent, cross-platform API for performing various asynchronous I/O operations, including:

a) File system operations (reading, writing, etc.)
b) Networking (sockets, TCP/UDP)
c) DNS resolution
d) Timers
e) Child process management

2. Thread Pool:

For operations that are inherently blocking at the operating system level (like some file system operations or computationally intensive tasks), Libuv utilizes a thread pool to offload these tasks. This ensures that the main event loop, which is single-threaded, remains non-blocked and responsive to other events.

3. Cross-Platform Abstraction:

Libuv abstracts away the complexities and differences of various operating system-specific I/O mechanisms (e.g., epoll on Linux, kqueue on macOS, IOCP on Windows), providing a unified and consistent interface for developers.

In essence, Libuv is the engine that enables Node.js to achieve its highly efficient, non-blocking, and event-driven architecture, making it suitable for building scalable and high-performance applications.

Thread Pool

In Node.js, the thread pool is a collection of pre-initialized worker threads, managed by the libuv library, that handle computationally intensive or blocking I/O operations. While Node.js itself runs on a single-threaded event loop, the thread pool allows it to offload tasks that would otherwise block the main thread, ensuring non-blocking behavior and responsiveness.

Main thread:
"Hey libuv, I need to read file contents but that is a time consuming task. | don't want to block further code from being executed during this time. Can I offload this task to you?"

Libuv:
"Sure, main thread. Unlike you, who is single threaded, | have a pool of threads that I can use to run some of these time consuming tasks. When the task is done, the file contents are retrieved and the associated callback function can be run."

Network I/O operations on node.js runs on the main thread. it's handled by the main thread using non-blocking I/O, which is what makes Node.js fast and efficient.

Yes, node.js spawns four threads in addition to the main thread but none of them are used for network I/O such as database operations. The threads are:

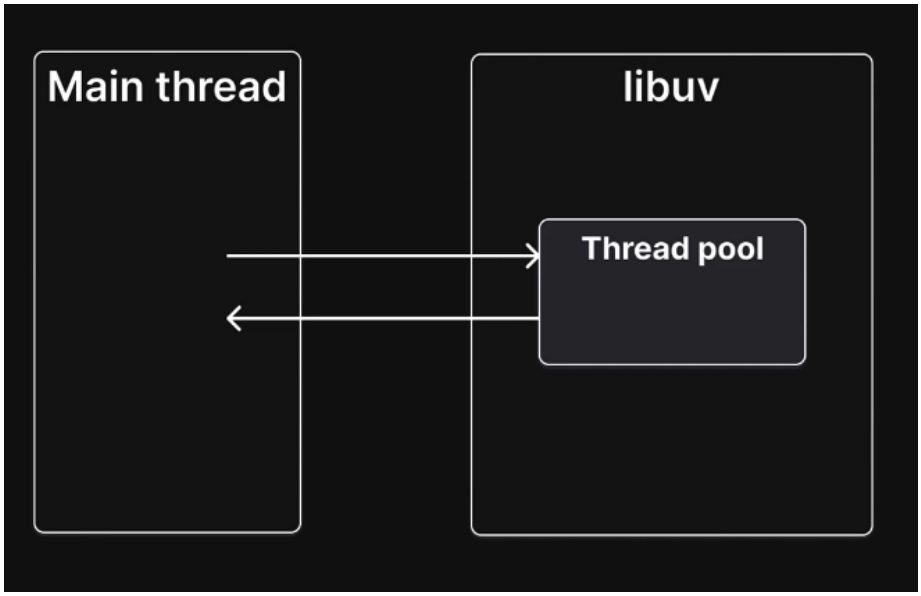
1. DNS resolver (since some OSes don't support async DNS natively)
2. File system API (because this is messy to do asynchronously cross-platform, reading/writing files)
3. Crypto (Because this uses the CPU, like password hashing)
4. Zlib (zlib operations like zipping files)

Managed by libuv:
The thread pool is not part of Node.js's core JavaScript engine (V8) but rather a feature provided by libuv, a multi-platform asynchronous I/O library that Node.js relies on.

Offloading Heavy Tasks:
It's used for operations that involve significant CPU usage or blocking I/O, such as:
File system operations (e.g., fs.readFile, fs.writeFile)
DNS lookups (e.g., dns.lookup)
Cryptographic functions (e.g., crypto.pbkdf2)
Compression/decompression operations (e.g., zlib.gzip)

Worker Threads vs. Thread Pool:
While the thread pool handles internal Node.js operations, the worker_threads module provides a way for developers to create and manage their own JavaScript worker threads for parallel execution of custom CPU-intensive tasks.

Ensuring Responsiveness:
By offloading these demanding tasks to the thread pool, the main Node.js event loop remains free to process other incoming requests and events, preventing delays and ensuring the application remains responsive.



Libuv's thread pool, as name indicates, its literally a pool of thread that node js uses to offload time consuming tasks and insure that the main thread is not blocked

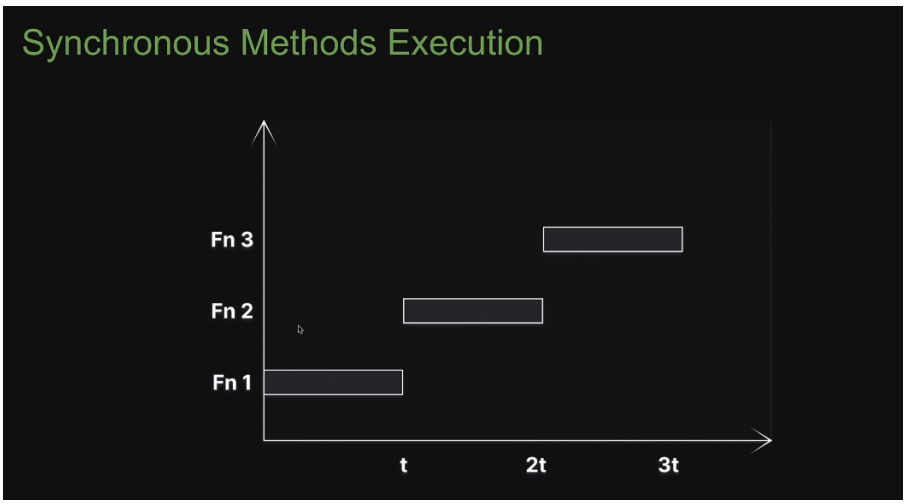
Note : Every method in node.js that has the "sync" suffix always runs on the main thread and is blocking

A few async methods like fs.readFile and crypto.pbkdf2 run on a seperate thread in libuv's thread pool. They do run synchronously in their own thread but as far as the main thread is concerned, it appears as if the method is running asynchronously

Eg : **Imagine a scenario where we need to run 3 tasks, each taking 1 sec**

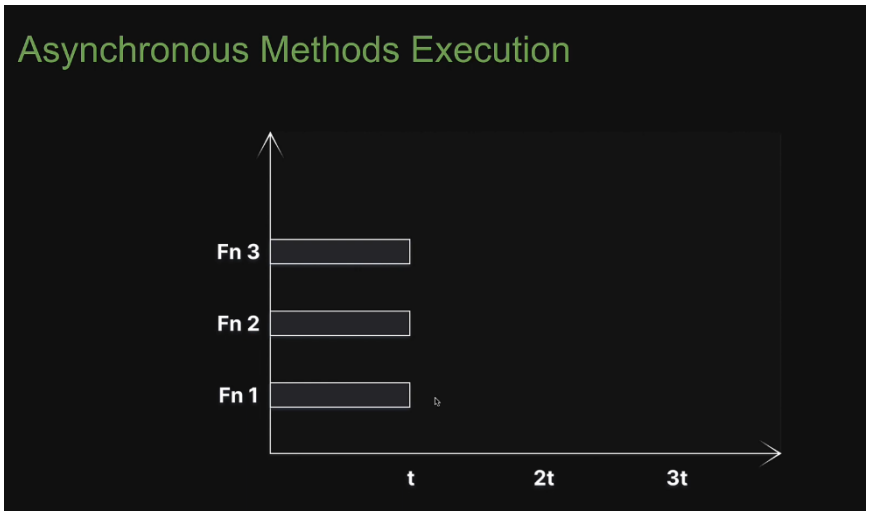
1. If we run synchronously : all 3 tasks will run in main thread simultaneously one by one. Because it only has single thread, all 3 will run in the same thread and next one will wait for the previous one to complete, eventually taking 3 seconds to run the whole process. (not in libuv's thread pool)
ex :

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

const start = Date.now();

const syncCalc = () => {
  crypto.pbkdf2Sync("password", "salt", 100000, 512, "sha512"); // sync version
  console.log("Took", Date.now() - start, "milliseconds to calculate");
}

syncCalc();
syncCalc();
syncCalc();

output :
Took 529 milliseconds to calculate
Took 1060 milliseconds to calculate
Took 1596 milliseconds to calculate
```



2. If we run asynchronously : main thread will offboard all 3 tasks to libuv's thread pool and will be run by seperate threads (3 thread for 3 tasks) in libuv's thread pool and the whole process will complete in 1 sec.
ex :

```
1. const crypto = require("crypto");
2.
3. const start = Date.now();
4.
5. const asyncCalc = () => {
6.   crypto.pbkdf2("password", "salt", 100000, 512, "sha512",()=>{  // async version
7.   console.log("Took", Date.now() - start, "milliseconds to calculate");
8.   });
9. }
10.
11. asyncCalc();
12. asyncCalc();
13. asyncCalc();
14.
15. output :
16. Took 535 milliseconds to calculate
17. Took 539 milliseconds to calculate
18. Took 540 milliseconds to calculate
```



What If the Number of Tasks > Number of Threads?
When more tasks are submitted than threads available, the next task has to wait for any thread to get free. Basically, it turns into synchronous process when thread reaches its limit:

1. The tasks go into a queue.
2. libuv assigns the next available thread to the next task in the queue.
3. Tasks are executed in FIFO order (generally).
4. Once a thread finishes a task, it picks the next task from the queue.

So tasks don’t get dropped, they’re just queued until a thread is available.

Thread Pool Size :
The thread pool typically has a default size of 4 threads, but this can be configured by setting the UV_THREADPOOL_SIZE environment variable
in the program :

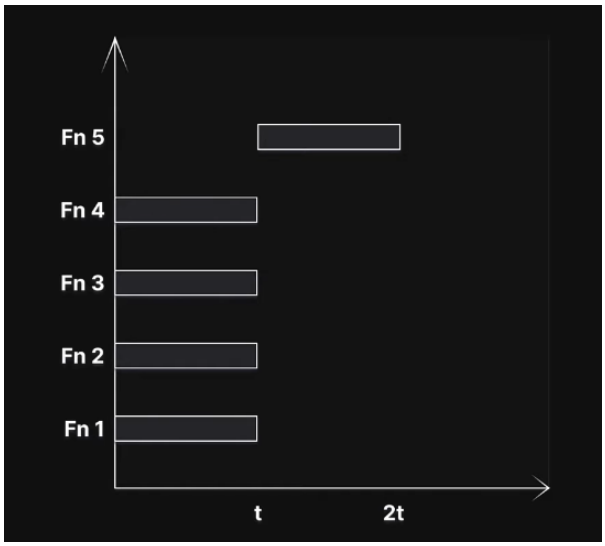
```
1. process.env.UV_THREADPOOL_SIZE = 8
```

or before the Node.js application starts :

```
1. UV_THREADPOOL_SIZE=8 node index.js
```

The maximum value for UV_THREADPOOL_SIZE in Node.js is 128.
While some sources might mention 1024, the hardcoded limit within libuv, which Node.js uses for its thread pool, is 128. Attempting to set UV_THREADPOOL_SIZE higher than 128 will result in it being capped at 128.
It is generally recommended to set UV_THREADPOOL_SIZE to the number of logical cores on the machine where the Node.js application is running, as setting it significantly higher than the available cores can lead to diminished performance due to increased context switching overhead.

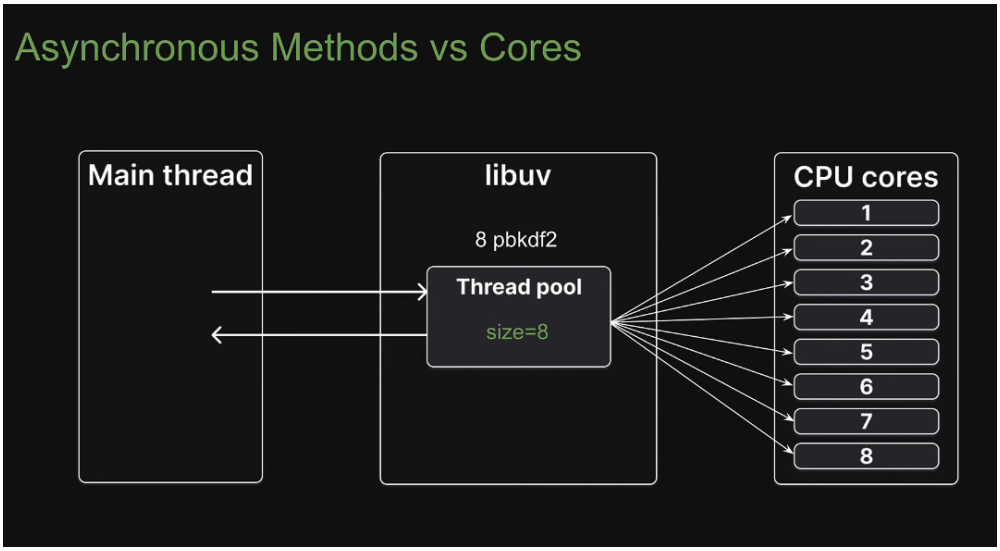
in the previous program, we called asyncCalc() 3 times (becuase of 4 default thread, those calls didn't cause any issue and all process took almost same time). But if we tried to call asyncCalc() 5 times, the 5th call will wait for any thread to get free and will take twice the amount of time (5th task become synchronous)
Tip : increase UV_THREADPOOL_SIZE to run as many task asynchronously and improve the total time taken to run multiple tasks



When Tasks/ThreadsPoolSize surpasses the amount of cores :
If we defined UV_THREADPOOL_SIZE to more than number of cpu cores we have, it take double the amount of time to all tasks to complete.
Increasing the thread pool size can help with performance but that is limited by the number of available CPU cores

Scanario 1 : Imagine we have 6 cpu cores and we defined UV_THREADPOOL_SIZE to 6 and run 6 tasks, the time taken for all 6 tasks will be :

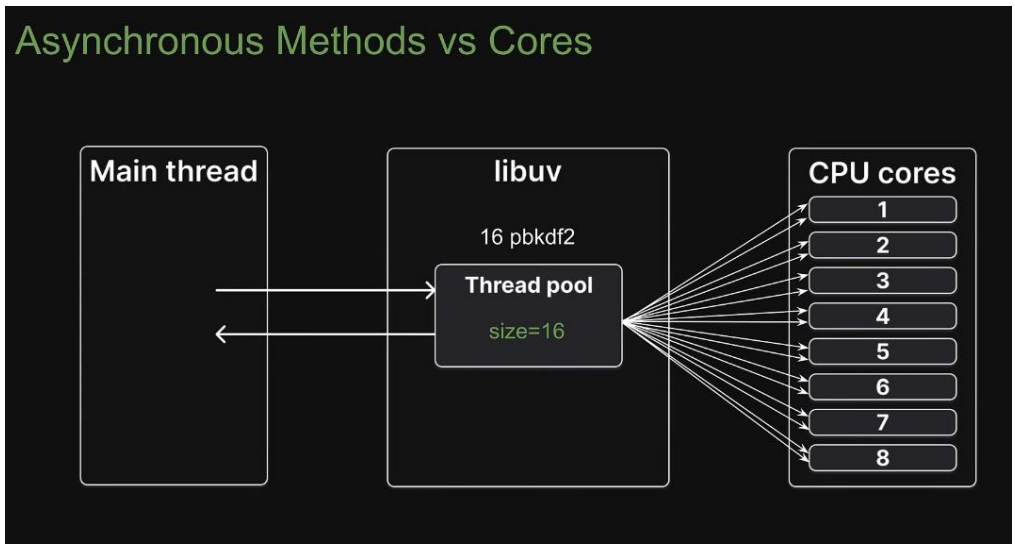
```
1. Task 4 Took 560 milliseconds to calculate
2. Task 1 Took 563 milliseconds to calculate
3. Task 5 Took 570 milliseconds to calculate
4. Task 2 Took 572 milliseconds to calculate
5. Task 3 Took 577 milliseconds to calculate
6. Task 6 Took 581 milliseconds to calculate
```



Scanario 2 : If we try to run 7 tasks while defining UV_THREADPOOL_SIZE to 7 (without defining this, the process will become synchronous once number of tasks reaches the number of default cpu cores (4)), the 6th and 7th task will take more time it took to complete the rest of the tasks. Because we only have 6 cores, Operating System has to provide 2 tasks (any two) to a single thread to juggle 7 tasks to 6 cores. that's why both tasks took more time than other tasks to complete.
Note : sometimes the 2 cores are left ideal by OS and only rest of cores are used even if we defined all cores to use using UV_THREADPOOL_SIZE

```
1. const crypto = require("crypto");
2.
3. const start = Date.now();
4.
5. process.env.UV_THREADPOOL_SIZE = 7;
6.
7. const asyncCalc = (i) => {
8.   crypto.pbkdf2("password", "salt", 100000, 512, "sha512",()=>{
9.   console.log(`Task ${i} Took`, Date.now() - start, "milliseconds to calculate");
10.   });
11. }
12.
13. for(let i = 1; i<=7; i++){
14.   asyncCalc(i);
15. }
16.
17. output :
18. Task 4 Took 560 milliseconds to calculate
19. Task 7 Took 563 milliseconds to calculate
20. Task 1 Took 570 milliseconds to calculate
21. Task 5 Took 572 milliseconds to calculate
22. Task 2 Took 577 milliseconds to calculate
23. Task 3 Took 779 milliseconds to calculate // took more time than rest
24. Task 6 Took 784 milliseconds to calculate // took more time than rest
```

If we try to complete 16 tasks while defining UV_THREADPOOL_SIZE to 16 in a 8 cores cpu



Thread Pool vs OS async mechanism

OS asyc mechanism :
Node.js leverages operating system (OS) asynchronous mechanisms to achieve its non-blocking, event-driven architecture, particularly for I/O operations. This allows Node.js to handle a large number of concurrent connections efficiently without creating a new thread for each, unlike traditional thread-per-connection models.
The core of this mechanism involves:

1. Native OS Asynchronous I/O APIs:
   For network I/O and timers, Node.js directly utilizes the asynchronous I/O interfaces provided by the underlying operating system. Examples include epoll on Linux, kqueue on macOS/FreeBSD, and IOCP (I/O Completion Ports) on Windows. These APIs allow the OS to handle the I/O operations in the background and notify Node.js when they are complete, without blocking the main event loop.
2. Thread Pool (for certain operations):
   For tasks that the OS might not provide native asynchronous APIs for, such as file system operations or CPU-bound tasks, Node.js utilizes a thread pool managed by libuv. libuv is a C library that provides cross-platform asynchronous I/O. When a file system operation is initiated, it's offloaded to a thread in this pool, allowing the main Node.js event loop to remain non-blocked and continue processing other events. Once the thread completes the operation, it notifies the event loop, which then processes the result.

This combination of native OS mechanisms and a thread pool enables Node.js to perform I/O operations asynchronously, leading to high scalability and responsiveness, which are key characteristics of its performance.

Nodejs utilize OS async mechanism instead of using thread pool if that specific mechanism is provided by OS

Node.js network requests do not primarily utilize the libuv thread pool because they are handled using native asynchronous I/O mechanisms provided by the operating system.
Here's why:

1. Event-Driven, Non-Blocking I/O:
   Node.js's core strength lies in its event-driven, non-blocking I/O model. For network operations, this means that instead of dedicating a thread to each connection and waiting for data, Node.js registers callbacks with the operating system. When network data is ready, the OS notifies Node.js, and the corresponding callback is placed in the event queue to be processed by the single-threaded event loop. This allows Node.js to handle a large number of concurrent connections efficiently without the overhead of creating and managing a thread for each.
2. Native Asynchronous APIs:
   Modern operating systems offer highly optimized, non-blocking APIs for network operations (e.g., epoll on Linux, kqueue on macOS/FreeBSD, I/O Completion Ports on Windows). Libuv, the underlying C++ library that powers Node.js's asynchronous I/O, leverages these native APIs directly for network requests.
3. Thread Pool for Blocking Operations:

The libuv thread pool is primarily used for operations that do not have native asynchronous APIs or are inherently CPU-bound and could block the main event loop if executed directly. Examples include:

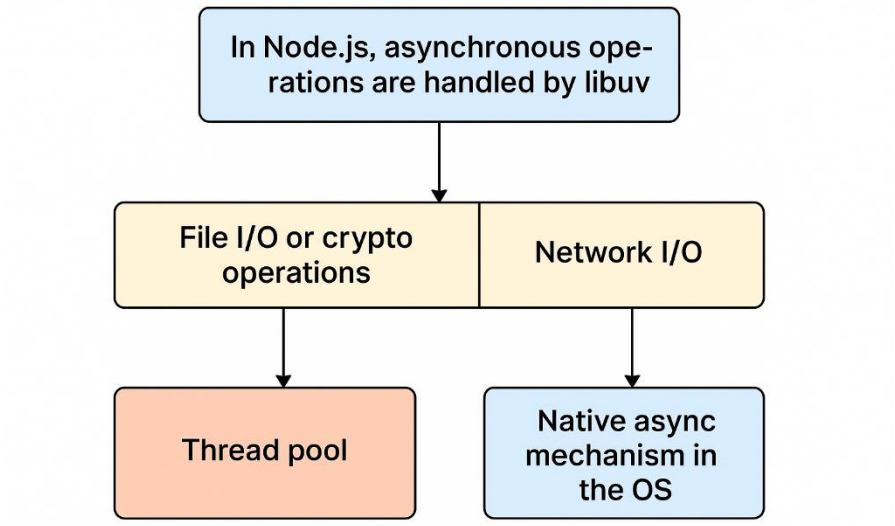
a) File System Operations: Many file system operations are synchronous and blocking at the OS level, so they are offloaded to the thread pool to prevent blocking the main thread.

b) DNS Resolution (default lookup): While some DNS lookups can be asynchronous, the default system-level DNS resolution often involves blocking calls, thus utilizing the thread pool.

c) Crypto and Zlib Operations: These are CPU-intensive tasks that are better handled in separate threads to avoid blocking the event loop.

In essence, Node.js prioritizes the use of native, non-blocking I/O for network requests to maximize concurrency and scalability, reserving the thread pool for tasks that genuinely require offloading to avoid blocking the single-threaded JavaScript execution.

```
1. const https = require("https");
2.
3. const start = Date.now();
4.
5. const asyncCall = (i) => {
6.   https.request("https://www.google.com",(res)=>{
7.     res.on("data",()=>{})
8.     res.on("end",()=>{
9.       console.log(`Request ${i} took ${Date.now()-start}`);
10.    });
11.  })
12.  .end();
13. }
14.
15. for(let i = 1; i<=12; i++){ // 12 call will not utilize thread pool and all calls will take approx same time
16.   asyncCall(i);
17. }
18.
19. output :
20. Request 9 took 620
21. Request 2 took 675
22. Request 12 took 745
23. Request 4 took 747
24. Request 3 took 749
25. Request 10 took 749
26. Request 1 took 750
27. Request 7 took 763
28. Request 11 took 830
29. Request 5 took 852
30. Request 6 took 860
31. Request 8 took 862
```



https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/ThreadPool%26OSasyncMechanism.png

Modules

A module is an encapsulated and reusable chunk of code that has its own context
In Node.js, each file is treated as a separate module
Types of Modules

1. Local modules - Modules that we create in our application
2. Built-in modules - Modules that Node.js ships with out of the box
3. Third party modules - Modules written by other developers that we can use in our application

CommonJS

CommonJS is a standard that states how a module should be structured and shared. Node.js adopted CommonJS when it started out and is what you will see in code bases. Each file is treated as a module
Variables, functions, classes, etc. are not accessible to other files by default.
Explicitly tell the module system which parts of your code should be exported via module.exports or exports
To import code into a file, use the require() function

ES Modules

At the time Node.js was created, there was no built-in module system in JavaScript
Node.js defaulted to CommonJS as its module system
As of ES2015, JavaScript does have a standardized module system as part of the language itself
That module system is called EcmaScript Modules or ES Modules or ESM for short
ES Modules is the ECMAScript standard for modules. It was introduced with ES2015. Node. js 14 and above support ES Modules.
Instead of module.exports, we use the export keyword. The export can be default or named. We import the exported variables or functions using the import keyword

Import Export Patterns (CommonJS)

⭐ Pattern 1 : Exporting an Object All at Once

```
math.js
const add = (a,b)=>{
    return a + b;
}
const subtract = (a,b)=>{
    return a - b;
}
module.exports = {add,subtract}  // short for -> module.exports = {add:add,subtract:subtract}

index.js
const module = require("./math.js")
console.log(module.add(1,2));
console.log(module.subtract(4,7));
```

⭐ Pattern 2 : Attaching Properties Directly to module.exports

```
math.js
module.exports.add = (a,b)=>{
    return a + b;
}
module.exports.subtract = (a,b)=>{
    return a - b;
}

index.js
const module = require("./math.js")
console.log(module.add(1,2));
console.log(module.subtract(4,7));
```

exports.add will work too because if you remember, module wrapper iife has a 'exports' parameter too but using exports.add without using precautions will break the reference to that object and throw error, so it is not used.
Full explanation : https://www.youtube.com/watch?v=ghUIlSNRru0

⭐ Pattern 3 : Exporting single instances

```
math.js
module.exports = (a,b)=>{
  return a + b;
}

index.js
const x = require("./math");
console.log(x(2,3)); // Output : 5
```

Import Export Patterns (ECMAScript Module aka ESModules)

⭐ Pattern 1 : (default Exports)

```
math.mjs
const add = (a, b) => {  // export default = (a, b) => {  <- this will work too
  return a + b;
};
export default add; // there can only be one export marked as `default`.

index.mjs
import add from “./math.mjs”;
console.log(add(5, 5));
```

⭐ Pattern 2 : (default exports more than one variable/function)

```
math.mjs
const add = (a, b)=>{
   return a + b;
}
const subtract = (a, b)=>{
   return a - b;
}
export default {add, subtract}

index.mjs
import math from "./math.mjs"; // ❌ import {add, subtract} from "./math.mjs";
console.log(math.add(5, 5));   // above won't work because it is used for **named exports**, not default exports
console.log(math.subtract(5, 5));
```

⭐ Pattern 3 : (named exports)

```
math.mjs
export const add = (a, b)=>{
   return a + b;
}
export const subtract = (a, b)=>{
   return a - b;
}
// we can also include default export with named exports but there should be only one default export

index.mjs
import * as maths from "./math.mjs"; // import {add, subtract} from "./math.mjs"; <- this will work too
console.log(maths.add(5, 5));
console.log(maths.subtract(5, 5));
```

Note :
If it is a default export, we can assign any name while importing
If it is a named export, the import name must be the same

Module Scope

Before a module's code is executed, Node.js will wrap it with a function wrapper (IIFE) that provides module scope
This saves us from having to worry about conflicting variables or functions
There is proper encapsulation and reusability is unaffected

```
1. (function(){
2.   // Module code actually lives in here
3. })
```

Each loaded module in Node.js is wrapped with an IIFE that provides private scoping of code

Module Wrapper & Module-scoped variables

Every module in node.js gets wrapped in an IIFE before being loaded
IIFE helps keep top-level variables scoped to the module rather than the global object
The IIFE that wraps every module contains 5 parameters which are pretty important for the functioning of a module
This wrapper function ensures that myVariable is private to myModule.js and doesn't conflict with other module variable if they are of the same name and that myFunction is exposed through the exports object.

```
1. myModule.js
2. (function(message) {
3.   const superHero = "Batman";
4.   console.log(message, superHero);
5. })("Hello");
```

syntax :

```
1. (function(exports, require, module, __filename, __dirname){
2.     // Module code goes here
3. })()
```

Note : we don't have to wrap the module in iife(the above is just the explanation of how node wrap the module with iife), the node does this automatically. If we try to print these parameters in the iife, we will get undefined because we have overwrote the default wrapper and these have become iife function's local variables. Try to print all these variables without iife and you will get their appropriate values.

1. exports : A reference to `module.exports`. It's used to export values from a module.
2. require : A function to import other modules (CommonJS).
3. module : Represents the current module, with info like `module.exports`, `id`, etc.
4. __filename : The full path to the current file (`index.js` in your case).
5. __dirname : The full path to the directory containing the current file.

These are automatically injected by Node into every file you run.

Module Caching

When you require() a module in Node.js, that module is loaded only once and then cached.
The first time a module is loaded, Node executes it and stores the result in memory.
Any subsequent require() calls to the same module in any other file/module return the cached version, not a re-executed version.
This is why changes in module state (like changing variables) are reflected in other files that require it afterward — because they share the same cached instance.
Example :

```
1. myModule.js
2. let x = 1;
3. function setValue() {
```

```
    x = 2;
}
function getValue() {
    return x;
}

file1.js
const {x,setValue,getValue} = require("./myModule"); // First time `myModule` is required -> it's executed and cached
console.log(x);    // Logs 1 (copied value at export)
setValue();
console.log(getValue()); // Logs 2 because it calls the function that accesses internal `x`
console.log(x); // this x will print the snapshot of the x instead of the actual x which was changed after calling setValue() function, that's why above function getValue() is used to get the updated value.
// ⚠ x is exported as a value in myModule.js, so its initial value (1) is copied. But setValue and getValue are functions, and they maintain a live reference to the x inside the module.

file2.js
const {x,getValue} = require("./myModule");
// `myModule` is NOT re-executed. It is loaded from cache!
// Therefore, `x` inside the module is already set to 2 (by `file1`)
console.log(getValue());
module.exports = {x,setValue,getValue};

index.js
console.log('file1 :-');
require("./file1");        // Loads file1, which loads `myModule`, sets `x = 2`
console.log('file2 :-');
require("./file2");        // Loads file2, reuses cached `myModule`, sees `x = 2`

output :
1  // x from file1 (copied when imported)
2  // getValue after setValue updated `x`
2  // getValue from file2 uses shared module state

```

Watch Mode

Watch mode in Node. js allows you to automatically restart your server whenever you make changes to your code, streamlining the development process. This eliminates the need to manually restart the server after every code modification, saving you time and effort. Introduced as an experimental feature in Node

To turn on watch mode, run the file with below command :

```
>> node --watch index.js
```

Build in Modules

Node.js includes several built-in modules that provide core functionalities without needing external libraries. These modules are part of the Node.js binary distribution and are essential for various tasks.

They are also referred to as core modules. It is mandatory to import the module before you can use it

Some common module are :

1. path
2. events
3. fs
4. stream
5. http

If you are interested in the source code of these build-in modules, they are present in the lib folder of the codebase of nodejs https://github.com/nodejs/node/tree/main/lib

Path Modules

The path module provides utilities for working with file and directory paths

The path module in Node.js is a built-in module that provides utilities for working with file and directory paths. It offers a variety of functions to manipulate, normalize, and resolve paths, ensuring cross-platform compatibility.

Commonly Used Functions:

- path.join(): Joins path segments.
- path.resolve(): Resolves a sequence of paths to an absolute path.
- path.normalize(): Normalizes a path string.
- path.dirname(): Extracts the directory name.
- path.basename(): Extracts the file name (with or without extension).
- path.extname(): Extracts the file extension.
- path.isAbsolute(): Checks if a path is absolute.
- path.parse(): Parses a path string into an object.
- path.format(): Constructs a path string from an object.
- path.sep: Provides platform-specific separator.

Events Module

The events module in Node.js is a core module that provides a way to work with events. It allows you to create, emit, and listen for custom events in your application. This module is crucial for building asynchronous and event-driven applications.
An event is an action or an occurrence that has happened in our application that we can respond to
Using the events module, we can dispatch our own custom events and respond to those custom events in a non-blocking manner

Events Module - Scenario

- Let's say you're feeling hungry and head out to Dominos to have pizza
- At the counter, you place your order for a pizza
- When you place the order, the line cook sees the order on the screen and bakes the pizza for you
- Order being placed is the event
- Baking a pizza is a response to that event

Streams

https://nodejs.org/api/stream.html
https://www.youtube.com/watch?v=FZOEL5nByYg
A stream is a sequence of data that is being moved from one point to another overtime
Ex: a stream of data over the internet being moved from one computer to another
Ex: a stream of data being transferred from one file to another within the same computer

- Work with data in chunks instead of waiting for the entire data to be available at once
- Process streams of data in chunks as they arrive instead of waiting for the entire data to be available before processing

Ex: watching a video on YouTube

- The data arrives in chunks and you watch in chunks while the rest of the data arrives over time

Ex: transferring file contents from fileA to fileB

- The contents arrive in chunks and you transfer in chunks while the remaining contents arrive over time
- Prevents unnecessary data download and memory usage

If you're transferring file contents from fileA to fileB, you don't wait for entire fileA content to be saved in temporary memory before moving it into fileB
Instead, the content is transferred in chunks over time which prevents unnecessary memory usage
Stream is infact a built-in node module that inherits from the event emitter class

We rarely use Streams direclty, other modules internally use stream for their functioning

```
1. const fs = require("node:fs");
2.
```

```
const readableStream = fs.createReadStream("./file1.txt", {  // readable stream to read data in chunks from file1.txt
    encoding: "utf-8"
});

const writeableStream = fs.createWriteStream("./file2.txt"); // writable stream to write data in chunks to file2.txt

readableStream.on("data", (chunk) => {
      console.log(chunk) ;
      writeableStream.write(chunk);
})

output : Hello Avinash
```

streams extend from event emitter class. Event emitter allow us to add listeners to events. above, the readableStream emmits the "data" event on which we can listen and we specify a callback function that should be executed on the data event.
The "data" event is emitted by readable streams (like fs.createReadStream) whenever a chunk of data becomes available to read.

Why Use Stream Instead of fs.readFile/fs.writeFile? (https://www.youtube.com/watch?v=64LJJhT6Ybo)
Because:

- It reads large files in small chunks (not all at once)
- Uses less memory and is non-blocking
- Ideal for large files, streaming video/audio, etc.

The Buffer that Streams use has a default size of 64kb. We can modify it using "highWaterMark" key :

```
const readableStream = fs.createReadStream("./file1.txt", {
    encoding: "utf-8",
    highWaterMark:2  // read data in chunks of 2 bytes
});
```

The Event Emmiter readableStream.on will listen on every 2 byte of chunk so the output when printing chunk in the event will be :

```
He
ll
o
Av
in
as
h
```

Types of Streams

1. Readable streams from which data can be read. Ex: Reading from a file as readable stream
2. Writable streams to which we can write data. Ex: Writing to a file as writable stream
3. Duplex streams that are both Readable and Writable. Ex: Sockets as a duplex stream
4. Transform streams that can modify or transform the data as it is written and read Ex: File compression where you write compressed data and read de-compressed data to and from a file as a transform stream

Chunk Size :
The Buffer that Streams use has a default size of 64kb.
and default chunk size in a readable stream depends on the type of stream you're working with:

1. File Streams (fs.createReadStream)
   Default chunk size: 64 KB → which is 64 * 1024 = 65536 bytes
   This is defined internally by Node as the highWaterMark option.
   ```
   const readable = fs.createReadStream('file.txt');
   console.log(readable._readableState.highWaterMark); // 65536 (64 KB)
   ```
   You can override it like this:
   ```
   fs.createReadStream('file.txt', { highWaterMark: 16 * 1024 }); // 16 KB
   ```
2. Network Streams (e.g., HTTP)

Default chunk size: 16 KB (16384 bytes)
Example: when reading from a TCP socket or HTTP request/response stream

NOTE : HTTP request is a readable stream and HTTP response is a writeable stream
https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/HTTP-request-response-streams.png
HTTP Streaming when querying large data from a database. Providing data in chunks to frontend :
https://www.loginradius.com/blog/engineering/guest-post/http-streaming-with-nodejs-and-fetch-api

Pipes

https://www.youtube.com/watch?v=5Mosdd3jwgU
It is used for large data, real-time processing, or when chaining stream operations.
The pipe() method on a readable stream sets up a mechanism to automatically pause and resume the readable stream based on the writable stream's capacity to handle data, preventing overruns. It also manages errors and stream completion.
It connects a readable stream to a writable stream, allowing data to flow automatically between them — chunk by chunk — without manually handling 'data' and 'end' events. .pipe() is a method available on readable streams in Node.js.
We cannot use .pipe() method on writable streams because it is meant to receive data, not send it. You can pipe data into a writable stream from a readable stream. You can't pipe data from a writable stream.
Readable streams : These streams emit data, and pipe() is used to connect them to writable streams for consumption.
Writable streams : These streams accept data and write it to a destination. They don't emit data to be piped elsewhere.
Benefits of .pipe()

1. Simplicity : Shorter and cleaner code
2. Backpressure : Automatically manages flow so memory doesn't overflow
3. Error handling : Can chain .on('error') listeners
4. Streaming efficiency : Great for large data (files, HTTP responses, etc.)

Pipe Chaining :
It returns destination streams which enable chaining, but the conditions is that stream has to be readable, duplex or a transform stream
Pipe chaining in Node.js refers to the process of connecting multiple streams together to create a data processing pipeline. This is achieved using the pipe() method, which is available on readable streams. The pipe() method takes a writable stream as an argument, and it automatically forwards data from the readable stream to the writable stream.
When chaining pipes, the output of one stream becomes the input of the next stream. This allows for complex data transformations to be performed in a sequential manner.
For example, consider the following scenario: reading data from a file, compressing it using gzip, and then writing the compressed data to another file. This can be achieved using pipe chaining as follows:

Pipe chaining offers several benefits:

- Modularity: It allows you to break down complex data processing tasks into smaller, manageable components.
- Readability: It makes code more readable and easier to understand by visually representing the flow of data.
- Efficiency: It avoids loading large amounts of data into memory at once, improving performance and memory usage.
- Flexibility: It allows you to easily combine different types of streams to perform various data transformations.

syntax :

```
readable.pipe(transform1).pipe(transform2).pipe(writable);
```

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

fs.createReadStream('input.txt')              // readable
  .pipe(zlib.createGzip())                    // transform (compress)
  .pipe(fs.createWriteStream('output.txt.gz')) // writable
```

Tip : What is this output.txt.gz file here ?
This is a Gzipped (compressed) version of your original file. This Compresses data using the Gzip algorithm. The compressed version of input.txt, smaller and optimized for storage/transmission
Use Cases:

- Reduce file size to save disk space or bandwidth.
- Send compressed files over HTTP (faster delivery).
- Store backups efficiently.
- Package logs or data for transmission.

It's not a text file anymore — it's a binary compressed file that needs to be decompressed to view its original content.
To view the content of the gz or de-compress it :
-> for windows - use winrar/7zip
-> for Linux/macOS terminal - gunzip output.txt.gz (gunzip is preinstalled in linux/mac)
-> In Node.js (to reverse the operation):

```
1. fs.createReadStream('output.txt.gz')
2.   .pipe(zlib.createGunzip())
3.   .pipe(fs.createWriteStream('original.txt'));
```

How .pipe() handle backpressure :
Backpressure occurs when a readable stream produces data faster than a writable stream can consume it. Imagine a scenarion where readable stream is 4mb/s and writable stream is 3mb/s. This incosistency of speed in both side will lead to potential memory issues and overwhelm the writable stream which can't handle this much data because of missmatch of the data speed.
Here's how pipe() fix this -

- Automatic Flow Control: The pipe() method automatically manages the flow of data between a readable and a writable stream. It pauses the readable stream when the writable stream's buffer is full, preventing data overload.
- 'drain' event: When the writable stream is ready to receive more data, it emits a 'drain' event. The pipe() method listens for this event and resumes the readable stream, ensuring a smooth data flow.
- Simplified Backpressure Management: By using pipe(), you don't have to manually manage pausing and resuming streams. Node.js handles this automatically, making your code cleaner and less error-prone.
- Error Handling: pipe() also manages errors. If an error occurs in either the readable or writable stream, it propagates the error to the other stream, preventing data loss.

```
1. const readableStream = fs.createReadStream("./file1.txt", {
2.    encoding: "utf-8"
3. });
4. const writeableStream = fs.createWriteStream("./file2.txt");
5. readableStream.pipe(writeableStream);
```

using .pipe() eliminates the need to manually use these EventEmitter (but you can still use these while still utilizing .pipe()) :

- .on('data', handler)
- .on('end', handler)
- .on('error', handler)

Pipe and Streams
Streams and pipes are closely related in Node.js.

- Streams represent a sequence of data that can be read or written over time. They are a fundamental concept for handling data efficiently, especially when dealing with large files or network communication. Node.js offers various stream types, including readable, writable, and transform streams.
- Pipes provide a mechanism to connect readable streams to writable streams. The pipe() method takes a readable stream and redirects its output to a writable stream. This allows for a streamlined data flow, where data chunks are automatically passed from one stream to another as they become available.

Buffers

Ex: Imaginge a scenario of rollar coaster with 30 Seating capacity

1. Scenario 1
   100 - People arrival
   30 - People accommodated
   70 - People in queue (waiting)
2. Scenario 2
   1 - Person arrives (waiting)
   Wait for at least 10

So the Buffer is the area where people wait

- Node.js cannot control the pace at which data arrives in the stream. It can only decide when is the right time to send the data for processing.
- If there is data already processed or too little data to process, Node puts the arriving data in a buffer
- It is an intentionally small area that Node maintains in the runtime to process a stream of

Ex: streaming a video online

1. If your internet connection is fast enough, the speed of the stream will be fast enough to instantly fill up the buffer and send it out for processing. That will repeat till the stream is finished
2. If your connection is slow, after processing the first chunk of data that arrived, the video player will display a loading spinner which indicates it is waiting for more data to arrive
3. Once the buffer is filled up and the data is processed, the video player shows the video While the video is playing, more data will continue to arrive and wait in the buffer

In Node.js, a Buffer is a built-in class used to handle binary data directly in memory. It's particularly useful when dealing with:

- File system operations (e.g., reading images, PDFs, etc.)
- Network protocols (e.g., TCP streams, HTTP responses)
- Binary streams (e.g., audio, video, or encrypted content)

Think of a Buffer as a raw storage area in memory — similar to an array of bytes.

JavaScript was originally created to work in the browser and mainly with strings and objects, not binary data. But Node.js is used on the server, where low-level operations like file I/O, TCP sockets, and cryptography are common — all of which involve binary data.
So Buffer fills that gap by letting JavaScript work with binary data efficiently.

🔧 Common Use Cases

1. File system reading -> Reading a .jpg or .zip file using fs.readFile
2. Networking -> Processing TCP or WebSocket data chunks
3. Encoding conversions -> Converting between UTF-8, ASCII, Base64, etc.
4. Cryptography -> Hashing or encrypting raw bytes using crypto module
5. Streaming -> Handling chunks of audio/video data

Nodejs internally uses Buffers whenever required

Ex:

```
const buffer = new Buffer.from("Avinash"); // the second argument utf-8 is optional becuase it is already utf-8 by-default new Buffer.from("Avinash","utf-8");
console.log('buffer -> ',buffer);
console.log('toJSON -> ',buffer.toJSON());
console.log('toString -> ',buffer.toString());
buffer.write("Ram");
console.log('toString -> ',buffer.toString());
output :
buffer ->  <Buffer 41 76 69 6e 61 73 68> //raw binary data of each character that is displayed in Hexadecimal.eg 41=101001
toJSON ->  {
  type: 'Buffer',
```

```
  data: [
     65, 118, 105, // These are the ASCII(or UTF-8) representation of each character in "Avinash" as a array
    110,  97, 115,
    104
  ]
}
toString ->  Avinash
toString ->  Ramnash

```

Explanation of above code and a Simple program to show the use of Buffer in real case scenario :
https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/Buffer.png

Character Encodings : (Unicode, ASCII, UTF-8 etc) : https://www.youtube.com/watch?v=ut74oHojxqo

1. Asynchronous JavaScript : JavaScript is a synchronous, blocking, single-threaded language
2. Synchronous: If we have two functions which log messages to the console, code executes top down, with only one line executing at any given time
3. Blocking : No matter how long a previous process takes, the subsequent processes won't kick off until the former is completed
4. Web app runs in a browser and it executes an intensive chunk of code without returning control to the browser, the browser can appear to be frozen
5. Single-threaded : A thread is simply a process that your javascript program can use to run a task
6. Each thread can only do one task at a time. JavaScript has just the one thread called the main thread for executing any code

```
function A() {
  console. log("A")
}

function B() {
  console. log("B")
}

A()
B()
Output :
-> Logs A and then B
```

Problem with synchronous, blocking, single-threaded model of JavaScript

```
let response = fetchDataFromDB('endpoint')
displayDataFromDB(response)
```

fetchDataFromDB('endpoint') could take 1 second or even more. During that time, we can't run any further code
JavaScript, if it simply proceeds to the next line without waiting, we have an error because data is not what we expect it to be

Just JavaScript is not enough
We need new pieces which are outside of JavaScript to help us write asynchronous code For front-end, this is where web browsers come into play.
For back-end, this is where Node.js comes into play
Web browsers and Node.js define functions and APIs that allow us to register functions that should not be executed synchronously, and should instead be invoked asynchronously when some kind of event occurs

For example, that could be
1. the passage of time (setTimeout or setinterval)
2. the user's interaction with the mouse (addEventListener)
3. data being read from a file system or the arrival of data over the network (callbacks, Promises, async-await)

You can let your code do several things at the same time without stopping or blocking your main thread

JavaScript is a synchronous, blocking, single-threaded language, This nature however is not beneficial for writing apps. We want non-blocking asynchronous behaviour which is made possible by a browser for frontend and Node.js for backend
This style of programming where we don't block the main JavaScript thread is fundamental to Node.js and is at the heart of how some of the built-in module code is written

fs Module

The file system (fs) module allows you to work with the file system on your computer
Ex :

```
file.txt
Hello Avinash
index.js
const fs = require("fs");

const file_Buffer = fs.readFileSync("./file.txt")
const file_Content = fs.readFileSync("./file.txt","utf-8")
console.log(file_Buffer);
console.log(file_Content);

output :
<Buffer 48 65 6c 6c 6f 20 41 76 69 6e 61 73 68 0a>
Hello Avinash
```

Sync/Async file read

```
index.js
const fs = require("fs");

console.log("First");
const fileContent = fs.readFileSync("./file.txt","utf-8") // Synchronous file read
console.log(fileContent);

console.log("Second");

fs.readFile("./file.txt","utf-8",(error,data)=>{ // Asynchronous file read. This patter is using callbacks where first argument is the error, is called "error first callback pattern"
    if(error){
       console.log(error,"Error reading file");
     }
     else{
      console.log(data);
     }
});

console.log("Third");

output :
First
Hello Avinash

Second
Third
Hello Avinash
```

Becuase of this asynchronous file read, the app will not freeze when multiple users interact with the application

Create files

```
index.js
fs.writeFileSync("./greet1.txt","Hello Avinash");
fs.writeFile("./greet2.txt","Hello again Avinash",(err)=>{
   if(err) return console.log(err);
   console.log("Content has been written in the greet2.txt file");
});
```

The Above write function will overwrite the content if the file already exist. To append to the existing content, add a flag in the object :

```
1. fs.writeFile("./greet2.txt","Hello again Avinash",{flag:'a'},(err)=>{
```

fs Promise Module

The fs/promises API provides asynchronous file system methods that return promises.
The promise APIs use the underlying Node.js threadpool to perform file system operations off the event loop thread. These operations are not synchronized or threadsafe. Care must be taken when performing multiple concurrent modifications on the same file or data corruption may occur.

```
 1. const fs = require("fs");
 2.
 3. //handling with then-catch
 4. fs.promises.readFile("./file.txt","utf-8")
 5. .then((data)=>{
 6.  console.log(data);
 7. }).0
 8. .catch((err)=>{
 9.  console.log(err);
10. })
11.
12. //handling with async-await
13. async function readFile(){
14.   try{
15.     const data = await fs.promises.readFile("./file.txt","utf-8")
16.     console.log(data);
17.   }
18.   catch(err){
19.     console.log(err);
20.   }
21. }
22. readFile();
```

Tip : this works too :

```
1. const fs = require("fs/promises");
2. fs.readFile("./file.txt","utf-8") // will still return a promise
```

Note : The callback based version of the fs module api are preferable over the use of promise api when maximum performance is required both in terms of execution time and memory allocation

HTTP Module

How the web works
Computers connected to the internet are called clients and servers

Clients : are internet-connected devices such as computers or mobile phones along with web-accessing software available on those devices such as a web browser
Servers : on the other hand are computers that store web pages, sites, or apps

When you type a url in the browser, the client device request access to the webpage. A copy of the webpage is downloaded from the server and send as a response to the client to displayed in the browser. This Model is popularly called Client-Server Model
Now we understood, the data transfer between client & server, but in what format is that data. What if the request send by the client cannot be understood by the server, or what if the response send by the server cannot be understod by the client, well that's where the HTTP comes into the picture.

-> HTTP

1. Hypertext Transfer Protocol

2. A protocol that defines a format for clients and servers to speak to each other
3. The client sends an HTTP request and the server responds with an HTTP response

-> HTTP and Node

1. We can create a web server using Node.js
2. Node.js has access to operating system functionality like networking
3. Node has an event loop to run tasks asynchronously and is perfect for creating
4. web servers that can simultaneously handle large volumes of requests
5. The node server we create should still respect the HTTP format
6. The HTTP module allows creation of web servers that can transfer data over HTTP

```
1. app.get('/getContent', (req, res) => {
2.   res.writeHead(200, { "Content-Type": "text/html" });  // these (writeHead & end) are outdated old http methods. use express methods instead
3.   res.end("<i>Hello world</i>");
4. }).listen(3000);
```

if we specified Content-Type as "text/plain" then the response will be treated as a plain text and tag will be shown with text in the browser
if we specified Content-Type as "text/html" then the response will be treated as html as it will be shown as italic font instead of as plan text with tags in the browser

content-type headers in express and sending response :
https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/content-types.png

Sending a html file from backend :

```
1. app.get('/getFile',(req, res) => {
2.   res.writeHead(200, { "Content-Type": "text/html" });
3.   const html = fs.readFileSync("./index.html", "utf-8") // readFileSync (synchronous) because res.end should't run before reading the file
4. res.end(html);
5. })
```

If the html file is big, we can pipe it to make it more performent so user don't have to wait for whole file to load, instead it will load in chunks :

```
1. app.get('/getFile',(req, res) => {
2.   res.writeHead(200, { "Content-Type": "text/html" });
3.   fs.createReadStream("./index.html").pipe(res);
4. })
```

View (MVC-> V(view))

A view is simply the web page we see. A page that displays the text, images of a website. The case is that the webpage(html file) is served from backend with dynamic values instead of being created from frontend. If you build websites with HTML, then every page you create is the view like the homepage, about page, and the contact page.
https://medium.com/front-end-weekly/what-is-a-view-in-web-application-6a2836eed4eb
How to create a view : https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/View.png

Web Framework

A framework simply abstracts the lower level code allowing you to focus on the requirements than the code itself
For example, Angular, React, Vue are all framework/libraries that help you build user interfaces without having to rely on the lower level DOM API in JavaScript
There are frameworks to build web or mobile applications without having to rely on the HTTP module in node.js

Ex: express, nest, hapi, koa and sails
They build on the to of the HTTP module making it easier for you to implement all the features.

Event Loop

The event loop in JavaScript is a fundamental mechanism that enables asynchronous, non-blocking behavior despite JavaScript's single-threaded nature. It is the core component that manages how code execution, asynchronous operations, and user interactions are handled.
It is a C program and is part of libuv(only in nodejs event loop, not in frontend js). A design pattern that orchestrates or co-ordinates the execution of synchronous and asynchronous code in Node.js.
https://dev.to/rajatoberoi/understanding-the-event-loop-callback-queue-and-call-stack-in-javascript-1k7c

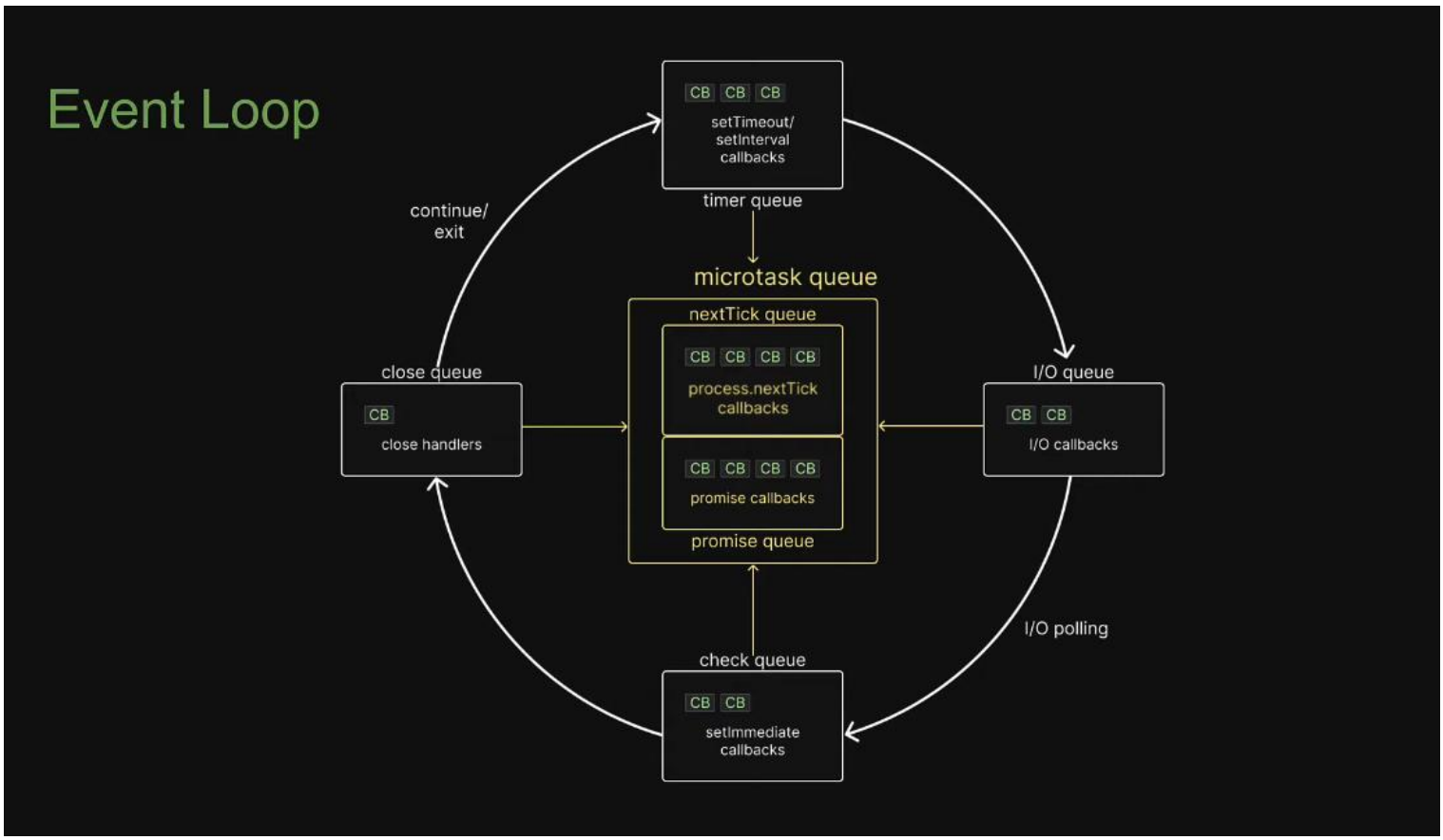
Here's how it works:

- Call Stack:
  JavaScript executes code in a single-threaded environment, meaning it processes one task at a time. Synchronous code is pushed onto the call stack and executed sequentially.
- Web APIs (Browser) / C++ APIs (Node.js):
  When an asynchronous operation, such as a setTimeout, fetch request, or an event listener, is encountered, it's offloaded to the browser's Web APIs (or Node.js's C++ APIs). These APIs handle the asynchronous task in the background.
- Callback Queue (Task Queue/Macrotask Queue):
  Once an asynchronous operation completes, its associated callback function is placed in the callback queue (also known as the task queue or macrotask queue).
- Microtask Queue:
  There's also a separate queue for microtasks, such as Promises and queueMicrotask. Microtasks have higher priority and are executed before macrotasks.
- The Event Loop:
  The event loop continuously monitors two things:
  1. The Call Stack: It checks if the call stack is empty.
  2. The Callback Queues: If the call stack is empty, it first checks the microtask queue. If there are microtasks, it moves them to the call stack for execution until the microtask queue is empty. Then, it checks the macrotask queue and, if there are tasks, moves one (or more, depending on the environment) to the call stack for execution.

This continuous cycle ensures that while long-running asynchronous operations are being handled in the background, the main thread remains responsive and can process other tasks, including user interactions, without blocking the entire application.

The Difference in Event Loop between JavaScript and Node.js
https://wearecommunity.io/communities/aep-js-community/articles/2893

Cohort 2.0 (Harkirat) => Week 2.1 – Revision of Async (Timestamp : 00:41:27) [The best explanation]
https://drive.google.com/file/d/1zDFu1_pVbOA4GL6Ow51qhB22pUCDocz-
https://www.youtube.com/watch?v=L18RHG2DwwA



Event loop is a loop that is alive as long as your application is up and running.
In every iteration of loop, we come across 6 different queues. Each queue hold one or more callback functions that need to be eventually executed on the callstack. And offcourse, the type of callback functions are different for each queue.

1. 1st we have timer queue, this contains callback associated with setTimeout & setInterval.
2. 2nd we have I/O queue, this contains callback associated with all the async methods eg. fs & https methods
3. 3rd we have check queue, this contains callback associated with a function called setImmediate. This function is specific to node and and is not something you would come across when writing javascript for the browser
4. 4th we have the close queue, this contains callbacks associated with the close event of an async task.
5. Finally we have Microtask queue at the center. This is actually 2 seperate queues. The 1st queue is called nextTick queue and contains callbacks associated with a function called process.nextTick, which is a Node.js-specific function. The 2nd queue is the promise queue which contains callbacks that are associated with the native promise in javascript.

One very important point to note is that (1)timer queue, (2)I/O queue, (3)check queue and (4)close queue are all part of libuv. The 2 microtasks are however not part of libuv. Hopefully they still the part of node runtime and play an important role in the order of execution of callbacks.

Event Loop - Execution Order

1. Any callbacks in the micro task queues are executed. First, tasks in the nextTick queue and only then tasks in the promise queue
2. All callbacks within the timer queue are executed
3. Callbacks in the micro task queues if present are executed. Again, first tasks in the nextTick queue and then tasks in the promise queue
4. All callbacks within the I/O queue are executed
5. Callbacks in the micro task queues if present are executed. nextTick queue followed by Promise queue.
6. All callbacks in the check queue are executed
7. Callbacks in the micro task queues if present are executed. Again, first tasks in the nextTick queue and then tasks in the promise queue
8. All callbacks in the close queue are executed

9. For one final time in the same loop, the micro task queues are executed. nextTick queue followed by promise queue.

If there are more callbacks to be processed, the loop is kept alive for one more run and the same steps are repeated
On the other hand, if all callbacks are executed and there is no more code to process, the event loop exits.
This is the role libuv's event loop plays in the execution of async code of nodejs

Node.js Event Loop Phases (Simplified)

1. Timers Phase — runs setTimeout, setInterval callbacks
2. Pending Callbacks Phase — handles system-level callbacks (not common)
3. Poll Phase — I/O callbacks are executed. This is a crucial phase where the event loop retrieves new I/O events (like network or file system events) and executes their corresponding callbacks. It can block here if there are no pending I/O events and no timers/immediate callbacks to process.
4. Check Phase — runs setImmediate callbacks
5. Close Callbacks Phase — e.g., socket.on('close')

Between each phase, microtasks run:
-> process.nextTick
-> Promise.then, queueMicrotask

Microtasks are Priority : (Imp)
Microtasks (nextTick and Promise queues) are executed in between each queue and between each callback. After each individual callback inside a macrotask, Node.js drain process.nextTick and Promise microtasks, before moving on to the next callback in that same macrotask queue.
This ensures microtasks are always given priority over the next macrotask — a behavior similar to how the browser event loop handles Promises.

Microtask vs Macrotask

microtasks and macrotasks represent different categories of asynchronous operations with distinct priorities and execution orders.

1. Macrotasks (Tasks):
- Macrotasks are the primary units of work processed by the event loop.
- They include larger, less time-sensitive operations like:
    - Timers (e.g., setTimeout, setInterval)
    - I/O operations (e.g., file system operations, network requests)
    - UI rendering (in browser environments, not directly applicable to Node.js's core event loop)
- The event loop processes one macrotask at a time. After executing a macrotask, it then checks and clears the microtask queue before proceeding to the next macrotask in the queue.

2. Microtasks:
- Microtasks are smaller, higher-priority asynchronous operations that are executed immediately after the current macrotask completes and before the next macrotask is picked up.
- They are designed for operations that need to be completed as quickly as possible without yielding to the event loop for a full cycle.
- Examples of microtasks include:
    - Promise callbacks (.then(), .catch(), .finally())
    - process.nextTick() (Node.js specific)
    - MutationObserver callbacks (in browser environments)
- The microtask queue is fully drained before the event loop moves on to the next macrotask. This means a microtask can enqueue other microtasks, and all of them will be executed before the next macrotask cycle.

Key Differences Summarized:

- Priority: Microtasks have a higher priority than macrotasks.
- Execution Order: Microtasks are executed after the current macrotask and before the next macrotask. Macrotasks are executed one at a time, with a full microtask queue clear between each.
- Use Cases: Macrotasks are suitable for longer-running, less urgent asynchronous operations, while microtasks are for immediate, high-priority asynchronous tasks that need to run in the same "tick" as the current operation.

Few Questions :

Que. Whenever an async task completes in libuv, at what point does Node decide to run the associated callback function on the call stack?
Ans. Callback functions are executed only when the call stack is empty. The normal flow of execution will not be interrupted to run a callback function

Que. What about async methods like setTimeout and setinterval which also delay the execution of a callback function?
Ans. setTimeout and setinterval callbacks are given first priority

Que. If two async tasks such as setTimeout and readFile complete at the same time, how does Node decide which callback function to run first on the call stack?
Ans. Timer callbacks are executed before I/O callbacks even if both are ready at the exact same time

How do we 'queue up' a callback function on each of these queues :

1. In Microtask Queue :

The microtask queue in the Node.js Event Loop fundamentally consists of two distinct queues:

1. nextTick queue:
   This queue holds callbacks scheduled using process.nextTick(). These callbacks are executed with the highest priority, even before any other microtasks or macrotasks are processed within the current phase of the event loop.
2. Promise queue:
   This queue contains callbacks associated with JavaScript Promises (e.g., then(), catch(), finally()). These callbacks are executed after process.nextTick() callbacks but before the event loop moves to the next phase (e.g., timers, I/O).

Therefore, while the term "microtask queue" is often used singularly, it encompasses these two separate, prioritized queues within the Node.js event loop's microtask handling mechanism.

1. nextTick queue :
to queue a callback function in nextTick queue, we use a build in process.nextTick function
syntax :

```
1. process.nextTick(()=>{
2.   console.log('this is process.nextTick');
3. })
```

process.nextTick() in Node.js is a function that schedules a callback to be executed immediately after the current operation on the call stack completes, but before the Node.js Event Loop proceeds to its next phase (e.g., timers, I/O, check).
Use nextTick() when you want to make sure that in the next event loop iteration that code is already executed.
Here's a breakdown of its key characteristics and uses:

- Immediate Execution: Callbacks passed to process.nextTick() are prioritized and executed as soon as the current JavaScript execution context finishes, even before any setTimeout() or setImmediate() callbacks are processed.
- Microtask Queue: process.nextTick() callbacks are placed in a special queue called the "nextTick queue" or "microtask queue," which is processed before other event loop phases.
- Non-Blocking for the Current Operation: While it executes quickly, it doesn't block the current operation from completing. It simply ensures its callback runs next.

Use Cases:

- API Development: It's often used in API design to allow users to attach event handlers after an object has been constructed but before any I/O operations have occurred, ensuring consistency.
- Error Handling: It can be used to defer error handling or cleanup operations, ensuring they run after the immediate task finishes.
- Preventing I/O Starvation: In some specific scenarios, it can help prevent I/O operations from being starved if a long-running synchronous task is followed by many I/O-dependent operations, by allowing critical tasks to be processed sooner.

While process.nextTick() offers high priority, excessive or recursive use of it can potentially "starve" the event loop, preventing other events (like I/O) from being processed, leading to performance issues. It should be used judiciously for specific, high-priority tasks.
Use of process.nextTick is discouraged as it can cause the rest of the event loop to starve
If you endlessly call process.nextTick. the control will never make it past the microtask queue

Two main reasons to use process.nextTick

1. To allow users to handle errors, cleanup any then unneeded resources, or perhaps try the request again before the event loop continues
2. To allow a callback to run after the call stack has unwound but before the event loop continues

2. Promise queue

Promise queue come into play when promises are resolved (using then-catch or async await)

```
1. Promise.resolve().then(()=>console.log("This is promise")); // or any api that returns a promise
```

- Experiment 1 : All user written synchronous JavaScript code takes priority over async code that the runtime would like to eventually execute

```
1. console.log("console.log 1");
2. process.nextTick(() => console.log("this is process.next 1"));
3. console.log("console.log 2");
4. output :
5. console.log 1
6. console.log 2
7. this is process.next 1
```

- Experiment 2 : All callbacks in nextTick queue are executed before callbacks in promise queue

```
1. Promise.resolve().then(()=>console.log("This is promise"));
2. process.nextTick(()=>console.log("This is nextTick"))
3. output :
4. This is nextTick
5. This is promise
```

- Experiment 3 : nested callbacks gets added at the end of the queue

```
 1. process.nextTick(() => console.log("this is process.nextTick 1"));
 2. process.nextTick(() => {
 3.   console.log("this is process.nextTick 2");
 4.      process.nextTick(() =>
 5.       console.log("this is the inner next tick inside next tick") //  even though it's nested, it’s queued after
 6.      );                                                           //  the current nextTick() callbacks finish.
 7. });
 8. process.nextTick(() => console.log("this is process.nextTick 3"));
 9. Promise.resolve().then(() => console.log("this is Promise.resolve 1"));
10. Promise.resolve().then(() => {
11.   console.log("this is Promise. resolve 2");
12.     process.nextTick(() =>
13.      console.log("this is the inner next tick inside Promise then block") // that nextTick() is not executed immediately
14.      );         // it gets pushed to the next round of the nextTickQueue. Because the control is still in the promise
15. });             // queue, it will complete all the promises first
16. Promise.resolve().then(() => console. log("this is Promise.resolve 3"));
17.
18. output :
19. this is process.nextTick 1
20. this is process.nextTick 2
21. this is process.nextTick 3
```

```
22. this is the inner next tick inside next tick
23. this is Promise.resolve 1
24. this is Promise. resolve 2
25. this is Promise.resolve 3
26. this is the inner next tick inside Promise then block
```

Que :
Why the nested process.nextTick inside the 2nd nextTick is queued for the next round. Why it couldn't execute right away inside the 2nd process.nextTick, while the control was inside the 2nd process.nextTick
Ans :
process.nextTick() doesn't recursively drain the queue during execution.
Node.js processes nextTick queue like this:

1. Collect all scheduled process.nextTick() calls into a queue.
2. Start executing them one by one in order of scheduling.
3. While one callback is executing, any new nextTick() call is added to the same queue, but after the currently executing batch finishes.

Node.js does not allow infinite recursion by immediately executing nested nextTick()s within the current tick — doing so would block the event loop forever.
Queue Execution Flow:

- [nextTick 1, nextTick 2, nextTick 3]
- During execution of nextTick 2, we schedule:
  [nextTick 1, nextTick 2, nextTick 3] // running
  ↑
  schedules → [inner nextTick]
- So the queue becomes:
  [nextTick 3, inner nextTick]

Why this behavior is intentional:
- Prevents infinite loops and starvation of I/O / other microtasks.
- Node.js drains the nextTick queue once per phase, then moves to Promises and other microtasks, allowing a fair execution balance.

Analogy:
Think of process.nextTick() like queuing people to speak at a mic:
1. You schedule a few to speak (1, 2, 3).
2. While 2 is speaking, they nominate someone else.
3. That new speaker goes to the back of the line, not straight to the mic.

2. In Timer Queue :

In Node.js, the Timer Queue is a part of the Event Loop that handles asynchronous operations related to time.
Timer queue is not a queue but a min-heap data structure but thinking it as a queue makes the process simpler
The primary functions used to interact with the Timer Queue are:
-> setTimeout() - clearTimeout()
-> setInterval() - clearInterval()

- Experiment 4 : Callbacks in the microtasks queues are executed before callbacks in the timer queue.

The order of priority in the below program is nextTick queue than promise queue than timer queue

```
1. setTimeout(()=>console.log("this is setTimeout 1"),0)
2. setTimeout(()=>console.log("this is setTimeout 2"),0)
3. setTimeout(()=>console.log("this is setTimeout 3"),0)
4. process.nextTick(() => console.log("this is process.nextTick 1"));
5. process.nextTick(() => {
6.   console.log("this is process.nextTick 2");
7.     process.nextTick(() =>
```

```
 8.       console.log("this is the inner next tick inside next tick")
 9.      );
10. });
11. process.nextTick(() => console.log("this is process.nextTick 3"));
12. Promise.resolve().then(() => console.log("this is Promise.resolve 1"));
13. Promise.resolve().then(() => {
14.   console.log("this is Promise. resolve 2");
15.      process.nextTick(() =>
16.       console.log("this is the inner next tick inside Promise then block")
17.      );
18. });
19. Promise.resolve().then(() => console. log("this is Promise.resolve 3"));
20.
21. output :
22. this is process.nextTick 1
23. this is process.nextTick 2
24. this is process.nextTick 3
25. this is the inner next tick inside next tick
26. this is Promise.resolve 1
27. this is Promise. resolve 2
28. this is Promise.resolve 3
29. this is the inner next tick inside Promise then block
30. this is setTimeout 1
31. this is setTimeout 2
32. this is setTimeout 3
```

- Experiment 5 : Callbacks in the microtask queue are executed in between the execution of callbacks in the timer queue

```
 1. setTimeout(()=>console.log("this is setTimeout 1"),0)
 2. setTimeout(()=>{
 3.    console.log("this is setTimeout 2");
 4.    process.nextTick(() => // this is queued in the microtask queue and won't run right away
 5.       console.log("this is the inner next tick inside setTimeout 2")
 6.      );
 7. },0) // after completion of this callback, event loop goes to microtask queue (because as we have learned, after
every callback execution, the event loop goes back and checks microtask queue
 8. setTimeout(()=>console.log("this is setTimeout 3"),0)
 9.
10. process.nextTick(() => console.log("this is process.nextTick 1"));
11. process.nextTick(() => {
12.   console.log("this is process.nextTick 2");
13.      process.nextTick(() =>
14.       console.log("this is the inner next tick inside next tick")
15.      );
16. });
17. process.nextTick(() => console.log("this is process.nextTick 3"));
18. Promise.resolve().then(() => console.log("this is Promise.resolve 1"));
19. Promise.resolve().then(() => {
20.   console.log("this is Promise. resolve 2");
21.      process.nextTick(() =>
22.       console.log("this is the inner next tick inside Promise then block")
23.      );
24. });
25. Promise.resolve().then(() => console. log("this is Promise.resolve 3"));
26.
27. output :
28. this is process.nextTick 1
29. this is process.nextTick 2
30. this is process.nextTick 3
31. this is the inner next tick inside next tick
32. this is Promise.resolve 1
33. this is Promise. resolve 2
34. this is Promise.resolve 3
35. this is the inner next tick inside Promise then block
36. this is setTimeout 1
37. this is setTimeout 2
38. this is the inner next tick inside setTimeout 2
39. this is setTimeout 3
```

3. In I/O Queue :

Most of the async methods from the built-in modules queue the callback function in the I/O queue.

Note : There is no separate "I/O queue" phase in the Node.js event loop instead everything related to I/O happens inside I/O polling which we are discussing in the next topic.

- Experiment 6 : Callbacks in the microtask queues are executed before callbacks in the I/O queue

```
1. const fs = require("fs");
2. fs.readFile(__filename, () => {
3.    console.log("this is readFile 1");
4. })
5. process.nextTick(() => console.log("this is process.nextTick 1"));
6. Promise.resolve().then(() => console.log("this is Promise.resolve 1"));
7.
8. output :
9. this is process.nextTick 1
10. this is Promise.resolve 1
11. this is readFile 1
```

- Experiment 7 : When running setTimeout with delay 0ms and an I/O async method, the order of execution can never be guaranteed

```
1. const fs = require("fs");
2.
3. setTimeout(() => console.log("this is setTimeout 1"), 0);
4.
5. fs.readFile(__filename, () => {
6.   console.log ("this is readFile 1");
7. });
8. output :
9. >> node index.js
10. this is readFile 1
11. this is setTimeout 1
12. >> node index.js
13. this is setTimeout 1
14. this is readFile 1
15. >> node index.js
16. this is readFile 1
17. this is setTimeout 1
```

Why can the order of execution never be guaranteed ?
https://github.com/69JINX/FrontEnd/blob/main/Notes/Notes_Pic/IOqueueWithTimerqueue.png

4. In Check Queue :

The check queue is a specific phase where callbacks scheduled with setImmediate() are executed.
Purpose:
The check queue is designed to execute callbacks immediately after the "poll" phase of the event loop has completed. This provides a way to schedule tasks that should run as soon as possible, but not necessarily in the same tick of the event loop as microtasks (like Promises or process.nextTick).
Event Loop Flow:

- The event loop continuously cycles through various phases.
- After the "poll" phase (which handles I/O events and can potentially block), if the poll phase becomes idle and there are callbacks in the check queue, the event loop will proceed to the check phase to execute those setImmediate() callbacks.
- This allows setImmediate() to execute before any pending setTimeout() callbacks in the timer queue, even if the setTimeout() has a 0ms delay.

In essence, the check queue provides a specific point in the Node.js event loop cycle for executing setImmediate() callbacks, ensuring they run promptly after I/O operations and before other timer-based tasks.

- Experiment 8 : Check queue callbacks are executed after microtask queues callbacks, Timer queue callbacks and I/O queue callbacks are executed

```
1. const fs = require("fs");
2. fs.readFile(__filename, () => {
3.    console.log("this is readFile 1");
4.    setImmediate(() => console.log("this is inner setInmediate inside readFile"));
5. });
```

```
process.nextTick(() => console.log("this is process.nextTick 1"));
Promise.resolve().then(() => console.log("this is Promise.resolve 1"));
setTimeout(() => console.log("this is setTimeout 1"), 0);

output :
this is process.nextTick 1
this is Promise.resolve 1
this is setTimeout 1
this is readFile 1
this is inner setInmediate inside readFile
```

- Experiment 9 : Microtask queues callbacks are executed after I/O callbacks and before check queue callbacks

```
const fs = require("fs");
fs.readFile(__filename, () => {
    console.log("this is readFile 1");
    setImmediate(() => console.log("this is inner setInmediate inside readFile"));
    process.nextTick(()=>{
       console.log("this is inner process.nextTick inside readFile");
    })
    Promise.resolve().then(()=>{
       console.log("this is inner Promise.resolve inside readFile");
    })
});
process.nextTick(() => console.log("this is process.nextTick 1"));
Promise.resolve().then(() => console.log("this is Promise.resolve 1"));
setTimeout(() => console.log("this is setTimeout 1"), 0);

output :
this is process.nextTick 1
this is Promise.resolve 1
this is setTimeout 1
this is readFile 1
this is inner process.nextTick inside readFile
this is inner Promise.resolve inside readFile
this is inner setInmediate inside readFile
```

- Experiment 10 : Microtask queues callbacks are executed in between check queue callbacks. After every single callback in check queue, it will check for microtask queue for any pending callbacks.

```
setImmediate(() => console.log("this is setImmediate 1"));
setImmediate(() => {
      console.log("this is setImmediate 2");
      process.nextTick(() => console. log("this is process.nextTick 1"));
      Promise.resolve().then(() => console.log("this is Promise.resolve 1"));
});
setImmediate(() => console.log("this is setImmediate 3"));

output :
this is setImmediate 1
this is setImmediate 2
this is process.nextTick 1
this is Promise.resolve 1
this is setImmediate 3
```

- Experiment 11 : When running setTimout with delay 0ms and setImmediate method, the order of execution can never be guaranteed cause it can vary slightly depending on the system's state and how fast the code before them executes.

```
setTimeout(() => console.log("this is setTimeout 1"), 0);
setImmediate(() => console.log("this is setImmediate 1"));
```

5. In Check Queue :

The "close queue" is a specific phase within the Node.js event loop, dedicated to handling callbacks associated with the close event of various resources. It is the final phase of a single iteration (or "tick") of the Node.js event loop.

Key aspects of the close queue:

1. Purpose: It holds callback functions that are registered to execute when a resource, such as a stream, socket, or HTTP server, is closed.
2. Triggering: Callbacks are added to the close queue when the close() method is called on a resource, leading to the emission of a close event.
3. Execution Order: Callbacks in the close queue are executed after all other types of callbacks in the event loop's main phases, including timers, I/O, and check phases.
4. Example: If you have a net.Server or http.Server and call server.close(), any callbacks attached to the 'close' event of that server will be placed in the close queue and executed during this phase.

In essence, the close queue ensures that cleanup operations and final actions related to closed resources are handled in an orderly manner at the very end of an event loop cycle.

- Experiment 12 : Close queue callbacks are executed after all other queues callbacks in a given iteration of the event loop

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

const readableStream = fs.createReadStream(__filename);
readableStream.close();

readableStream.on("close", () => { // this callback gets queued and invoked in the close callbacks phase.
    console. log("this is from readableStream close event callback");
})

setImmediate(() => console.log("this is setImmediate 1"));
setTimeout(() => console.log("this is setTimeout 1"), 0);

Promise.resolve().then(() => console.log("this is Promise.resolve 1"));
process.nextTick(() => console.log("this is process.nextTick 1"));

output :
this is process.nextTick 1
this is Promise.resolve 1
this is setTimeout 1
this is setImmediate 1
this is from readableStream close event callback
```

I/O Polling

The event loop in In Node.js, I/O polling is part of the event loop's I/O phase. When asynchronous operations like file reads or network requests are initiated, the OS handles them. The event loop then polls to check if any of those operations are complete. Once completed, the associated callback is pushed into the callback queue to be executed in the appropriate phase.

In Node.js, I/O polling happens during a part of the event loop called the I/O phase.
When you run something like reading a file or making a network request, Node.js asks the operating system (OS) to handle that task in the background.
While the OS is working on it, Node.js doesn't stop — it keeps doing other things. Then, during the I/O phase, Node.js checks (polls) to see if those background tasks are finished.
If a task is done, Node.js adds its callback function to a queue so it can be run soon after.

```
fs.readFile("file.txt", () => {
  console.log("File is read!");
});
```

- The OS reads the file in the background.
- Node.js checks during the I/O phase if it's done.
- When it's ready, it runs your console.log() callback.

There is no separate "I/O queue" phase in the Node.js event loop.
Here's what actually exists:
I/O Polling :

- Happens in the Poll Phase of the event loop.
- This is where Node.js:
  - Waits for I/O (file reads, network requests, etc.) to complete
  - Executes callbacks for completed I/O events

[Timers] → [Pending Callbacks] → [Poll] → [Check] → [Close Callbacks]

What's not true:
"I/O queue" as a distinct named queue or phase in the event loop
➡️ This doesn't exist in the Node.js event loop diagram.

✅ But there is a kind of internal queue:
- Even though we don't call it “I/O queue” officially, internally, Node.js does:
- Keep track of I/O events waiting to be processed
- When an I/O completes (like a file read), the corresponding callback is registered to be executed in the poll phase
So you could say: "There’s no separate I/O queue phase, but internally, completed I/O callbacks are managed and processed in the poll phase."

Analogy:
Think of it like this:
- Poll phase is the I/O handler
- Internally, it has access to a list of "I/O callbacks that just finished"
- It runs them in this phase — no need for a separate “I/O queue” phase like “Timers” or “Check”

To queue a callback function into the check queue, we can use a function called setImmediate.

```
1. setImmediate(() => {
2.   console.log( "this is setImmediate 1")
3. })
```

➢ Experiment 9 : check queue runs before I/O queue if I/O queue is empty

```
 1. const fs = require("fs");
 2.
 3. fs.readFile(__filename, () => {    // if we used readFileSync instead, whole below code will be blocked until the
file is read.
 4.   console. log("this is readFile 1");  // ex : const data = fs.readFileSync(__filename, 'utf-8');
 5. });
 6.
 7. process.nextTick(() => console.log("this is process.nextTick 1"));
 8. Promise.resolve().then(() => console.log("this is Promise.resolve 1"));
 9. setTimeout(() => console.log("this is setTimeout 1"), 0);
10. setImmediate(() => console.log("this is setImmediate 1"));
11.
12. for (let i = 0; i < 2000000000; i++) {}
13.
14. output :
15. this is process.nextTick 1
16. this is Promise.resolve 1
17. this is setTimeout 1
18. this is setImmediate 1
19. this is readFile 1
```

Why check queue (setImmediate) ran before I/O queue (readFile) even though the cheque comes after the i/o queue in event loop ?

Imagine this :

```
1. fs.readFile("file.txt", () => {
2.   console.log("file read");
3. });
```

When Node reaches the Poll phase, if the file is already read (I/O is complete), the callback will run right away in this phase.
If not, Node will wait (poll) until the I/O is done.
This polling means: "Wait until the OS tells us the file is read, or there's some data ready."
So, it blocks in this phase only if there are no setTimeout, setImmediate, or other things scheduled. If there are tasks ready in other phases, Node won't wait too long.

Here's the timeline step-by-step:

1. fs.readFile(__filename) is triggered:
    a. It asks the OS to read a file asynchronously.
    b. The callback is registered to be run later, once the OS completes the read.
2. Microtasks run:
    a. process.nextTick → logs
    b. Promise.resolve().then → logs
3. Timers Phase:
    a. setTimeout(0) → logs
4. Poll Phase:
    a. Node enters the Poll phase.
    b. It checks if any I/O is complete:
    c. Your file has not finished reading yet. It's still in progress.
    d. Poll queue is empty.
    e. Since there's nothing else to do and setImmediate is waiting...
5. Node jumps to Check Phase:
    a. Runs setImmediate() → logs
6. Meanwhile, file reading completes
    a. Callback for fs.readFile is queued to run in the next loop's Poll phase.
    b. Then fs.readFile callback → logs

So the reason is:
The I/O operation hadn't finished yet during the Poll phase because the File reading is done by the OS, not Node
Since setImmediate() was waiting in the Check Phase, Node said:
"Let's not wait here in Poll. Let's move on and run setImmediate."

Inference : I/O events are polled and callback functions are added to the I/O queue only after the I/O is complete

NPM

1. Is is the world's largest software library (registry)
2. It is a software package manager

npm is a Software Library
A book library contains books written by various authors
npm is a library or a registry which contains code packages written by various developers
It is a large public database of JavaScript code that developers from all over the world can use to share and borrow code
If you author a "code package", you can publish it to the npm registry for others to use
If you come across a code package that is authored by someone else and solves the problem you have at hand, you can borrow that code without having to reinvent the wheel

package.json

package.json is npm's configuration file

It is a json file that typically lives in the root directory of your package and holds various metadata relevant to the package
package.json is the central place to configure and describe how to interact with and run your package
It is primarily used by the npm CLI

Node Playlist : https://www.youtube.com/watch?v=4KC-TxNBWsE

Versioning

Versioning of node packages
ex: package.json

```
1. "dependencies": {
2.   "react-icons": "^5.3.0",
3.  }
```

Semantic Versioning
SemVer - is one of the most widely adopted versioning systems
A simple set of rules and requirements that dictate how version numbers are assigned and incremented
It is crucial to keep a semantic and historical track of changes
Version numbers and the way they change convey meaning about the underlying code and what has been modified from one version to the next
syntax : X.Y.Z (where x stands for Major version, Y stands for Minor version and Z stands for a Patch)

Versioning Rules :

- When you fix a bug and the code stays backwards-compatible you increment the patch version.
  For example 1.1.1 to 1.1.2

- When you add new functionality but the code still stays backwards-compatible,
  you increment the minor version
  You also reset the patch version to zero
  For example 1.1.1 to 1.2.0

- When you make changes and the code is no more backwards compatible, you
  increment the major version
  You have to reset the minor and patch version to zero
  For example 1.1.1 to 2.0.0.

Semantic versioning always starts with 0.1.0
0.Y.Z (a major version of zero) is used for initial development
When the code is production-ready, you increment to version 1.0.0
Even the simplest of changes has to be done with an increase in the version number

Scripts

An npm script is a convenient way to bundle common commands for use in a project
They are typically entered in the command line in order to do something with the application
npm scripts are stored in a project's package.json file, giving access to everyone who has access to the codebase
They also ensure that everyone is using the same command with the same options
Common use cases for npm scripts include building your project, starting a development server, compiling CSS, linting, minifying etc
npm scripts are executed using the command npm run <SCRIPT_NAME>

ex: package.json

```
1. "scripts": {
2.   "start": "node index.js",
3.  }
```

```
1. >> npm run start
         OR
2. >> npm start
```

npm test, npm start, npm restart, and npm stop are all aliases for npm run xxx.
For all other scripts you define, you need to use the npm run xxx syntax.
(See the docs at https://docs.npmjs.com/cli/commands/npm-run for more information.)

Buiding CLI Tools

CLI stands for Command Line Interface. A program that you can run from the terminal
Ex: npm and git

- Create a basic CLI tool using node and npm
- Pass options to the CLI
- Add interactivity to the CLI

Cluster Module

Node is single threaded
No matter how many cores you have, node only uses a single core of your CPU
This is fine for I/O operations but if the code has long running and CPU intensive operations, your application might struggle from a performance point of view
The cluster module enables the creation of child processes (also called workers) that run simultaneously

- The Cluster module allows you to leverage multi-core CPUs by creating multiple worker processes that share a single server port. This enables your Node.js application to handle more concurrent requests and improve performance for CPU-intensive tasks.
- The Cluster module creates child processes (workers), each running independently and in its own event loop. A master process manages these workers, distributing incoming connections among them.
- It is essential for scaling Node.js applications that need to handle a high volume of concurrent requests or perform significant CPU-bound computations, allowing them to utilize all available CPU cores effectively.

ex (no_cluster.js) :

```
 1. const express = require('express');
 2. const app = express();
 3.
 4. app.get('/',(req,res)=>{
 5.   res.end("Home Page");
 6. });
 7.
 8. app.get('/slow-page',(req,res)=>{
 9.   for(let i = 0; i<6000000000;i++){} // Simulate CPU work
10.   res.end("Slow Page");
11. });
12.
13. app.listen(4000,()=>{
14.   console.log('Server is running at port 4000');
15. })
```

If you try to access both pages. In the network tab on the Time column (usually 6th column), you can see the home page took almost 4ms and fetched very quickly, on the other hand the /slow-page page took 6.32s (that is way slower then home page because we have a long running for-loop in the controller)

➢ Lets observe a little scenario.

Try to reload (/slow-page) on a tab. While the slow-page is being load, reload the home page on a new tab right after reloading the /slow-page. Because of the /slow-page, the home page will be postpond and will wait for the /slow-page to finish loading and it will also take around 5s.

Why is this happening ?

Well that is because the single thread of nodejs is blocked with the for-loop and server won't be able to respond to any new request. /slow-page is basically blocking the home page. If 1 user hits /slow-page, Everyone else — even those trying to access, will get stuck waiting.

The solutions for this scenarion are :

Use non-blocking code or offload heavy tasks using:

- Async patterns: e.g., setTimeout, Promise, async/await with non-blocking I/O
- Worker threads (built-in in Node.js): offload CPU-intensive tasks
- Child processes (for heavy computation)
- Cluster mode (multiple Node.js processes via PM2 or native cluster module)

We will be using Cluster Module to solve this issue.

How cluster module works in nodejs ?

By default, index.js runs in a single process — and this process does handle all logic: routing, file access, database operations, etc.

No master/worker separation happens unless you explicitly implement clustering using Node.js's cluster module.

Now, if you use the cluster module :

- When we run index.js (node index.js) in the terminal, the file is treated as a "Cluster Master" and this master is encharge of spawning new workers which runs an instance of our node application.
- It is very important to note that the Master is only in charge of the workers (starting, stopping, restarting them if they crash etc.) but doesn not execute the application code itself (like handling HTTP requests or running your Express app). It is not incharge of handling incoming request, reading files etc. That is up to indivisual worker instance.
- Each worker gets its own event loop, memory, and V8 instance and Each worker handles its own incoming requests(like HTTP requests). and doing so we are able to share the workload across different instances without having to block incoming requests

ex (with_cluster.js) :

```
1. const cluter = require("cluster");
2.
3. if(cluter.isMaster) console.log(`Master process ${process.pid} is running`);
4. else console.log(`Worker ${process.pid} started`);
5.
6. output :
7. Master process 66249 is running
```

Creating new workers with cluster.fork()

Now let's focus on what code to run as Master vs Worker. As master we need to create new workers. To do that, we use fork() method on the cluster object. Create multiple workers by calling the fork method on the cluster multiple times.

```
1. cluster.fork(); // Worker 1
2. cluster.fork(); // Worker 2
```

once the workers are created, no incoming requests will be handle by master cluster. every process will be handle by cluster workers.

ex (creating workers) :

```
1. const express = require("express");
2. const app = express();
3. const cluster = require("cluster");
4.
5. if(cluster.isMaster) {
6.   console.log(`Master process ${process.pid} is running`);
7.   cluster.fork();
8.   cluster.fork();
```

```
}
else {
  console.log(`Worker ${process.pid} started`);
  app.listen(4000,()=>{
    console.log('Server is running at port 4000');
  })
}

output :
Master process 67409 is running
Worker 67417 started
Worker 67416 started
Server is running at port 4000
Server is running at port 4000

```

Flow of the code :

1. cluster.isMaster is true for the main process:
   Logs:
   ```
   Master process 67409 is running
   ```
   Then it forks two worker processes:
   ```
   cluster.fork();
   cluster.fork();
   ```
2. Each worker process re-runs the entire file: (in this case, 2 clusters = 2 times file running)
   BUT now cluster.isMaster === false, so both workers start their own Express app on port 4000 so now we have 2 servers to handle more requests and traffic.
3. Each worker runs independently and listens on the same port — this is allowed because Node.js Cluster lets the kernel handle load balancing between them.

How the worker is choosen to process a specific task ?

1. The master process listens on the port behind the scenes.
2. When a request comes in, the OS kernel (like Linux) or Node's cluster module:
   a. Picks a worker (usually via round-robin strategy),
   b. And passes the connection to that worker process using IPC (inter-process communication).
3. So workers don't compete — the master dispatches the requests.

Let's revisit the previous scenario where accessing the /slow-page endpoint caused the entire application to become unresponsive — blocking access to other routes like the home page — until the /slow-page request completed.

With clustering enabled and multiple worker processes spawned, this bottleneck is eliminated. If one worker is occupied handling a high-computation route like /slow-page, incoming requests (such as to /) can now be delegated to an available worker. As a result, the home page remains responsive and loads almost instantly (within 2–3 ms), even while the /slow-page is still processing (which may still take ~4.5 seconds due to CPU-intensive operations).

In essence, clustering leverages multi-core systems by distributing load across separate worker processes, ensuring that a single blocking request does not degrade the responsiveness of the entire application.

example 2 :

```
const express = require("express");
const cluster = require("cluster");
const os = require("os");


if(cluster.isPrimary) {
  console.log(`Master process ${process.pid} is running`);

  for(let i=0;i<os.cpus().length;i++) // creating worker as there are no. of cpus cores
  {
    cluster.fork();
  }
}
```

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

  app.get('/',(req,res)=>{
    res.end(`Worker ${process.pid} completed you request`);
  });

  console.log(`Worker ${process.pid} started`);

  app.listen(4000,()=>{
    console.log('Server is running at port 4000');
  })
}
output :
Master process 14702 is running
Worker 14709 started
Worker 14710 started
Server is running at port 4000
Worker 14711 started
Server is running at port 4000
Worker 14717 started
Server is running at port 4000
Server is running at port 4000
Worker 14719 started
Server is running at port 4000
Worker 14727 started
Server is running at port 4000
```

Try to access localhost:4000 on different browsers, you will see different workers id as the request was solved by different workers.

- browser 1 : Worker 14709 completed you request
- browser 2 : Worker 14710 completed you request

isMaster vs isPrimary :
cluster.isMaster is an older, deprecated property.
cluster.isPrimary is the current, recommended equivalent.
For new development or when updating existing code, cluster.isPrimary should be used to ensure compatibility with recent Node.js versions and best practices. If supporting older Node.js versions is necessary, a check like if (cluster.isPrimary || cluster.isMaster) can be employed for broader compatibility.

Note : It is very important that you create 2 worker threads minimum. If you create only 1, it is the same as no-cluster scenario. The master will not handle any incoming request, resulting in only 1 node instance responsible for all requests.

Why shouldn't we simply create a large number of workers using cluster.fork()?
We should only create as many workers as there are CPU cores on the machine the app is running
If you create more workers than there are logical cores on the computer it can cause an overhead as the system will have to schedule all the created workers with fewer number of cores

How to see number of cores of your system ?

```
In cmd :
>> node
>> require("os").cpus().length;
```

You can use packages to run your application as cluster which will also decide the best number of workers to create for your machine. One of those npm packages is "PM2"

PM2 (Process Manager 2)

PM2, short for Process Manager 2, is a popular production process manager for Node.js applications. It helps developers manage, monitor, and keep Node.js applications running continuously, especially in a production environment.

```
1. >> npm install pm2 -g
```

PM2 should be installed globally so it can be accessible from anywhere on your system using (CLI) Global installation ensures that pm2 commands like start, stop, restart, and logs are available system-wide, simplifying application management, especially in production environments.
for more info, visit : http://pm2.io/
PM2 quick start : https://pm2.keymetrics.io/docs/usage/quick-start/

Key Features and Functions:

1. Process Management:
   PM2 allows you to start, stop, restart, and manage your Node.js applications, ensuring they remain online and available.
2. Process Monitoring:
   It provides features to monitor application status, logs, and resource usage (CPU and memory).
3. Self-Recovery:
   PM2 automatically restarts applications if they crash or encounter errors, minimizing downtime.
4. Clustering and Load Balancing:
   PM2 can distribute traffic across multiple instances of an application, enabling scalability and improving performance.
5. Hot Reloading:
   PM2 can reload applications without any downtime when code changes are detected.
6. Deployment System:
   PM2 offers a deployment system that can streamline the process of deploying applications to different environments.
7. Configuration Files:
   You can create configuration files (Ecosystem files) to manage multiple applications and their settings.
8. Keymetrics Integration:
   PM2 integrates with Keymetrics, a monitoring platform, for more advanced application insights.
9. Cross-Platform Compatibility:
   PM2 works on Linux, macOS, and Windows operating systems.
10. Supports various languages:
    While primarily known for Node.js, PM2 can also manage other applications written in languages like Python and Ruby, and even binaries in your PATH.

How it Works:

1. Installation:
   PM2 is installed globally as an npm package.
2. Starting Applications:
   You use the pm2 start app.js command to start your application, which then runs as a daemon (a background process).
3. Background Daemon:
   PM2 runs in the background, continuously monitoring and managing the application.
4. Commands for Management:
   PM2 provides commands like pm2 list (to see running processes), pm2 stop app.js (to stop), pm2 restart app.js (to restart), and pm2 delete app.js (to remove).

Running a node app using PM2
when running our app using pm2, we don't have to import the cluster module. The pm2 automatically create workers.
to run our index.js using pm2

```
1. >> pm2 start no_cluster.js -i 0
```

To enable the cluster mode, just pass the -i option. the zero indicates we want pm2 to figure out optimum number of workers to create, if you specify the number eg. 2, pm2 will create only 2 workers

Once the pm2 has registered a process with a certain cpu core, you can restart/stop your app with only id of the process instead of using the full file name.

If you want to stop a certain process/worker, use the id of the process/worker

```
1. >> pm2 stop [id]
```

or stop all process

```
1. >> pm2 stop all
```

Same command for the "start" command

If you want to unregistered some process from the cpu core, you can use

```
1. >> pm2 delete [id]
```

or delete unregister all process with

```
1. >> pm2 delete all
```

Logs :

All console logs or errors will not be printed directly into the terminal once your app has been started using pm2 start. Instead, to see all console logs of your app in realtime, use the below command.

```
1. >> pm2 logs
```

It will also show the worker number associated with the file name (main index file, if there are multipe files) which was used to print that console.

Note : we should spawn only 1 app per core. If you try to run processes than there are no. of cpu cores, the Context switching overhead might increase, Performance may degrade, especially under heavy CPU load, CPU-intensive routes in one worker may still block requests for that worker's queue.

Running Frontend nextjs app with pm2 (usually on production environment like EC2) :

```
1. >> pm2 start npm --name My-Frontend_production -- start (in production build for 'npm run start'/'npm start')
2. >> pm2 start npm --name My-Frontend_development -- run dev (in development mode for 'npm run dev')
```

(if you don't want to give any name to the process, it can be as simple as :

```
1. >> pm2 start npm -- start (production)
```

or

```
1. >> pm2 start npm -- run dev  (development)
```

Breakdown :

How 'npm start' converted to this 'long pm2 code' from above

1. pm2 start → tells PM2 to start a process
2. npm → the executable to run (like server.js for backend)
3. --name next-app → PM2 name for managing the app
4. -- → very important — separates PM2's own arguments from the arguments you want to pass to the script
5. start → argument passed to npm, making it:

Every item after '--' is treated as a space-separated argument to the script (npm/server.js), not to PM2 itself.

So this:

```
1. >> pm2 start npm --name next-app -- start
```

is like manually running:

```
1. >> npm start
```

And PM2 just manages the process.

For example:

```
1. >> pm2 start npm --name next-app -- run start -- --port 3001
```

This would translate to:

```
1. >> npm run start -- --port 3001
```

(npm run dev ignores args like port after the script name unless they come after a --, so port 3001 will be ignored if we didn't use double dash before --port 3001. This is npm's rule, not PM2's)

✅ General Syntax:

- PM2 args (like --name) must go before the first --
- Everything after the first -- goes to the script (like npm start)

so this won't work :

```
1. >> pm2 start npm -- run dev --name My-Frontend
```

but this will :

```
1. >> pm2 start npm --name My-Frontend -- run dev
```

Golden Rule for PM2 Command Arguments:

- All arguments for PM2 go before the first --
- All arguments for your script (e.g., npm, next, Node.js) go after the first --
- Environment variables can go either inline+ or in a config file

Syntax:

```
1. >> pm2 start <command> [PM2 options] -- [script args]
```

if including env variables in the command, they should come before the pm2 command

```
1. >> PORT=3001 pm2 start server.js
```

Note : It is recommended to restart your already running frontend instance instead of deleting it and creating new with above script. The above script is usually used to start a nextjs app for the first time with pm2

```
1. >> npm run build // (create a new build after code updation)
2. >> pm2 restart 0 // (use the number/id on whatever id the frontend process is running ["pm2 list" to see all running processes and their ids])
```

(incase of scenario where frontend is not reflecting changed even after restarting, then it is advised to delete the already running process (with 'pm2 delete pid' and start a new one)

Alternative with ecosystem file (recommended for production):

Create ecosystem.config.js:

```
1. module.exports = {
2.   apps: [
3.     {
4.       name: "next-app",
5.       script: "npm",
6.       args: "start",
7.       env: {
8.         NODE_ENV: "production",
9.         PORT: 3000
10.       }
11.     }
12.   ]
13. };
```

Start it with:

```
1. pm2 start ecosystem.config.js
```

NODE_ENV : NODE_ENV is an environment variable in Node.js that specifies the environment in which the application is running, typically either "development", "production", or "test". It allows applications to behave differently based on the environment they are in, enabling configurations, optimizations, and behaviors specific to each environment.

Worker Thread

let's revisit some facts about js and nodejs :

How JS/Nodejs is single threaded and can still handle asynchronous tasks ?
JavaScript is fundamentally a single-threaded language. This means it has one call stack and one memory heap, and it executes code sequentially, one instruction at a time. It must complete the execution of a piece of code before moving on to the next.

However, JavaScript achieves asynchronous behavior and appears to handle multiple tasks concurrently through mechanisms like the event loop and Web APIs (in browsers) or the libuv library (in Node.js). This allows for non-blocking operations, such as handling network requests or I/O operations, without freezing the main execution thread.

While JavaScript itself is single-threaded, environments like browsers and Node.js provide features like Web Workers (in browsers) that allow for true multi-threading by running scripts in separate background threads, enabling the execution of computationally intensive tasks without impacting the main thread's responsiveness.

in Nodejs, the event loop continuously checks for completed async operations and runs their callbacks without blocking the main thread. When an I/O task (like file or network access) is triggered, it's offloaded to the OS or libuv's thread pool. Once done, the callback is queued and executed when the call stack is clear.

libuv enables non-blocking I/O by handling heavy I/O work in background threads, allowing the main thread to stay responsive. This design lets Node.js manage many concurrent connections efficiently. However, CPU-intensive tasks can still block the event loop. For such cases, Node.js provides Worker Threads—a way to run CPU-heavy JavaScript code in parallel threads, enabling true multithreading without affecting the main thread's performance.

>> revisit the Thread Pool topic for better understanding

Network I/O operations on node.js runs on the main thread. it's handled by the main thread using non-blocking I/O, which is what makes Node.js fast and efficient.

Yes, node.js spawns four threads (as we previously read in Thread pool topic) in addition to the main thread but none of them are used for network I/O such as database operations. The threads are:

1. DNS resolver (since some OSes don't support async DNS natively)
2. File system API (because this is messy to do asynchronously cross-platform, reading/writing files)
3. Crypto (Because this uses the CPU, like password hashing)
4. Zlib (zlib operations like zipping files)

Why is Node.js fast if it's single-threaded?
Most I/O operations (like calling a database) don't use the CPU much. They mostly involve waiting for a response from another machine. Node.js takes advantage of this by not blocking the main thread while waiting. Instead:

- When a request (like a DB query) is sent, Node.js moves on to handle other tasks.
- When the response comes back, Node.js runs the callback function that handles the result.
- This is called non-blocking I/O, and it allows Node.js to handle many requests at the same time, even with just one main thread.

What if you want to use multiple CPU cores?
If you have CPU-heavy tasks (like image processing or math), they can block the main thread. In such cases, you can:
>> Use worker_threads to run that logic in separate threads.
>> Use the cluster module or a process manager like PM2 to run multiple Node.js processes on different CPU cores.

Node.js manages thread pools through two primary mechanisms:

1. Libuv's Thread Pool:
    a. Node.js leverages the libuv library, which provides a default thread pool of four threads.
    b. This pool is automatically used for offloading I/O-bound and CPU-intensive operations that would otherwise block the main event loop. Examples include file system operations (e.g., fs.readFile), DNS lookups, and cryptographic functions (e.g., crypto.pbkdf2).
    c. The size of this thread pool can be adjusted using the UV_THREADPOOL_SIZE environment variable, though increasing it beyond the number of logical CPU cores may not always yield performance benefits.
2. Worker Threads Module:
    a. Introduced in Node.js 10.5.0, the worker_threads module allows developers to create and manage their own threads explicitly.
    b. These worker threads run in separate execution contexts, similar to child processes, but within the same Node.js process. This enables parallel execution of CPU-bound tasks without blocking the main event loop, while still allowing for efficient data sharing through mechanisms like SharedArrayBuffer and MessagePort.
    c. These are separate, independent threads of execution, each with its own memory space and V8 instance. They are useful for CPU-intensive tasks that would otherwise block the main thread.
    d. Developers are responsible for managing the lifecycle of these worker threads, including creation, communication, and termination.

Worker Thread:

Node.js Worker Threads, introduced in Node.js 10.5.0 and stable since Node.js 12, provide a mechanism for running JavaScript code in parallel on separate threads within a single Node.js process. This addresses the traditional single-threaded nature of Node.js and its event loop, which can be blocked by CPU-intensive tasks.

Why Worker Threads?

1. Parallelism for CPU-bound tasks:
    Node.js excels at I/O-bound operations due to its non-blocking event loop. However, CPU-intensive tasks (e.g., complex calculations, image processing, data compression) can block the main thread, leading to application unresponsiveness. Worker threads allow offloading these tasks to separate threads, preventing event loop blockage and improving overall performance.
2. Improved responsiveness:
    By delegating heavy computations to workers, the main thread remains free to handle incoming requests, user interactions, and other I/O operations, ensuring a responsive application.

Key Concepts:

1. worker_threads module:
    The built-in module providing the API for creating and managing worker threads.
2. Worker class:
    Used to create new worker threads. It takes a path to a JavaScript file or a textual script as an argument, which will be executed in the worker thread.
3. isMainThread:
    A boolean property within the worker_threads module that indicates whether the current code is running in the main thread or a worker thread. This allows using the same script for both main and worker logic.
4. parentPort:
    An object available within a worker thread, representing the communication channel to its parent thread.
5. Message Passing:
    Communication between the main thread and worker threads occurs via message passing using worker.postMessage() (from main to worker) and parentPort.postMessage() (from worker to main). Data passed between threads is copied, not shared directly, except for ArrayBuffer instances which can be transferred, and SharedArrayBuffer instances which can be shared.
6. Event Handling:

Worker threads emit events like 'message' (for received messages), 'error' (for uncaught errors within the worker), and 'exit' (when the worker thread terminates).

How they work:

Each worker thread runs in its own isolated V8 instance and has its own event loop. This isolation prevents a crashing worker from affecting the main application. While workers share some resources with the main process (like network sockets), they operate independently, enabling true parallelism for JavaScript execution.

Use Cases:

Heavy data processing and transformations, Image and video manipulation, Cryptographic operations, Complex mathematical computations, and Machine learning inference.

index.js

```
1. const express = require('express');
2. const app = express();
3. const { Worker } = require('worker_threads');
4.
5. app.get('/',(req,res)=>{
6.   res.end("Home Page");
7. });
8.
9. app.get('/slow-page',(req,res)=>{
10.   const worker = new Worker('./worker-thread.js'); // creating new worker thread
11.   worker.on('message',(j)=>{  // listening to messages from worker thread using message event
12.     res.end("Slow Page " + j);
13.   });
14. });
15.
16. app.listen(4000,()=>{
17.   console.log('Server is running at port 4000');
18. })
```

worker-thread.js

```
1. const { parentPort } = require('worker_threads');
2.
3. let j = 0;
4.
5. for(let i = 0; i<6000000000;i++) j++;
6.
7. parentPort.postMessage(j); // sending message to main (parent) thread
```

Concept about Number of Worker Threads to create :

1. CPU-Bound Tasks: For tasks that heavily utilize the CPU (e.g., complex calculations, image processing, video compression), a common recommendation is to create a number of worker threads equal to or slightly less than the number of available CPU cores. This allows for parallel execution without excessive context switching overhead. For example, if your machine has 8 cores, you might create 7 worker threads

2. I/O-Bound Tasks: For tasks that primarily involve waiting for external resources (e.g., database queries, network requests), adding more worker threads beyond the number of CPU cores may not significantly improve performance as the bottleneck lies in the external system. Node.js's asynchronous I/O model already handles these efficiently.

Key Considerations:

1. Overhead: Too many threads can hurt performance due to memory usage and context switching.
2. Worker Pools: To efficiently manage worker threads and avoid the overhead of creating and destroying them for each task, consider using a worker pool library (e.g., workerpool, piscina, poolifier). These libraries manage a reusable pool of workers and assign tasks to available threads.
3. Benchmarking: Always test with different thread counts to find what works best for your specific app and workload.

Libuv Threads vs Worker Threads

Both libuv threads and Node.js Worker Threads involve background threads, but they serve different purposes in the Node.js environment.
Worker threads are for CPU-bound tasks that need true parallelism, while the libuv thread pool handles I/O-bound tasks in a non-blocking way.
In essence, Node.js provides both an automatic, behind-the-scenes thread pool for common asynchronous operations via libuv, and a more explicit, developer-managed mechanism for parallelizing custom CPU-intensive tasks using the worker_threads module. when you need to perform a CPU-intensive task in Node.js, you would use a worker_thread to create a separate thread to handle it, rather than relying on the libuv thread pool.
worker_threads in Node.js do not directly utilize the libuv thread pool. They are a separate mechanism for achieving true parallelism by creating new operating system threads, each with its own V8 instance and event loop. The libuv thread pool is primarily used for handling I/O-bound operations in the main event loop.

1) Libuv Thread Pool

1) Purpose: Used for asynchronous I/O operations that could block the event loop, such as:
    a. File system access (fs.readFile)
    b. DNS lookups (dns.lookup)
    c. Compression (zlib)
    d. Some crypto operations (pbkdf2, scrypt)
2) Mechanism: When such an operation is triggered, libuv offloads it to its internal thread pool (default size: 4). A worker thread performs the task, and once done, a callback is queued to run on the main thread (event loop).
3) Scope & Access:
    a. Internally managed by libuv
    b. Not accessible or directly controllable via JavaScript
    c. You can't write or run your own code on these threads

JavaScript Execution (imp):

No — JavaScript code always runs on the main thread. These threads only handle native (C++) I/O operations behind the scenes.
In **Node.js** (and in browsers too), your JavaScript runs in a single thread called the **main thread**.
It doesn't matter if you're looping, parsing JSON, or running an async function — the JS execution itself always happens **one thing at a time**.
Example :

```
1. console.log("Start");
2.
3. setTimeout(() => {
4.   console.log("Timeout finished");
5. }, 1000);
6.
7. console.log("End");
```

Execution flow:

1. The timer counting is *not* done by JavaScript.
2. It's handled by a native thread from Node's underlying C++ library (**libuv**) or by Web APIs if its in frontend
3. The callback only gets pushed back into the main thread when the timer ends.

2) Node.js Worker Threads

1) Purpose: Designed for CPU-intensive JavaScript code that would otherwise block the event loop (e.g., image processing, complex calculations).
2) Mechanism: Worker threads create independent V8 instances (isolates), each with:
    i) Its own event loop
    ii) Its own memory space

JavaScript code runs in parallel across these threads. Communication with the main thread happens through message passing (postMessage, MessagePort).

3) Scope & Access:
    i) Created and managed explicitly by the developer
    ii) Part of the worker_threads module

JavaScript Execution (imp):

Yes — Worker Threads run full JavaScript code independently from the main thread.

Features	Libuv Threads	Worker Threads
Primary Use Case	Async I/O operations	CPU-heavy JS computation
Runs JS Code?	❌ No (callback only on main thread)	✅ Yes (full JS execution)
Isolation	Shared thread pool	Separate V8 isolate per worker
Management	Internal (libuv)	Explicit (via worker_threads)
Memory	Shared with main process	Own memory space
Examples	fs.readFile, dns.lookup	Custom number crunching, ML inference, etc.

Conclusion:

1) Use libuv thread pool (automatically) for I/O-heavy operations — no need to manage it.
2) Use Worker Threads for CPU-heavy JavaScript — when you need true parallelism.

V8 Engine

The V8 engine is an open-source JavaScript and WebAssembly engine developed by Google, primarily written in C++. It is the core component that allows Node.js to execute JavaScript code outside of a web browser environment.
Key functions and features of V8 within Node.js:

1. JavaScript to Machine Code Compilation:
   V8's primary role is to compile JavaScript code directly into native machine code, which the computer's CPU can understand and execute. This process, known as Just-In-Time (JIT) compilation, significantly improves performance compared to interpreting JavaScript line by line.
2. Runtime Environment:
   V8 provides the essential runtime environment for JavaScript, managing memory through efficient garbage collection and optimizing code execution with techniques like hidden classes and inline caching.
3. Bridging JavaScript and C++:
   Node.js, also written in C++, leverages V8 to provide JavaScript with access to low-level system functionalities that JavaScript alone cannot directly handle. This allows Node.js to interact with the operating system for tasks like file system operations and networking, enabling server-side development.
4. Support for Modern JavaScript Features:
   V8 continuously updates to support the latest ECMAScript standards, including features like classes, Promises, and async/await, ensuring Node.js can utilize modern JavaScript syntax and capabilities.

ExpressJS

Express.js, often referred to simply as Express, is a minimal and flexible Node.js web application framework. It is designed for building robust web applications and APIs. Express simplifies the development of server-side applications by providing a set of features for handling routing, middleware, and HTTP utilities.

Key characteristics of Express.js include:

1) **Minimal and Un-opinionated:**

Express provides a lightweight core and does not enforce a specific architectural pattern, allowing developers flexibility in structuring their applications.

2) **Routing:**
   It offers a powerful routing system that allows defining different functions to handle specific HTTP methods (GET, POST, PUT, DELETE, etc.) and URL patterns.
3) **Middleware:**
   Express heavily utilizes middleware functions, which are functions that have access to the request object, the response object, and the next middleware function in the application's request-response cycle. Middleware can be used for tasks like parsing request bodies, authentication, logging, and error handling.
4) **HTTP Utility Methods:**
   It provides a range of utility methods for handling HTTP requests and responses, such as sending various types of responses (text, JSON, files), setting status codes, and redirecting.
5) **Integration with Node.js:**
   Express is built on top of Node.js, leveraging its non-blocking I/O and event-driven architecture for building scalable and high-performance server-side applications.

Express.js is a de facto standard for building web applications and APIs with Node.js and is a core component of the MEAN stack (MongoDB, Express.js, AngularJS, Node.js).