Agenda

- NodeJs
- BuiltIn and User Defined Modules
- http server

NodeJs

- A software platform that is used to build scalable network applications.
- It is developed by Rayn Dahl in 2009
- It is a javascript runtime built on Google's V8 JavaScript Engine
- Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.
- It provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.
- When an I/O operation like reading from the network, accessing a database or the filesystem, then instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.
- This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.
- It has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.
- In Node.js the new ECMAScript standards can be used without problems.

Components of NodeJs

1. JavaScript runtime: V8
2. EventLoop : Libuv (A multi-platform support library which focuses on asynchronous I/O, primarily developed for use by Node.js.)
3. Standard Components
   - Filesystem Access
   - Crypto
   - TCP/UDP
   - HTTP
   - Buffer
   - etc

JavaScript as SingleThreaded

- JavaScript is single-threaded, but it has the unique ability to handle asynchronous tasks in a non-blocking manner.
- This apparent contradiction is resolved by a key architectural feature of the JavaScript runtime environment: the Event Loop.
- To be single-threaded means that the JavaScript engine has only one call stack and can execute only one piece of code at a time.

- The engine processes code sequentially, line by line, on this single thread.
- For most code, this is fine. If a function is called, it is added to the call stack and executed. If that function calls another function, the new function is added on top.
- However, this could be a problem for long-running operations. If the single thread gets stuck waiting for a database query, network request, or a file to load, the entire application would freeze and become unresponsive.

How asynchronous programming works in JS

- Asynchronous programming is the technique that allows JavaScript to perform these long-running tasks without blocking the main execution thread.
- It is a way to handle operations concurrently, not simultaneously.
- This is made possible by the JavaScript runtime environment (like a web browser or Node.js), which consists of several components:

1. Call Stack:

- As mentioned, this is where synchronous JavaScript code is executed. It works on a Last-In, First-Out (LIFO) principle.

2. Web APIs / Background APIs:

- These are features of the runtime environment, not the JavaScript engine itself. When the JavaScript engine encounters an asynchronous task (like setTimeout, a fetch request, or an event listener), it delegates the task to the appropriate API.
- The API runs the operation in the background, freeing up the main thread to continue executing other code.

3. Callback Queue:

- After a background task is completed by a Web API, its associated callback function is placed in a queue, where it waits to be executed.

4. Microtask Queue:

- A separate queue for high-priority asynchronous tasks, such as those from Promises and async/await.
- The Event Loop prioritizes the Microtask Queue over the Callback Queue.

5. Event Loop:

- This is the core mechanism that connects everything.
- It constantly monitors the call stack. When the stack is empty, it checks the Microtask Queue first, then the Callback Queue.
- If it finds any pending callbacks, it moves the first one onto the call stack for execution.

Event Loop & libuv

- The Concept (Event Loop): The idea of a single main thread that delegates long-running tasks and then processes callbacks from a queue after the stack is clear. This pattern allows for high concurrency without the overhead of creating a new thread for every request.

- The Implementation (libuv): The actual code that makes the event loop possible in Node.js. Libuv abstracts the underlying complexities of different operating systems (like using epoll on Linux, kqueue on macOS, and IOCP on Windows) to provide a consistent event-driven API.
- The Event Loop is the mechanism in JavaScript (and Node.js) that allows asynchronous and non-blocking execution.
- It runs on the main thread
- It Manages callbacks and events
- Uses queues to execute things when they're ready
- Even though JavaScript is single-threaded, Node.js can handle many tasks at once because of the event loop + callback queue + background workers(libuv).
- libuv is the C++ library that powers Node.js's I/O.
- It Manages the event loop
- Handles file system, DNS, timers, sockets
- Uses a thread pool for non-blocking execution

Example with code and flow visualization

```
console.log("Start");

setTimeout(() => {
  console.log("From Macrotask");
}, 0);

Promise.resolve().then(() => {
  console.log("From Microtask");
});

console.log("End");
```

- Step 1: Start Execution
  - console.log("Start") is pushed onto the Call Stack.
  - It is executed, and "Start" is printed to the console. The stack is now empty.
- Step 2: Offload setTimeout
  - setTimeout() is pushed onto the Call Stack. It is a Web API, so the timer is offloaded to the Background APIs.
  - The setTimeout function is popped off the Call Stack. The main thread is not blocked.
- Step 3: Offload Promise
  - Promise.resolve() is pushed to the Call Stack, which immediately resolves.
  - Its .then() callback is a microtask.
  - The microtask is placed into the Microtask Queue.
- Step 4: Continue Synchronous Code

  - console.log("End") is pushed onto the Call Stack.
  - It is executed, and "End" is printed to the console. The stack is now empty.

- Step 5: Event Loop Activation (Microtasks)

  - The Event Loop sees the Call Stack is empty.
  - It checks the Microtask Queue, finds the promise callback, and pushes it onto the Call Stack.
  - The callback executes, and "From Microtask" is printed. The Call Stack is empty again.

- Step 6: Event Loop Activation (Macrotasks)

  - The Event Loop checks the Microtask Queue again (it's empty).
  - It then checks the Macrotask Queue, finds the setTimeout callback, and pushes it onto the Call Stack.
  - The callback executes, and "From Macrotask" is printed. The Call Stack is empty.

```
Start
End
From Microtask
From Macrotask
```

- This demonstrates how microtasks (from the Promise) are prioritized over macrotasks (from setTimeout), even though the setTimeout had a 0ms delay.

Use Cases of NodeJs

- Where to use
  - I/O bound applications
  - Data Streaming applications
  - IoT applications
  - JSON API based applications
  - Single Page applications
- Where not to use
  - CPU intensive applications
  - Heavy server sided processing

Installation

1. click on the below link to download the node

```
https://nodejs.org/en/download
```

```
# to check for the installed node version
node --version
```

```
# to run the node application
node hello.js
```

JSON (JavaScript Object Notation)

- Created by Douglas Crockford is defined as a part of javascript langugage in the early 2000s.
- It wasn't untill 2013 that the format was offically specified
- Javascript Objects are simple associative containers where a string key is mapped with a value.
- JSON shows up in many different cases.

1. API
2. Configuration Files
3. Log Messages
4. Database storage

Module

- It is a collection of functions
- We can share a javascript library with multiple applications with the help of this module
- Their are two types of modules

1. Built-in Modules
   - OS
   - FileSystem
   - HTTP
2. User defined Modules

REST (Representational State Transfer) Services

- It is a design pattern
- A request is sent and response is received
- response is always in the form of JSON
- request consists of 2 parts atleast
  1. HTTP method (get,post,put,delete) operation
     - get used to select
     - post used to submit
     - put used for update/edit
     - delete used to remove
  2. path

http server

- An HTTP server is a program/application that listens for incoming HTTP requests (like from a browser) and sends back HTTP responses (like HTML, JSON, etc.).
- In Node.js, we can build an HTTP server using the built-in http module

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

// Create server
const server = http.createServer((req, res) => {
  // Set headers
  res.writeHead(200, { 'Content-Type': 'text/plain' });

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

// Start listening on port 3000
server.listen(3000,'localhost', () => {
  console.log('Server is running at port 3000');
});
```

- `http.createServer()` Creates the server object
- `req` Represents the incoming request (URL, headers, etc.)
- `res` Used to send back data (HTML, JSON, etc.)
- `.listen()` Starts the server on a specific port