**Node.js** - Node.js is an open-source and cross-platform JavaScript runtime environment. Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. This allows Node.js to be very performant.

**Advantages and Disadvantages of NodeJS:**

Advantages:

Scalability: Node.js is highly scalable due to its event-driven, non-blocking architecture, which allows it to handle a large number of concurrent connections efficiently. This makes it suitable for building real-time applications that require high scalability, such as chat applications or streaming services.

Performance: Node.js offers high performance thanks to its asynchronous, event-driven model. It can handle I/O-heavy operations efficiently, making it well-suited for applications with high I/O requirements, such as web servers or APIs.

Large Ecosystem: Node.js has a vast ecosystem of libraries and frameworks available via npm (Node Package Manager), making it easy to find and integrate third-party modules and tools into Node.js applications.

Fast Development: Node.js enables rapid development cycles due to its simplicity, event-driven architecture, and the availability of numerous libraries and tools. This can lead to shorter time-to-market for applications.

Disadvantages:

Callback Hell: Writing asynchronous code in Node.js often involves heavy use of callbacks, which can lead to complex and nested code structures known as "callback hell." While modern JavaScript features such as Promises and async/await can mitigate this issue, it still requires careful handling.

Single-Threaded: Node.js operates on a single-threaded event loop, which means it can be less suitable for CPU-intensive tasks that require parallel processing. Long-running CPU-bound operations can block the event loop and degrade the performance of other requests.

**Understanding of V8 engine:**

V8 is the name of the JavaScript engine that powers Google Chrome. It's the thing that takes our JavaScript and executes it while browsing with Chrome.

V8 is the JavaScript engine i.e. it parses and executes JavaScript code. The DOM, and the other Web Platform APIs (they all makeup runtime environment) are provided by the browser.

**Compilation:**

JavaScript is generally considered an interpreted language, but modern JavaScript engines no longer just interpret JavaScript, they compile it.

JavaScript is internally compiled by V8 with just-in-time (JIT) compilation to speed up the execution.

Just-In-Time (JIT) compilation is a technique used in computer science and programming language implementations to improve the runtime performance of programs. It involves compiling code at runtime, rather than ahead of time (AOT), which allows the compiler to optimize code based on runtime information.

**Run Node.JS using command**

In terminal, write command - node yourfilename.js

**Environment Variable**

Environment variables in Node.js are dynamic values that are set outside of the application code and are accessible to the application during runtime.

They are commonly used to configure the behavior of the application, such as database connection strings, API keys, or application settings.

Environment variables are particularly useful for configuration that may vary between different environments, such as development, staging, and production. By using environment variables, developers can avoid hardcoding sensitive information directly into the codebase and instead configure it dynamically based on the environment in which the application is running.

In Node.js, environment variables are accessed via the process.env object. This object provides a mapping of environment variable names to their values.

**Dotenv** - Dotenv is a zero-dependency module that loads environment variables from a .env file into process.env. Storing configuration in the environment separate from code is based on The Twelve-Factor App methodology.

**Node.JS module**

In Node.js, a module is a reusable piece of code that encapsulates related functionality. Modules are a fundamental building block of Node.js applications, allowing developers to organize code into logical units and facilitate code reuse.

Exporting: Modules can export functions, variables, or objects that they want to make available to other parts of the application. This is typically done using the module.exports or exports object within the module file. (module.exports=something\_class/functions)

Importing: Other modules can import functionality from a module using the require() function. This allows them to use the exported functions, variables, or objects within their own code. (const name = require(‘path of the exported module file’);).

In modern JS (ECMASCRIPT), use import keyword for importing the module. (import React from 'react')).

**Working with npm packages in Node.JS**

We can install the npm packages like nodemon, dotnev, express and many more. (npm i nodemon@5).

**Overview of How the web works.**

A screenshot of a webpage

Description automatically generated

**Understanding of synchronous and asynchronous**

In JavaScript, synchronous and asynchronous operations refer to how code is executed and how it interacts with the event loop. These concepts are essential for understanding how JavaScript handles tasks like I/O operations, network requests, and timeouts.

Synchronous Operations in Node.js:

Blocking Execution: In synchronous code execution in Node.js, each statement is executed one after the other in a sequential manner. If one operation takes a long time to complete (e.g., reading from a file, querying a database), it blocks the execution of subsequent statements until it finishes.

Asynchronous Operations in Node.js:

Non-Blocking Execution: In asynchronous code execution in Node.js, operations are initiated, but the program doesn't wait for them to be completed. Instead, it continues to execute other statements, and once the asynchronous operation completes, a callback function or a promise is triggered to handle the result.

**Libuv:**

Libuv is a multi-platform C library that provides support for asynchronous I/O based on event loops. It was initially developed for Node.js to handle the event-driven architecture.

Libuv provides non-blocking I/O operations, allowing Node.js to handle multiple tasks concurrently without waiting for an operation to complete. This is achieved through a combination of callbacks, event-driven programming, and a worker thread pool.

A screenshot of a computer

Description automatically generated

**Event Loop**

The event loop is a central component of libuv's architecture, responsible for polling for events and executing the corresponding callbacks. It consists of multiple phases, each handling a specific type of event:

Timers: Executes timer callbacks scheduled for the current time. (setTimeout(), setInterval() callbacks)

Pending callbacks: Executes callbacks for completed I/O operations. (I/O callbacks)

Poll: Waits for new events and processes them.

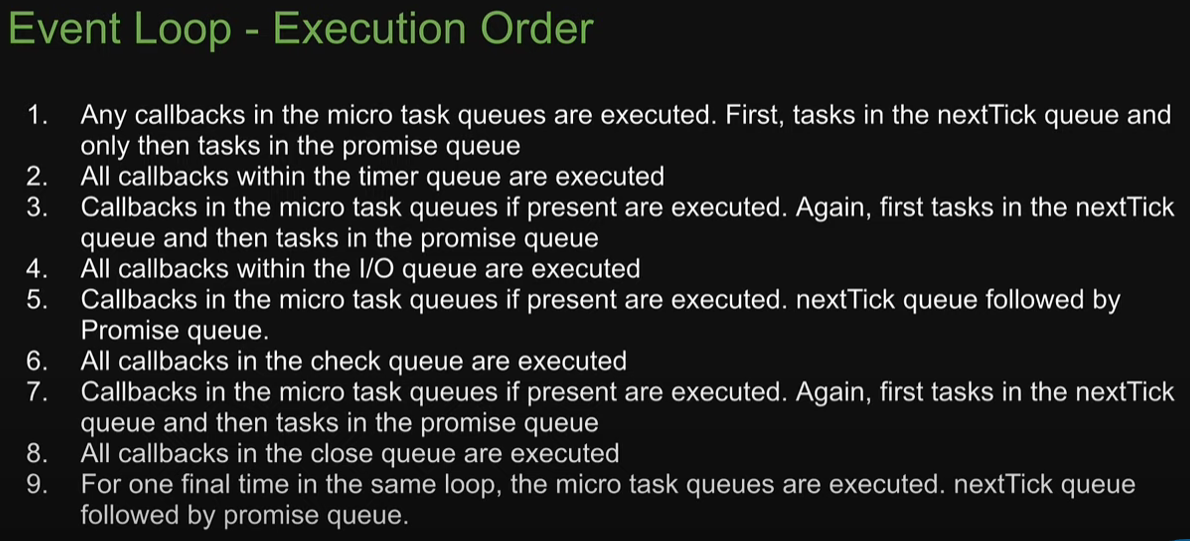
Check: Executes check callbacks after the poll phase. (setImmediate() callbacks)

Close callbacks: Executes callbacks for closing handles. (fs.close() callbacks)

The event loop continually iterates through these phases, ensuring that all events are processed, and their corresponding callbacks are executed.

A diagram of a computer

Description automatically generated



Example: <https://www.youtube.com/watch?v=L18RHG2DwwA&list=PLC3y8-rFHvwj1_l8acs_lBi3a0HNb3bAN&index=1>

Worker Thread Pool

Libuv uses a worker thread pool to offload some I/O operations, such as file system and DNS operations, which can block the event loop. The thread pool allows these operations to be performed asynchronously without blocking the main event loop, ensuring that Node.js remains responsive and efficient.

**Callback and Promises**

A callback is a function that is passed as an argument to another function and is executed after the completion of an asynchronous operation or some other event. (function in another function).

The asynchronous function accepts a callback function as an argument.

Inside the asynchronous function, the callback is called with either an error (if any) or the result of the operation.

The caller of the asynchronous function passes a callback function to handle the result or error.

readFileAsync('example.txt', (err, data) => {

if (err) {

console.error("Error reading file:", err);

} else {

console.log("File contents:", data);

}

});

"Callback hell" is a term used to describe a situation in asynchronous JavaScript programming where multiple nested callbacks are used to handle asynchronous operations. This can lead to code that is difficult to read, understand, and maintain due to its deeply nested structure.

Callback hell typically occurs when dealing with multiple asynchronous operations that depend on each other or need to be executed sequentially. Each asynchronous operation requires a callback function to handle the result or error, and when these operations are nested inside each other, the code can become deeply nested and hard to follow.

For avoiding callback hell, we used promises or async/await.

Promises:

Promises are a built-in feature of JavaScript introduced in ECMAScript 6 (ES6) that provide a cleaner and more structured way to deal with asynchronous operations compared to traditional callback-based approaches. Promises represent the eventual completion or failure of an asynchronous operation and allow you to handle asynchronous code in a more synchronous-like manner.

Here are the key aspects of Promises:

**States**: A Promise can be in one of three states: pending, fulfilled, or rejected.

Pending: Initial state, neither fulfilled nor rejected.

Fulfilled: The operation completed successfully.

Rejected: The operation failed or encountered an error.

**Chaining**: Promises allow you to chain multiple asynchronous operations together using .then() and .catch() methods, which makes it easier to write and reason about asynchronous code.

**Error Handling**: Promises have built-in error handling mechanisms through the .catch() method, which allows you to handle errors in a centralized manner and avoid callback hell.

**Composition**: Promises can be composed using methods like Promise.all() and Promise.race(), allowing you to perform multiple asynchronous operations concurrently or sequentially.

**Immutability**: Once a Promise is fulfilled or rejected, its state cannot be changed. This ensures that the result of an asynchronous operation is determined only once and cannot be modified later.



Async/Await: Use async/await syntax, which provides a more synchronous looking way to write asynchronous code using Promises. Async/await makes asynchronous code easier to read and reason about by avoiding callback hell altogether.

const getDogPic = async () => {

  try {

    const data = await readFilePro(`${\_\_dirname}/dog.txt`);

    console.log(`Breed: ${data}`);

    const res1Pro = superagent.get(

      `https://dog.ceo/api/breed/${data}/images/random`

    );

    const res2Pro = superagent.get(

      `https://dog.ceo/api/breed/${data}/images/random`

    );

    const res3Pro = superagent.get(

      `https://dog.ceo/api/breed/${data}/images/random`

    );

    const all = await Promise.all([res1Pro, res2Pro, res3Pro]);

    const imgs = all.map((el) => el.body.message);

    console.log(imgs);

    await writeFilePro("dog-img.txt", imgs.join("\n"));

    console.log("Random dog image saved to file!");

  } catch (err) {

    console.log(err);

    throw err;

  }

  return "2: READY";

};

(async () => {

  try {

    console.log("1: Will get dog pics!");

    const x = await getDogPic();

    console.log(x);

    console.log("3: Done getting dog pics!");

  } catch (err) {

    console.log("ERROR");

  }

})();

**Node.JS files module:**

fs module.

const fs = require(‘fs’);

fs.readFile(), fs.writeFile(), fs.unlink(), fs.readFileSync(), fs.writeFileSync();

**Working with Folders:**

Models

Controllers

Routes

Services

Server.js

App.js

Node.js path Module:

**`${\_\_dirname}/**tours-simple.json`

REST API Concepts:

A screenshot of a phone

Description automatically generated

**Express** - Express is the most popular **Node web framework** and is the underlying library for a number of other popular Node web frameworks. **Middleware to get body data from server.**

**MongoDB** – Database (NoSQL) (Collections (Table), Documents (Row))

**Mongoose** – Mongoose is an Object Data Modeling (ODM) library for Node.js and MongoDB, higher level of abstraction.

Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB collection and defines the shape of the documents within that collection.

**Morgan** - Morgan is an HTTP request level Middleware. It is a great **tool that logs the requests** along with some other information depending upon its configuration and the presets used. It proves to be very helpful while **debugging** and if you want to create Log files.

**Postman** - Postman is one of the most popular software testing tools which is used for API testing.

