What is Node.js?

Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside of a browser. It is built on Chrome's V8 JavaScript engine.

A cross-platform JavaScript runtime environment refers to a software framework that allows JavaScript code to be executed on multiple operating systems without requiring modifications.

The ability to run on different operating systems like Windows, macOS, and Linux.

A runtime environment provides the necessary infrastructure for executing code written in a specific programming language. For JavaScript, this includes the engine that interprets and executes the code, as well as the necessary libraries and modules.

Node.js uses the V8 JavaScript engine (developed by Google for the Chrome browser) to execute JavaScript code. It provides various built-in modules and APIs to perform tasks like file system operations, networking, and more.

What are some key features of Node.js?

Asynchronous and Event-Driven, Fast Execution, Single-Threaded but Highly Scalable, No Buffering, Cross-Platform.

Single-Threaded: Node.js runs on a single thread, using a single process to handle all requests.

Event Loop :

1. An event loop is an endless loop, which waits for tasks, executes them, and then sleeps until it receives more task
2. The event loop executes the tasks starting from the oldest first.
3.

https://medium.com/@mmoshikoo/event-loop-in-nodejs-visualized-235867255e81

What is the difference between setImmediate() and process.nextTick()?

`setImmediate()` schedules a callback to be executed in the next iteration of the event loop, while process.nextTick() schedules a callback to be executed at the end of the current operation, before the event loop continues.

Explain the use of streams in Node.js.

Streams in Node.js provide an efficient way to handle data by processing it in chunks. They are ideal for handling large files, network communications, and any situation where you need to manage data flow without loading everything into memory at once. Using readable, writable, duplex, and transform streams, you can build powerful and memory-efficient applications. Node.js provides several types of streams: readable, writable, duplex, and transform.

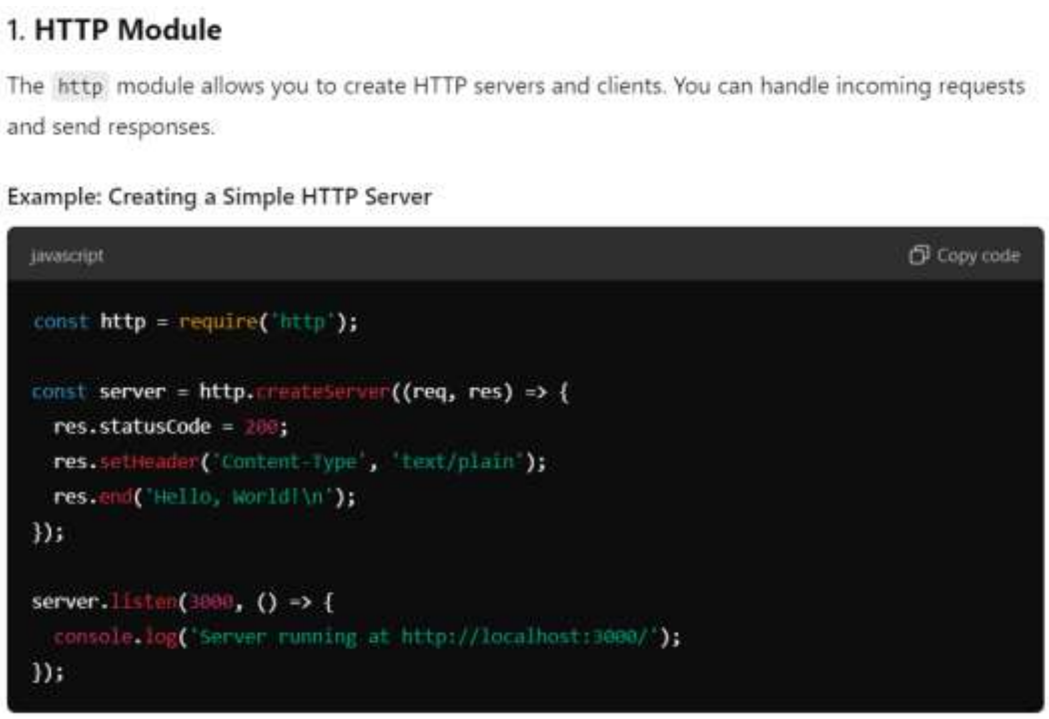
Example: Readable Stream fs.createReadStream

Example: Writable Stream const writableStream = fs.createWriteStream('output.txt');

Explain callback in Node.js.

A callback is a function passed as an argument to another function, which is then executed (called back) after the completion of some asynchronous operation. This allows for non-blocking code execution, enabling other operations to continue while waiting for the asynchronous task to complete.

Node js inbuilt modules



1. HTTP Module

The `http` module allows you to create HTTP servers and clients. You can handle incoming requests and send responses.

Example: Creating a Simple HTTP Server

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

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello, World!\n');
});

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

2. File System Module (fs)

The `fs` module provides an API to interact with the file system, allowing you to read, write, and manipulate files.

Example: Reading a File

javascript Copy code

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

fs.readFile('example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error reading file:', err);
    return;
  }
  console.log('File content:', data);
});
```

3. Path Module

The `path` module helps in handling and transforming file paths. It provides utilities for working with file and directory paths.

Example: Joining Paths

javascript Copy code

```javascript
const path = require('path');

const filePath = path.join(__dirname, 'example.txt');
console.log('File path:', filePath);
```

4. OS Module

The `os` module provides operating system-related utility methods and properties, allowing you to interact with the underlying OS.

Example: Getting System Information

javascript Copy code

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

console.log('Operating System:', os.platform());
console.log('CPU Architecture:', os.arch());
console.log('Free Memory:', os.freemem());
```

5. Events Module

The `events` module allows you to create and handle custom events. It provides an event-driven architecture for Node.js applications.

Example: Creating and Using an Event Emitter

javascript Copy code

```javascript
const EventEmitter = require('events');

const myEmitter = new EventEmitter();

myEmitter.on('event', () => {
  console.log('An event occurred!');
});

myEmitter.emit('event');
```

6. Buffer Module

The `buffer` module provides a way to handle binary data. It allows you to manipulate raw memory buffers directly.

Example: Creating and Using Buffers

javascript Copy code

```javascript
const buf = Buffer.from('Hello, World!');
console.log('Buffer content:', buf.toString());
```

7. Crypto Module

The `crypto` module provides cryptographic functionalities, including hashing, HMAC, encryption, and decryption.

Example: Creating a SHA-256 Hash

javascript Copy code

```javascript
const crypto = require('crypto');

const hash = crypto.createHash('sha256').update('Hello, World!').digest('hex');
console.log('SHA-256 Hash:', hash);
```

8. **URL Module**

The `url` module provides utilities for URL resolution and parsing.

Example: Parsing a URL

```javascript
const { URL } = require('url');

const myURL = new URL('https://example.com:8000/path?query=1#fragment');
console.log('Hostname:', myURL.hostname);
console.log('Pathname:', myURL.pathname);
console.log('Search Params:', myURL.search);
```

9. **Query String Module**

The `querystring` module provides utilities for parsing and formatting URL query strings.

Example: Parsing a Query String

```javascript
const querystring = require('querystring');

const parsed = querystring.parse('foo=bar&abc=xyz&abc=123');
console.log('Parsed Query String:', parsed);
```

10. Child Process Module

The `child_process` module allows you to spawn child processes in Node.js, enabling you to execute shell commands.

Example: Spawning a Child Process

javascript   Copy code

```javascript
const { exec } = require('child_process');

exec('ls -l', (error, stdout, stderr) => {
  if (error) {
    console.error(`Error: ${error.message}`);
    return;
  }
  if (stderr) {
    console.error(`stderr: ${stderr}`);
    return;
  }
  console.log(`stdout: ${stdout}`);
});
```

11. Cluster Module

The `cluster` module allows you to create multiple child processes that share the same server port, improving performance on multi-core systems.

Example: Creating a Simple Cluster

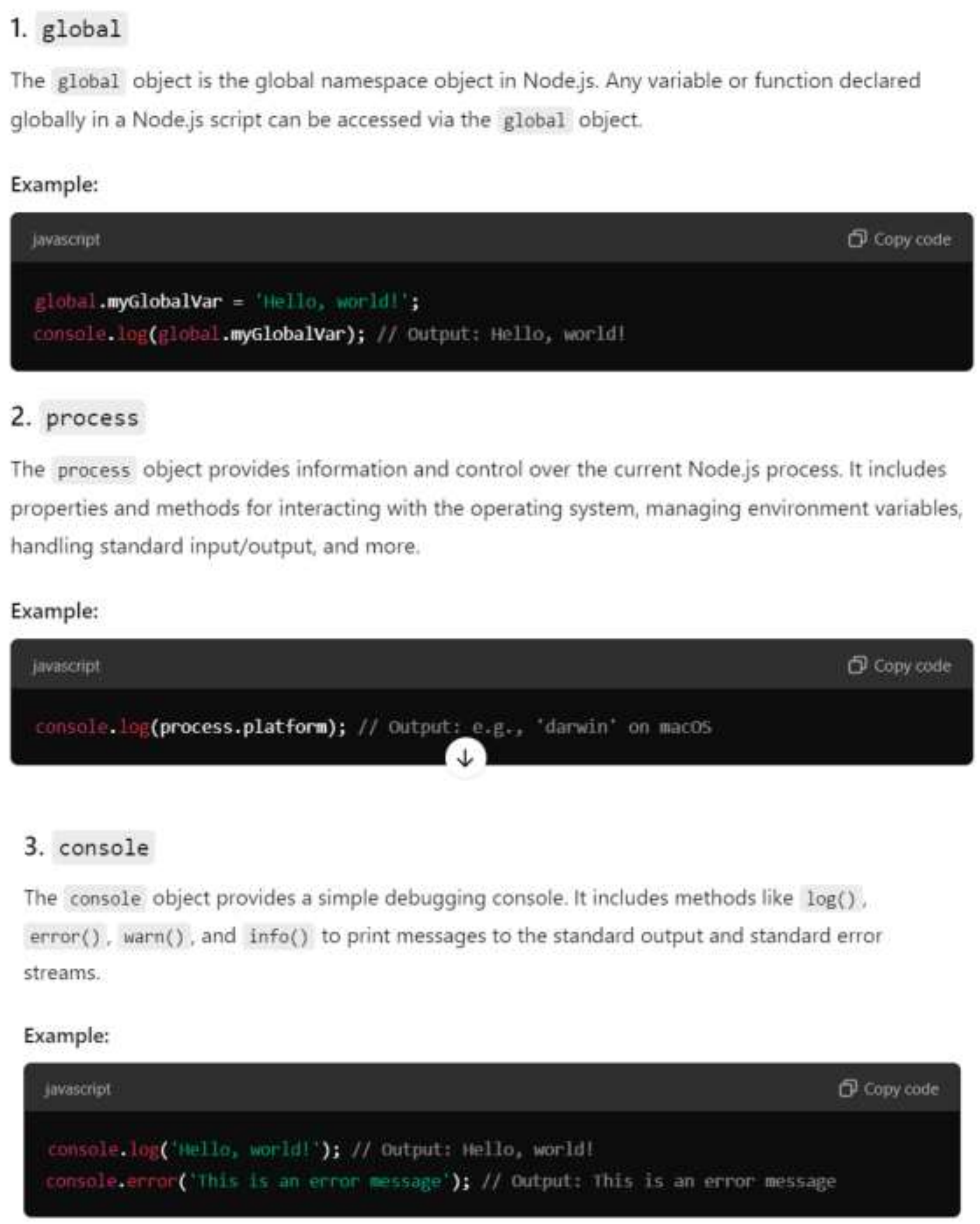
javascript   Copy code

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

if (cluster.isMaster) {
  // Fork workers.
  for (let i = 0; i < 4; i++) {
    cluster.fork();
  }
} else {
  // Workers can share any TCP connection.
  http.createServer((req, res) => {
    res.writeHead(200);
    res.end('Hello from worker: ' + cluster.worker.id);
  }).listen(8000);
}
```

Global Objects In Node JS

In Node.js, there are several global objects that are available in all modules and can be accessed without importing or requiring any module. These global objects provide a variety of functionalities that are essential for Node.js development. Here’s an overview of some of the most important global objects in Node.js:



1. `global`

The `global` object is the global namespace object in Node.js. Any variable or function declared globally in a Node.js script can be accessed via the `global` object.

Example:

```javascript
global.myGlobalVar = 'Hello, world!';
console.log(global.myGlobalVar); // Output: Hello, world!
```

2. `process`

The `process` object provides information and control over the current Node.js process. It includes properties and methods for interacting with the operating system, managing environment variables, handling standard input/output, and more.

Example:

```javascript
console.log(process.platform); // Output: e.g., 'darwin' on macOS
```

3. `console`

The `console` object provides a simple debugging console. It includes methods like `log()`, `error()`, `warn()`, and `info()` to print messages to the standard output and standard error streams.

Example:

```javascript
console.log('Hello, world!'); // Output: Hello, world!
console.error('This is an error message'); // Output: This is an error message
```

4. `Buffer`

The `Buffer` object is used to handle binary data directly. It can be used to read or manipulate streams of binary data.

Example:

javascript   Copy code

```javascript
const buf = Buffer.from('Hello, world!');
console.log(buf.toString()); // Output: Hello, world!
```

5. `setImmediate()` and `clearImmediate()`

`setImmediate()` schedules a function to be executed immediately after the current event loop cycle. `clearImmediate()` can be used to cancel the execution of a function scheduled with `setImmediate()`.

Example:

javascript   Copy code

```javascript
const immediateId = setImmediate(() => {
  console.log('This runs immediately after the current event loop cycle.');
});

clearImmediate(immediateId); // Cancels the immediate execution
```

6. `setTimeout()` and `clearTimeout()`

`setTimeout()` schedules a function to be executed after a specified delay in milliseconds. `clearTimeout()` cancels the execution of a function scheduled with `setTimeout()`.

Example:

javascript   Copy code

```javascript
const timeoutId = setTimeout(() => {
  console.log('This runs after 1 second.');
}, 1000);

clearTimeout(timeoutId); // Cancels the timeout
```

7. `setInterval()` and `clearInterval()`

`setInterval()` schedules a function to be executed repeatedly at specified intervals in milliseconds. `clearInterval()` cancels the execution of a function scheduled with `setInterval()`.

Example:

javascript Copy code

```javascript
const intervalId = setInterval(() => {
  console.log('This runs every 1 second.');
}, 1000);

clearInterval(intervalId); // Cancels the interval
```

8. `require()`

`require()` is a function used to include modules in a Node.js application. It loads modules that are defined in separate files.

Example:

javascript Copy code

```javascript
const fs = require('fs'); // Includes the file system module
```

9. `module` and `exports`

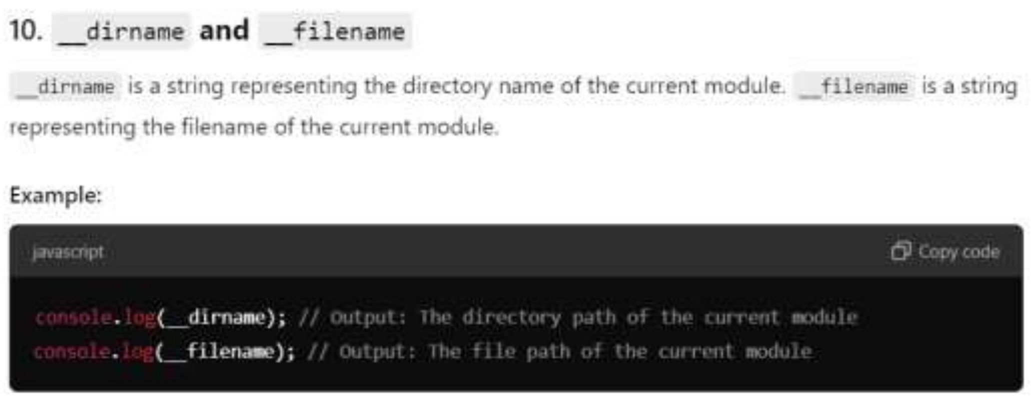
The `module` object represents the current module and allows you to export variables or functions from the module using `module.exports`. `exports` is an alias for `module.exports`.

Example:

javascript Copy code

```javascript
// In a file named myModule.js
exports.myFunction = function() {
  console.log('This is my function');
};

// In another file
const myModule = require('./myModule');
myModule.myFunction(); // Output: This is my function
```



10. `__dirname` and `__filename`

`__dirname` is a string representing the directory name of the current module. `__filename` is a string representing the filename of the current module.

Example:

```javascript
console.log(__dirname); // Output: The directory path of the current module
console.log(__filename); // Output: The file path of the current module
```

REPL

The Node. js Read-Eval-Print-Loop (REPL) is an interactive shell that processes Node. js expressions. The shell reads JavaScript code the user enters, evaluates the result of interpreting the line of code, prints the result to the user, and loops until the user signals to quit. The REPL is bundled with every Node

Different between package.json and package.lock.json

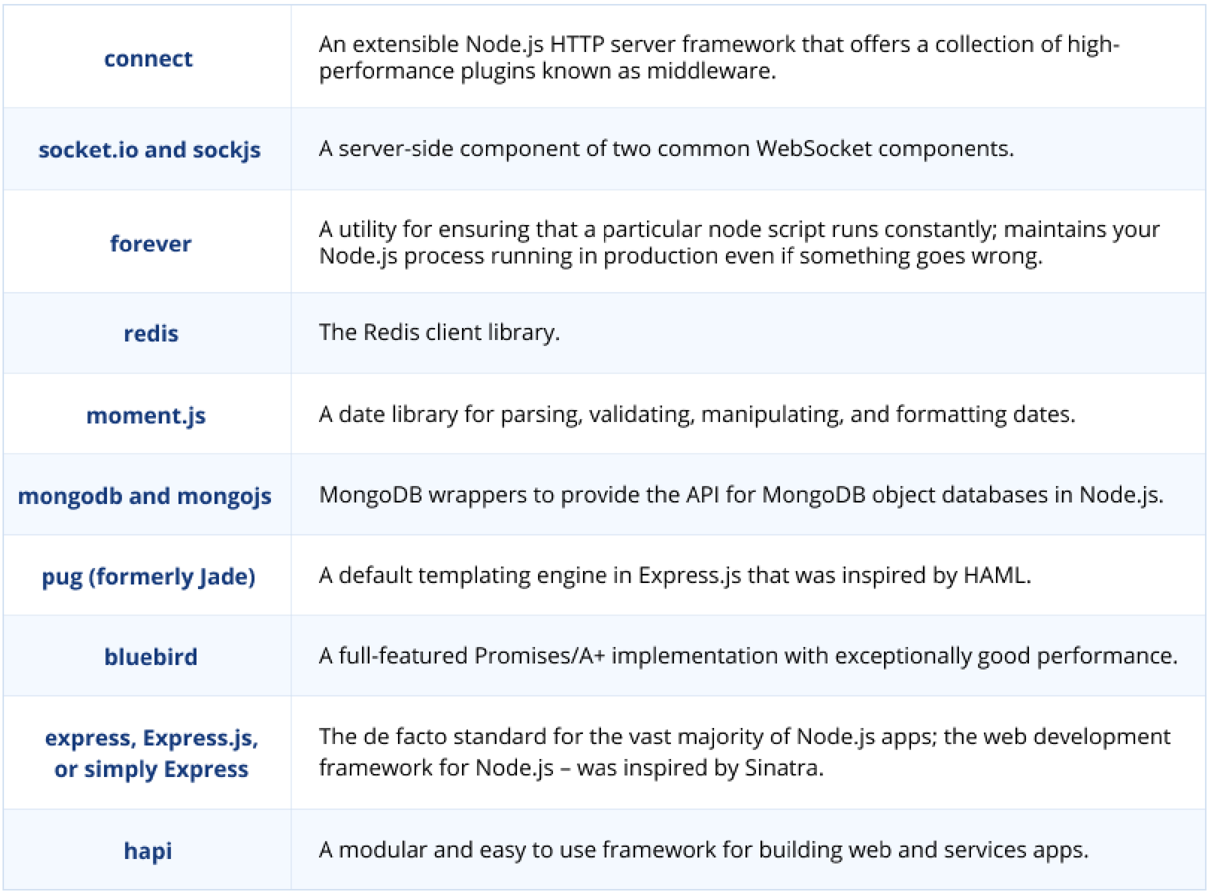
package.json: The main configuration file for a Node.js project, listing metadata, dependencies, and scripts.

package-lock.json: A lock file that records the exact versions of dependencies installed, ensuring consistency across different environments.

How can clustering be used to upgrade Node performance?

Clustering enables Node applications to make the best use of multi-core system resources. As a single-thread platform, Node just uses one processor. Therefore, the remaining cores can be left idle. Developers can launch several processes in cluster mode, creating numerous instances of the event loop. The Cluster Manager aids developers in keeping track of each instance's state.

Mention the different NPM module kinds that are accessible and often used in Node.js



connect	An extensible Node.js HTTP server framework that offers a collection of high-performance plugins known as middleware.
socket.io and sockjs	A server-side component of two common WebSocket components.
forever	A utility for ensuring that a particular node script runs constantly; maintains your Node.js process running in production even if something goes wrong.
redis	The Redis client library.
moment.js	A date library for parsing, validating, manipulating, and formatting dates.
mongodb and mongojs	MongoDB wrappers to provide the API for MongoDB object databases in Node.js.
pug (formerly Jade)	A default templating engine in Express.js that was inspired by HAML.
bluebird	A full-featured Promises/A+ implementation with exceptionally good performance.
express, Express.js, or simply Express	The de facto standard for the vast majority of Node.js apps; the web development framework for Node.js – was inspired by Sinatra.
hapi	A modular and easy to use framework for building web and services apps.

Express JS

```
const express = require('express');
const app = express();
const PORT = 8000;

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(PORT, () => {
  console.log(`Server is listening at port :${PORT}`);
});
```

Which major tools can be integrated with Express JS?

- Database tools: MongoDB, MySQL, PostgreSQL.
- Template Engines: EJS, Pug, Mustache.
- Authentication libraries: Passport.js.
- Logging libraries: Morgan, Winston.
- Validation libraries: Joi, express-validator.
- ORM libraries: Sequelize, Mongoose.

Explain what CORS is in Express JS?

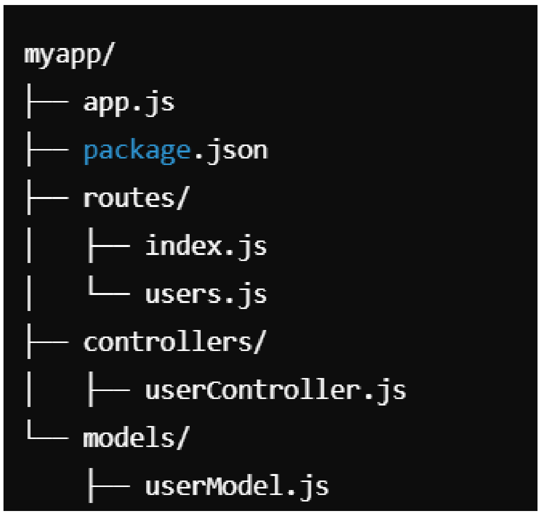
CORS (Cross-Origin Resource Sharing) is a security feature implemented by web browsers to control how web pages in one domain can request and interact with resources hosted on another domain.
In the context of Express.js, CORS refers to a middleware that enables Cross-Origin Resource Sharing for your application. This allows the application to control which domains can access your resources by setting HTTP headers.

How to secure Express.Js application?

It is very important to secure your application to protect it against various security threats. We can follow few best practices in our Express.js app to enhance the security of our application.

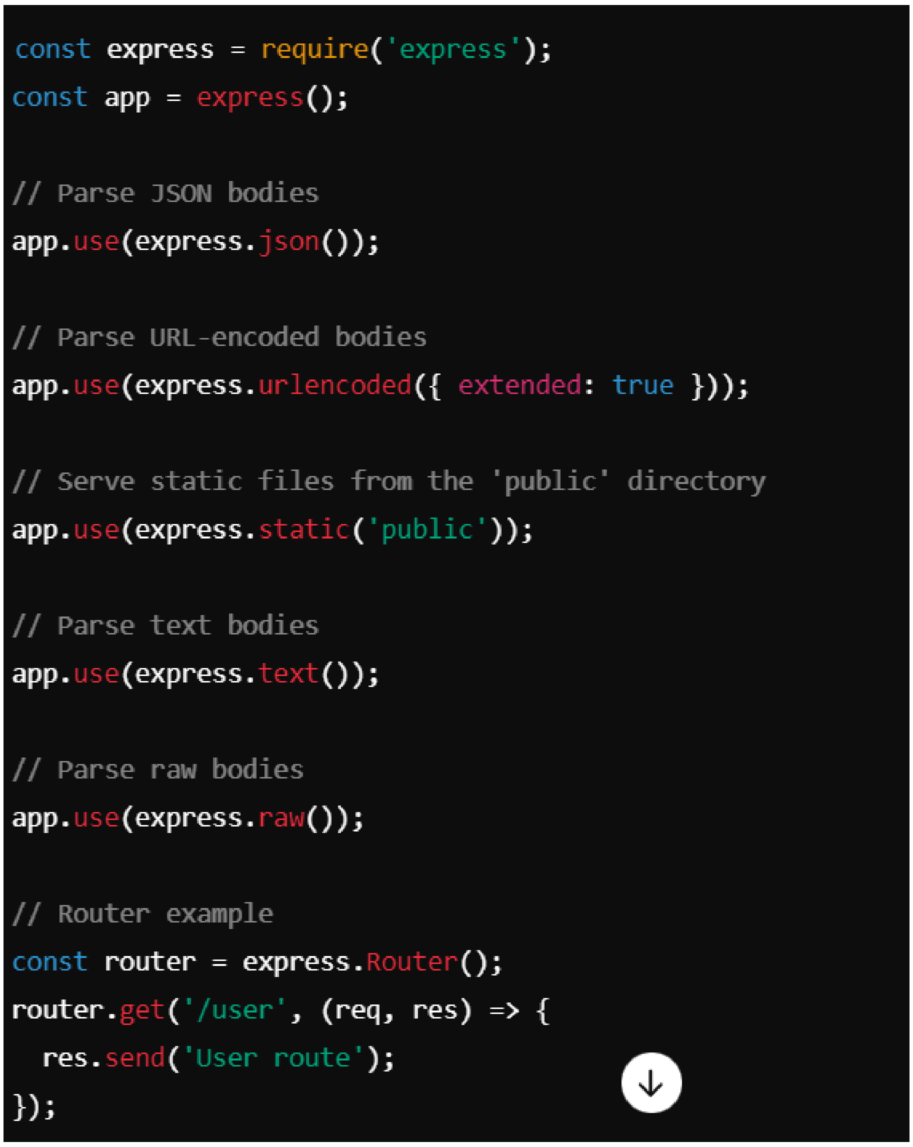
- Keep Dependencies Updated: Regularly update your project dependencies, including Express.js and other npm packages.
- Use Helmet Middleware: The helmet middleware helps secure your application by setting various HTTP headers. It helps prevent common web vulnerabilities.
- Set Secure HTTP Headers: Configure your application to include secure HTTP headers, such as Content Security Policy (CSP), Strict-Transport-Security (HSTS), and others.
- Use HTTPS: Always use HTTPS to encrypt data in transit. Obtain an SSL certificate for your domain and configure your server to use HTTPS.
- Secure Database Access: Use parameterized queries or prepared statements to prevent SQL injection attacks. Ensure that your database credentials are secure and not exposed in configuration files.

How do you structure an Express.js application?



```
myapp/
├── app.js
├── package.json
├── routes/
│   ├── index.js
│   └── users.js
├── controllers/
│   ├── userController.js
└── models/
    ├── userModel.js
```

Express Middleware



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

// Parse JSON bodies
app.use(express.json());

// Parse URL-encoded bodies
app.use(express.urlencoded({ extended: true }));

// Serve static files from the 'public' directory
app.use(express.static('public'));

// Parse text bodies
app.use(express.text());

// Parse raw bodies
app.use(express.raw());

// Router example
const router = express.Router();
router.get('/user', (req, res) => {
  res.send('User route');
});
```

Node JS cluster module

https://www.youtube.com/watch?v=9RLeLngtQ3A

Fork method in child-process

https://www.youtube.com/watch?v=7cFNTD73N88

exec, execFile and spawn in child-process

https://www.youtube.com/watch?v=bbmFvCbVDqo

exec: Runs a command in a shell, buffers the output, suitable for simple commands or when you need to capture the full output.

spawn: Starts a process without a shell, streams the output, suitable for handling large outputs.

execFile: Runs an executable file directly, without spawning a shell, more secure than exec.

Data Structures

- **Arrays and Linked Lists** for basic collections and sequential data.
- **Stacks and Queues** for LIFO and FIFO access patterns.
- **Hash Tables** for fast key-value lookups.
- **Trees** for hierarchical data and balanced operations.
- **Heaps** for efficient priority queue implementation.
- **Graphs** for representing networks and relationships.
- **Tries** for prefix-based searching.
- **Sets** for unique collections and set operations.

Design Pattern

1. **Module Pattern**

- **Purpose:** Encapsulates functionality and exposes a public API while hiding implementation details.
- **Example:** IIFE (Immediately Invoked Function Expression) to create private and public members.

2. **Observer Pattern**

- **Purpose:** Allows objects to subscribe to events and get notified when those events occur.
- **Example:** Using Node.js's EventEmitter to listen and emit events.

3. **Singleton Pattern**

- **Purpose:** Ensures a class has only one instance and provides a global point of access.
- **Example:** A class that checks if an instance already exists and returns it.

4. **Factory Pattern**

- **Purpose:** Creates objects without specifying the exact class of the object.
- **Example:** A function that creates different vehicle types (e.g., car or truck) based on input.

5. **Middleware Pattern**

- **Purpose:** Defines a series of functions that process request and response objects in a pipeline.
- **Example:** In Express.js, middleware functions handle tasks like logging and authentication.

6. **Promise Pattern**

- **Purpose:** Manages asynchronous operations to avoid callback hell.
- **Example:** Using promises to handle the result of an asynchronous file read operation.

7. **Strategy Pattern**

- **Purpose:** Defines a family of algorithms, encapsulating each one to make them interchangeable.
- **Example:** A class that takes different functions (e.g., add, subtract) to perform operations.

8. **Async/Await Pattern**

- **Purpose:** Writes asynchronous code that looks synchronous, enhancing readability.
- **Example:** Using async and await keywords to handle asynchronous API calls in a cleaner way.

These patterns help structure Node.js applications effectively, promoting better organization, scalability, and maintainability.

Solid Principle

The **SOLID** principles are a set of five design principles aimed at making software designs more understandable, flexible, and maintainable. They are particularly relevant in object-oriented programming but can be applied in other paradigms as well. Here's a brief overview:

1. Single Responsibility Principle (SRP)

- **Definition:** A class should have only one reason to change, meaning it should only have one job or responsibility.
- **Benefit:** Simplifies code by separating concerns, making classes easier to understand and maintain.

2. Open/Closed Principle (OCP)

- **Definition:** Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
- **Benefit:** Allows for new functionality to be added without altering existing code, reducing the risk of introducing bugs.

3. Liskov Substitution Principle (LSP)

- **Definition:** Subtypes must be substitutable for their base types without altering the correctness of the program.
- **Benefit:** Ensures that derived classes extend base classes without changing their expected behavior, promoting polymorphism.

4. Interface Segregation Principle (ISP)

- **Definition:** A client should not be forced to depend on interfaces it does not use. Instead of one large interface, multiple smaller interfaces should be created.
- **Benefit:** Reduces the impact of changes and increases the flexibility of the system by ensuring that clients only know about methods that are relevant to them.

5. Dependency Inversion Principle (DIP)

- **Definition:** High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces). Abstractions should not depend on details; details should depend on abstractions.
- **Benefit:** Promotes decoupling of components, making the system easier to manage and test.

Summary

- **SRP**: One class, one responsibility.
- **OCP**: Extend without modifying.
- **LSP**: Subtypes must be substitutable.
- **ISP**: Small interfaces are better than large ones.
- **DIP**: Depend on abstractions, not on concrete implementations.

Software Development Life Cycle (SDLC)

1. Planning

- **Objective:** Define the scope and purpose of the project, including objectives, feasibility, and resources required.
- **Activities:** Requirement gathering, project planning, and risk assessment.

2. Requirements Analysis

- **Objective:** Gather and analyze business requirements to understand what the software should achieve.
- **Activities:** Stakeholder interviews, requirement specification, and documentation.

3. Design

- **Objective:** Create the architecture and design of the software based on requirements.
- **Activities:** Designing system architecture, user interfaces, and database structures.

4. Implementation (Coding)

- **Objective:** Write the actual code to build the software application.
- **Activities:** Developers write code based on the design specifications and conduct initial testing.

5. Testing

- **Objective:** Identify and fix defects in the software to ensure it meets requirements.
- **Activities:** Various testing methods (unit, integration, system, user acceptance) are performed to verify functionality.

6. Deployment

- **Objective:** Release the software to users.
- **Activities:** Install the software in the production environment and make it available to end-users.

7. Maintenance

- **Objective:** Address issues, provide updates, and improve the software after deployment.
- **Activities:** Bug fixing, performance tuning, and adding new features as needed.

Summary

The SDLC provides a systematic approach to software development, ensuring quality and efficiency. It allows teams to manage complex projects, minimize risks, and deliver software that meets user needs. The main phases are **Planning, Requirements Analysis, Design, Implementation, Testing, Deployment,** and **Maintenance**.

https://www.knowledgehut.com/interview-questions/software-architect-interview-questions