400+ Interview Questions and Answers

Node JS

MCQ Format Questions

Interview Questions and Answers

Manish Dnyandeo Salunke

425 Mulesoft Interview Questions and Answers

MCQ Format

Created by: Manish Dnyandeo Salunke

Online Format: https://bit.ly/online-courses-tests

About Author

Manish Dnyandeo Salunke is a seasoned IT professional and passionate book writer from Pune, India. Combining his extensive experience in the IT industry with his love for storytelling, Manish writes captivating books. His hobby of writing has blossomed into a significant part of his life, and he aspires to share his unique stories and insights with readers around the world.

What is the primary role of the Node.js Event Loop?

Option 1:
Managing asynchronous operations

Option 2:
Handling file I/O operations

Option 3:
Parsing JSON data

Option 4:
Running SQL queries

Correct Response:
1.0

Explanation:
The primary role of the Node.js Event Loop is to manage asynchronous operations. Node.js is designed to be non-blocking and event-driven, and the Event Loop ensures that callbacks are executed when asynchronous operations complete.

How do you import a module in Node.js using CommonJS module syntax?

Option 1:
require('module_name')

Option 2:
import module_name from 'module'

Option 3:
include('module_name')

Option 4:
import('module_name')

Correct Response:
1.0

Explanation:
To import a module in Node.js using CommonJS module syntax, you use the require('module_name') syntax. CommonJS is the default module system in Node.js, and this syntax allows you to include and use external modules in your code.

Which method is used to create a simple server that returns "Hello, World!" using the HTTP module in Node.js?

Option 1:
http.createServer()

Option 2:
server.start()

Option 3:
createServer('Hello, World!')

Option 4:
http.startServer()

Correct Response:
1.0

Explanation:
To create a simple server that returns "Hello, World!" using the HTTP module in Node.js, you use the http.createServer() method. This method creates an HTTP server instance, and you can define how it responds to incoming requests.

What is the difference between setImmediate() and process.nextTick() in the Node.js Event Loop?

Option 1:
They both perform the same function.

Option 2:
setImmediate() runs in the next iteration of the event loop.

Option 3:
process.nextTick() runs before setImmediate().

Option 4:
setImmediate() runs before process.nextTick().

Correct Response:
3.0

Explanation:
setImmediate() and process.nextTick() are both used to schedule a callback function, but they have different timing within the event loop. Understanding their order is crucial for managing asynchronous tasks.

How does Node.js handle module caching, and what is a potential side effect of this feature?

Option 1:
Node.js caches modules after the first require.

Option 2:
Caching only occurs for built-in modules.

Option 3:
Module caching can lead to stale data issues.

Option 4:
Modules are never cached in Node.js.

Correct Response:
1.0

Explanation:
Node.js caches modules to improve performance, but developers should be aware of potential issues, such as stale data, when dealing with frequently changing modules.

Can you explain how to use the exports object to make certain functions or objects available to other modules in Node.js?

Option 1:
exports is an alias for module.exports.

Option 2:
exports can be reassigned to a new object.

Option 3:
Changing exports after assignment has no effect.

Option 4:
exports is a reference to the calling module.

Correct Response:
2.0

Explanation:
The exports object in Node.js allows developers to expose functions and objects to other modules. However, reassigning exports can lead to unexpected behavior, so it's essential to understand its behavior.

Describe how the libuv library contributes to Node.js Event Loop's non-blocking I/O operations.

Option 1:
Manages thread pools efficiently

Option 2:
Implements the JavaScript engine

Option 3:
Handles HTTP requests

Option 4:
Provides database connections

Correct Response:
1.0

Explanation:
The libuv library in Node.js manages thread pools, allowing for efficient execution of non-blocking I/O operations. It handles tasks such as managing file systems, networking, and timers, contributing to the asynchronous nature of Node.js.

In Node.js, what is the purpose of the module.wrapper array, and how does it affect the scope of modules?

Option 1:
Modifies the module's behavior

Option 2:
Wraps code around a module

Option 3:
Defines global variables

Option 4:
Exports the module

Correct Response:
2.0

Explanation:
The module.wrapper array in Node.js is used to wrap code around a module. It allows developers to modify the behavior of a module by adding custom code before and after the module's code. This affects

the scope of modules by introducing additional logic during the module's execution.

Discuss how circular dependencies are handled in Node.js modules and the potential issues they can cause.

Option 1:
Automatically resolved

Option 2:
Result in a runtime error

Option 3:
Ignored by the Node.js runtime

Option 4:
Handled by using try-catch

Correct Response:
2.0

Explanation:
Circular dependencies in Node.js modules can result in a runtime error. Node.js doesn't automatically resolve them. Developers need to be cautious and refactor code to avoid circular dependencies as

they can lead to hard-to-debug issues and impact the application's performance.

In Node.js, the _____ function is used to import modules.

Option 1:
load

Option 2:
import

Option 3:
require

Option 4:
include

Correct Response:
3.0

Explanation:
In Node.js, the correct function to import modules is the require function. It allows you to include external modules in your code, making their functionality accessible. The load and import options are not valid in the context of Node.js module loading. The include option is also incorrect as it is not used in Node.js for module imports.

The Event Loop in Node.js operates on a _____ model, ensuring asynchronous operations do not block the main thread.

Option 1:
Single-threaded

Option 2:
Multi-threaded

Option 3:
Dual-threaded

Option 4:
Quad-threaded

Correct Response:
1.0

Explanation:
The Event Loop in Node.js operates on a single-threaded model. This means that it uses a single main thread to handle all incoming requests and execute asynchronous operations. This design ensures that asynchronous tasks do not block the main thread, allowing for

efficient handling of concurrent operations. The options b), c), and d) are incorrect as Node.js primarily relies on a single-threaded event-driven architecture.

Node.js modules that are loaded using require are cached as _____ objects.

Option 1:
Singleton

Option 2:
Prototype

Option 3:
Instance

Option 4:
Module

Correct Response:
1.0

Explanation:
Node.js modules loaded using require are cached as Singleton objects. This means that, upon the first require call for a module, it is loaded and cached. Subsequent require calls for the same module return the cached instance, promoting the concept of a Singleton pattern. The options b), c), and d) are incorrect as they do not accurately describe the caching behavior of Node.js modules.

The Node.js Event Loop is part of the _____ library, which provides asynchronous I/O capabilities.

Option 1:
V8

Option 2:
Libuv

Option 3:
Request

Option 4:
Async

Correct Response:
2.0

Explanation:
In Node.js, the Event Loop is a crucial part of managing asynchronous operations. It is implemented in the Libuv library, which handles the event-driven architecture, enabling efficient I/O operations.

When exporting a single entity such as a function or object in Node.js, you should use _____ instead of exports.

Option 1:
module.exports

Option 2:
global.exports

Option 3:
local.exports

Option 4:
single.exports

Correct Response:
1.0

Explanation:
While exports is commonly used, when exporting a single entity in Node.js, it is recommended to use module.exports for clarity and consistency. This ensures that you are exporting the intended object or function.

To enable debugging of lazy-loaded internal modules in Node.js, you can use the _____ environment variable.

Option 1:
DEBUG

Option 2:
TRACE

Option 3:
NODE_DEBUG

Option 4:
VERBOSE_DEBUG

Correct Response:
3.0

Explanation:
The NODE_DEBUG environment variable in Node.js allows you to enable debugging output for specific internal modules, even those that are loaded lazily. This can be helpful for troubleshooting and gaining insights into module interactions.

You are debugging a performance issue in a Node.js application and notice that asynchronous callbacks are executing much later than expected. What areas of the Event Loop would you investigate?

Option 1:
Callback Queue

Option 2:
Microtask Queue

Option 3:
Event Loop Phases

Option 4:
Timers Queue

Correct Response:
3.0

Explanation:
When debugging delayed callbacks, it's crucial to understand the different phases of the Event Loop, as events move through the Callback Queue, Microtask Queue, and other phases. Investigating the Event Loop can help identify the bottleneck.

A module in your Node.js application is not behaving as expected, and you suspect it's because of a version mismatch caused by module caching. How would you confirm and resolve this issue?

Option 1:
Clearing the Module Cache

Option 2:
Inspecting the Node.js Version

Option 3:
Using npm update

Option 4:
Verifying Node.js Documentation

Correct Response:
1.0

Explanation:
Clearing the module cache is a common step to resolve issues related to version mismatches caused by module caching. This ensures that the latest version of the module is loaded and can resolve unexpected behavior.

You've been tasked with optimizing a Node.js application for better performance. What strategies would you employ to reduce the workload on the Event Loop?

Option 1:
Implementing Worker Threads

Option 2:
Increasing Event Loop Capacity

Option 3:
Minimizing Blocking Code

Option 4:
Utilizing Asynchronous I/O Operations

Correct Response:
4.0

Explanation:
To optimize Node.js performance, it's crucial to minimize blocking code and leverage asynchronous I/O operations. This reduces the workload on the Event Loop, enhancing overall application performance. Implementing worker threads is another strategy to parallelize tasks.

In Node.js, which global object is used to delay the execution of a function?

Option 1:
setTimeout

Option 2:
setInterval

Option 3:
setImmediate

Option 4:
process.nextTick

Correct Response:
1.0

Explanation:
In Node.js, the setTimeout function is used to delay the execution of a function. It schedules a one-time execution of a function after a specified delay in milliseconds.

What global method in Node.js can be used to terminate the current process?

Option 1:
process.exit()

Option 2:
exit()

Option 3:
terminate()

Option 4:
endProcess()

Correct Response:
1.0

Explanation:
The process.exit() method in Node.js is used to terminate the current process. It allows you to exit from the application with an optional exit code.

Which Node.js global object is used to handle uncaught exceptions?

Option 1:
process.on('uncaughtException')

Option 2:
global.uncaughtExceptionHandler

Option 3:
process.catchException

Option 4:
process.handleException

Correct Response:
1.0

Explanation:
The correct option is process.on('uncaughtException'). This event is emitted when an unhandled exception occurs in the Node.js process, providing an opportunity to perform cleanup or log the error.

How does Node.js achieve non-blocking asynchronous execution under the hood?

Option 1:
Event Loop

Option 2:
Callbacks

Option 3:
Promises

Option 4:
Generators

Correct Response:
1.0

Explanation:
Node.js achieves non-blocking asynchronous execution through its event-driven architecture and the event loop. The event loop continually checks the message queue for new events and executes them in a non-blocking manner, allowing for efficient handling of concurrent operations.

What is the purpose of the global Buffer class in Node.js?

Option 1:
Buffering data

Option 2:
Manipulating binary data

Option 3:
Handling HTTP requests

Option 4:
Implementing middleware

Correct Response:
2.0

Explanation:
The global Buffer class in Node.js is used for handling binary data. It provides methods to manipulate raw binary data, making it crucial for tasks such as reading from or writing to streams, interacting with file systems, and handling network protocols.

Which global object provides information about, and control over, the current Node.js process?

Option 1:
process

Option 2:
global

Option 3:
context

Option 4:
system

Correct Response:
1.0

Explanation:
The global process object in Node.js provides information and control over the current process. It allows access to environment variables, command-line arguments, and provides methods to interact with the process, such as terminating it or changing its behavior.

Explain the role of the process.nextTick() method in Node.js's asynchronous execution.

Option 1:
It schedules a callback to be invoked in the next iteration of the event loop.

Option 2:
It delays the execution of the callback until all I/O events are processed.

Option 3:
It executes the callback after a specific timeout period.

Option 4:
It is used for synchronous execution of code.

Correct Response:
1.0

Explanation:
The process.nextTick() method is used to schedule a callback function to be invoked in the next iteration of the event loop. This allows certain tasks to be executed after the current operation but before the event loop continues. It is often used to defer the

execution of a function to the next tick of the event loop, making it a key tool in managing asynchronous operations in Node.js.

What is the difference between global in Node.js and window in the browser?

Option 1:
global refers to the global object in Node.js, providing global scope for variables and functions in a Node.js application.

Option 2:
window is the global object in browser environments, representing the global scope for variables and functions in a web page.

Option 3:
Both global and window refer to the same global object in different environments.

Option 4:
window is specific to client-side JavaScript, whereas global is used in server-side Node.js.

Correct Response:
2.0

Explanation:
In Node.js, the global object represents the global scope, providing a context for variables and functions throughout the application. On the other hand, in browser environments, the window object serves a similar purpose, acting as the global object for client-side JavaScript. It's important to distinguish between the two, as they are specific to their respective environments.

How can Node.js's setImmediate() be crucial in implementing complex asynchronous flows?

Option 1:
setImmediate() is similar to setTimeout(), but the callback is executed in the next iteration of the event loop.

Option 2:
It is used for blocking I/O operations to ensure immediate execution of the callback.

Option 3:
It is primarily used for scheduling periodic tasks at regular intervals.

Option 4:
It executes the callback before any other I/O events in the next event loop cycle.

Correct Response:
1.0

Explanation:
The setImmediate() function in Node.js is designed to execute the provided callback in the next iteration of the event loop, right after the current event loop cycle. This makes it useful for scenarios where immediate execution is required, and it plays a crucial role in

implementing complex asynchronous flows by ensuring that certain tasks are prioritized and executed without unnecessary delay.

In Node.js, callbacks that take an error object as their first argument follow the _____ pattern.

Option 1:
Observer

Option 2:
Callback

Option 3:
Promise

Option 4:
Singleton

Correct Response:
2.0

Explanation:
In Node.js, the convention for callbacks is to have the error as the first argument, known as the Callback pattern. This allows developers to handle errors in an organized way.

The global _____ object in Node.js is used to emit and listen for events.

Option 1:
Event

Option 2:
Process

Option 3:
EventEmitter

Option 4:
Global

Correct Response:
3.0

Explanation:
The global EventEmitter object in Node.js is used for handling events. It allows objects to emit and listen for custom events.

To read the current version of Node.js running, one would use process._____.

Option 1:
version

Option 2:
platform

Option 3:
versions

Option 4:
versioning

Correct Response:
1.0

Explanation:
To obtain the current version of Node.js, you would use process.version. This property provides information about the Node.js version running the script.

Node.js leverages the _____ module internally for asynchronous programming.

Option 1:
fs

Option 2:
http

Option 3:
events

Option 4:
async

Correct Response:
3.0

Explanation:
Node.js internally uses the 'events' module to handle asynchronous programming. This module provides an EventEmitter class that allows you to bind and listen for custom events. It is a core part of Node.js for handling asynchronous operations.

The process._____ method is an asynchronous version of console.log that prints to stdout.

Option 1:
log

Option 2:
stdout

Option 3:
print

Option 4:
nextTick

Correct Response:
4.0

Explanation:
The process.nextTick() method is an asynchronous version of console.log. It is used to defer the execution of a function until the next iteration of the event loop. This helps in executing the function asynchronously, making it suitable for scenarios where you want to delay the execution.

Node.js provides a global object named _____, which is a small wrapper around native JavaScript timer functions like setTimeout and setInterval.

Option 1:
process

Option 2:
global

Option 3:
timer

Option 4:
setTimeout

Correct Response:
2.0

Explanation:
The global object in Node.js is a global namespace object. It provides

important functions and properties, and it acts as a container for global variables. It is not specific to timers, but it is the correct answer in this context as it is a global object in Node.js.

A Node.js application needs to execute a series of asynchronous database queries in a specific order. How would you implement this while taking advantage of Node.js's asynchronous features?

Option 1:
Use Promises and the async/await syntax

Option 2:
Utilize callbacks and nesting

Option 3:
Employ the setImmediate function for precise ordering

Option 4:
Use synchronous functions and handle errors synchronously

Correct Response:
1.0

Explanation:
In Node.js, leveraging Promises and the async/await syntax ensures readable and sequential asynchronous code. This pattern enhances code maintainability and allows the ordered execution of asynchronous tasks. Callbacks, especially nested ones, can lead to callback hell, making the code less readable and maintainable.

You're optimizing a Node.js application and notice that the use of global variables is causing memory leaks. How would you identify and solve this problem?

Option 1:
Utilize the global object cautiously

Option 2:
Implement garbage collection strategies

Option 3:
Use the process.memoryUsage() method to identify memory usage

Option 4:
Refactor code to use local variables

Correct Response:
4.0

Explanation:
Global variables in Node.js can lead to memory leaks. Refactoring

code to use local variables helps prevent this issue, as local variables have a limited scope and are automatically garbage-collected. It's important to avoid unnecessary use of the global object and regularly monitor memory usage using tools like process.memoryUsage().

When working with a large global state in a Node.js application, what strategies would you use to ensure that state is managed efficiently and safely across asynchronous operations?

Option 1:
Use a state management library like Redux

Option 2:
Implement mutex locks to control access to the global state

Option 3:
Break down the global state into smaller, more manageable parts

Option 4:
Utilize Node.js cluster module for parallel processing

Correct Response:
3.0

Explanation:
Breaking down a large global state into smaller parts makes it more manageable and avoids potential bottlenecks. Using a state management library like Redux can also help organize and manage state efficiently. Avoiding mutex locks is crucial, as they can lead to deadlocks and negatively impact performance.

What will Node.js do by default if an error is not caught by any error handler?

Option 1:
Print to console

Option 2:
Crash the process

Option 3:
Log to a separate file

Option 4:
Retry the operation

Correct Response:
2.0

Explanation:
In Node.js, if an unhandled error occurs, it will crash the process to prevent potential issues from spreading. Proper error handling is essential.

In Node.js, how do you parse a JSON string to convert it into a JavaScript object?

Option 1:
JSON.parse()

Option 2:
parseJSON()

Option 3:
convertToJSON()

Option 4:
stringifyJSON()

Correct Response:
1.0

Explanation:
The JSON.parse() method in Node.js is used to parse a JSON string and convert it into a JavaScript object.

Which object is used in Node.js to represent errors and can be thrown as an exception?

Option 1:
ErrorObject()

Option 2:
ExceptionObject()

Option 3:
ErrorInstance()

Option 4:
Error()

Correct Response:
4.0

Explanation:
In Node.js, the Error object is used to represent errors, and instances of this object can be thrown as exceptions.

When working with Promises in Node.js, how are errors propagated to the caller?

Option 1:
Through the catch method

Option 2:
Via the then method

Option 3:
Using the finally block

Option 4:
Automatically to the next catch in the chain

Correct Response:
1.0

Explanation:
In Node.js, errors in Promises are typically handled through the catch method, allowing developers to handle and propagate errors in asynchronous operations effectively. The then method is used for handling successful promises. The finally block is used for code that needs to be executed regardless of whether the Promise is resolved or rejected.

In Node.js, which method is typically used to stringify an object to JSON, considering error handling?

Option 1:
JSON.parse()

Option 2:
JSON.stringify()

Option 3:
stringifyJSON()

Option 4:
parseJSON()

Correct Response:
2.0

Explanation:
The JSON.stringify() method in Node.js is commonly used to convert an object to a JSON string. It also provides options for customization and error handling. JSON.parse() is used for parsing a JSON string to an object. Options 3 and 4 are not standard JSON methods.

What is the purpose of the try...catch statement in Node.js error handling?

Option 1:
To force the code to execute in all cases

Option 2:
To handle asynchronous operations

Option 3:
To gracefully handle and recover from errors

Option 4:
To terminate the program on error

Correct Response:
3.0

Explanation:
The try...catch statement in Node.js is used for graceful error handling and recovery. It allows developers to wrap potentially error-prone code in a try block and specify how to handle any resulting exceptions in the catch block, preventing the program from crashing.

How does the domain module in Node.js help with error handling, and what are its drawbacks?

Option 1:
Using try-catch with domains.

Option 2:
Isolating I/O operations.

Option 3:
Providing a way to handle multiple asynchronous operations.

Option 4:
Wrapping code in a domain to catch errors.

Correct Response:
4.0

Explanation:
The domain module in Node.js was introduced to handle errors by encapsulating code within a domain. It provides an event-driven architecture for error handling. However, it has drawbacks, such as not being a silver bullet for all error scenarios and being deprecated in recent Node.js versions. This question assesses knowledge of error handling using the domain module and awareness of its limitations.

Explain how you would handle errors in an asynchronous Node.js callback pattern.

Option 1:
Using synchronous try-catch blocks.

Option 2:
Utilizing the 'error' event emitter.

Option 3:
Using promises and the .catch() method.

Option 4:
Employing the 'async/await' syntax.

Correct Response:
3.0

Explanation:
Handling errors in an asynchronous Node.js callback pattern involves using promises and the .catch() method. This approach allows for chaining asynchronous operations and catching errors in a centralized manner. Understanding the 'async/await' syntax is crucial for managing asynchronous code fluently. This question assesses proficiency in error handling within the context of asynchronous operations in Node.js.

What are the best practices for logging errors and exceptions in a Node.js application?

Option 1:
Logging all errors to the console.

Option 2:
Using a custom logger with log levels.

Option 3:
Ignoring errors to avoid cluttering logs.

Option 4:
Utilizing the 'debug' module for logging.

Correct Response:
2.0

Explanation:
Best practices for logging errors in a Node.js application include using a custom logger with log levels to differentiate between errors and debugging information. This approach enhances the maintainability of logs. Logging to the console is generally discouraged for production environments. This question evaluates knowledge of logging best practices in a Node.js context.

When an error occurs in Node.js, an 'error' event is emitted, which can be handled by attaching an _____ listener to an EventEmitter object.

Option 1:
error'

Option 2:
'onError'

Option 3:
'handleError'

Option 4:
'eventHandler'

Correct Response:
1.0

Explanation:
In Node.js, when an error occurs, the 'error' event is emitted, and we

handle it by attaching an 'error' listener using the 'on' method. Hence, option a ('error') is the correct choice for this scenario.

To safely handle JSON parsing errors, wrap the JSON.parse call within a _____ block.

Option 1:
try-catch

Option 2:
error-catch

Option 3:
parse-catch

Option 4:
exception-catch

Correct Response:
1.0

Explanation:
In JavaScript, particularly in Node.js, wrapping JSON.parse in a try-catch block allows us to handle parsing errors gracefully. Option a ('try-catch') is the correct choice for achieving this error handling mechanism.

A common practice in Node.js to ensure JSON errors are caught is to use _____ when writing JSON to a response stream.

Option 1:
res.json

Option 2:
res.send

Option 3:
res.writeJson

Option 4:
res.jsonError

Correct Response:
1.0

Explanation:
In Node.js, using res.json is a common practice to ensure that JSON errors are caught and handled properly when writing JSON to a response stream. Option a ('res.json') is the correct choice for this context.

Advanced error handling in Node.js may involve creating custom error objects by extending the _____ class.

Option 1:
Error

Option 2:
Exception

Option 3:
EventEmitter

Option 4:
Process

Correct Response:
1.0

Explanation:
In Node.js, advanced error handling can be achieved by creating custom error objects that extend the built-in Error class. This allows developers to have more control and specificity in error handling.

To prevent a Node.js process from crashing upon an unhandled promise rejection, one should listen for the _____ event on the process object.

Option 1:
exit

Option 2:
unhandledRejection

Option 3:
rejectionHandled

Option 4:
processError

Correct Response:
2.0

Explanation:
Listening for the unhandledRejection event on the process object

allows developers to handle unhandled promise rejections, preventing the process from crashing.

Node.js provides a global _____ method that can be used to catch uncaught exceptions, but relying on it is not recommended as a best practice for error handling.

Option 1:
catchException

Option 2:
handleException

Option 3:
uncaughtException

Option 4:
processException

Correct Response:
3.0

Explanation:
The uncaughtException method in Node.js is a global exception handler. However, relying on it is not recommended as it can lead to unpredictable behavior. It's better to handle exceptions in a more controlled manner using other error handling techniques.

You're designing an API in Node.js and need to ensure that JSON parsing errors from client input do not crash the server. Describe the error handling strategy you would implement.

Option 1:
Implement a global error handler middleware

Option 2:
Use try-catch blocks for JSON parsing

Option 3:
Validate input before JSON parsing

Option 4:
Log errors to a centralized system

Correct Response:
1.0

Explanation:
To handle JSON parsing errors without crashing the server, implementing a global error handler middleware is crucial. This middleware can catch errors, including JSON parsing errors, and respond appropriately without affecting the entire server. Using try-catch blocks for JSON parsing, validating input before parsing, and logging errors to a centralized system are good practices but might not directly prevent server crashes due to JSON parsing errors.

A Node.js service you're maintaining has been experiencing intermittent crashes, and logs suggest unhandled promise rejections. How would you approach solving this issue?

Option 1:
Use Promise.catch() to handle unhandled promise rejections

Option 2:
Increase the server's memory allocation

Option 3:
Use async/await instead of Promises

Option 4:
Implement a global unhandledRejection event listener

Correct Response:
4.0

Explanation:
To address unhandled promise rejections causing intermittent crashes, implementing a global unhandledRejection event listener is crucial. This allows capturing unhandled promise rejections globally and taking appropriate actions. Using Promise.catch(), increasing server memory, and using async/await are good practices but might not directly solve the issue of intermittent crashes due to unhandled promise rejections.

While developing a Node.js application, you notice that certain errors are not being logged. What are some mechanisms you could use to ensure that all errors are caught and logged?

Option 1:
Implement a centralized error logging service

Option 2:
Use a process manager to monitor the application

Option 3:
Add logging statements strategically

Option 4:
Use process.on('uncaughtException') to catch unhandled exceptions

Correct Response:
1.0

Explanation:
To ensure that all errors, including unhandled exceptions, are caught and logged, implementing a centralized error logging service is crucial. This allows consistent and centralized error logging. Using a process manager, adding logging statements strategically, and using process.on('uncaughtException') are good practices but might not cover all types of errors and exceptions.

What is npm primarily used for in a Node.js environment?

Option 1:
Node Package Manager

Option 2:
Network Performance Monitoring

Option 3:
New Programming Method

Option 4:
Node Process Manager

Correct Response:
1.0

Explanation:
npm (Node Package Manager) is primarily used for managing and installing packages in a Node.js environment. It helps in dependency management, version control, and easy integration of third-party libraries into Node.js projects.

Explain what a buffer is in Node.js and why it is used.

Option 1:
Temporary storage of binary data

Option 2:
A tool for rendering graphics

Option 3:
A database management system

Option 4:
A module for handling HTTP requests

Correct Response:
1.0

Explanation:
In Node.js, a buffer is used as temporary storage for binary data. It is particularly useful when working with streams, file systems, and other scenarios where raw binary data needs to be processed. Buffers provide a way to work with binary data directly, without the need for encoding and decoding.

How would you use npm to install a package globally on your system?

Option 1:
npm install -g <package-name>

Option 2:
npm add -g <package-name>

Option 3:
npm global install <package-name>

Option 4:
npm link <package-name>

Correct Response:
1.0

Explanation:
To install a package globally using npm, the correct command is npm install -g <package-name>. This installs the specified package globally on your system, making it accessible from any directory in the command line.

What file is automatically updated when you install a new npm package in your Node.js project?

Option 1:
package.json

Option 2:
package-lock.json

Option 3:
npm.json

Option 4:
dependencies.json

Correct Response:
2.0

Explanation:
When you install a new npm package, the package-lock.json file is automatically updated. This file keeps track of the exact versions of dependencies and their transitive dependencies, ensuring consistency across different installations.

How do Node.js streams differ from buffers, and when would you use a stream over a buffer?

Option 1:
Streams provide a way to read or write data piece by piece, making them suitable for handling large amounts of data or handling data in real-time. Buffers, on the other hand, are used to handle fixed-size chunks of binary data. Streams are preferable when dealing with large files or continuous data, while buffers are suitable for smaller, fixed-size operations.

Option 2:
Buffers are used for asynchronous I/O operations, while streams are used for synchronous I/O operations. Buffers provide a way to store and manipulate binary data directly, while streams are limited to text-based data.

Option 3:
Buffers are only used for reading data, while streams are used for both reading and writing data. Buffers are preferable for real-time applications, while streams are suitable for batch processing.

Option 4:
Buffers and streams are interchangeable and can be used interchangeably based on personal preference.

Correct Response:
1.0

Explanation:
Node.js Streams and Buffers

What command would you use to update all the npm packages in a Node.js project to their latest version?

Option 1:
npm update

Option 2:
npm upgrade

Option 3:
npm install -g

Option 4:
npm install --save

Correct Response:
2.0

Explanation:
The correct command to update all npm packages to their latest versions is npm upgrade. This command updates the packages as per the version specified in the package.json file. Using npm update might not update to the latest versions as it respects the version constraints specified in the package.json.

Can you describe what semantic versioning is and how npm uses it?

Option 1:
It represents versions using major, minor, and patch numbers.

Option 2:
It ensures backward compatibility with new features.

Option 3:
It utilizes caret (^) and tilde (~) for version ranges.

Option 4:
It is only applicable to npm packages.

Correct Response:
1.0

Explanation:
Semantic versioning, also known as SemVer, uses major, minor, and patch numbers to convey meaning about the underlying code changes. Caret (^) and tilde (~) are used in npm's package.json to specify version ranges, ensuring compatibility while allowing updates.

What are Duplex streams in Node.js, and how do they differ from Readable and Writable streams?

Option 1:
Duplex streams are unidirectional, either readable or writable.

Option 2:
Readable streams are for reading input, writable streams for writing output.

Option 3:
Duplex streams allow both reading and writing.

Option 4:
Writable streams can only be read.

Correct Response:
3.0

Explanation:
Duplex streams in Node.js enable both reading and writing, combining features of Readable and Writable streams. They provide a two-way communication channel. Readable streams are for input, while Writable streams are for output, but Duplex streams offer bidirectional capabilities.

Explain the purpose of the package-lock.json file in npm and how it's used in dependency management.

Option 1:
It contains package metadata.

Option 2:
It locks down the version of installed packages.

Option 3:
It is optional and doesn't affect dependency resolution.

Option 4:
It is used for storing package documentation.

Correct Response:
2.0

Explanation:
The package-lock.json file in npm locks down the version of installed packages, ensuring consistency across environments. It is crucial for reproducibility in dependency management. While package metadata is stored in package.json, package-lock.json provides specific version information for each installed package.

Node.js Buffers are typically used to handle raw binary data for _____ operations.

Option 1:
file I/O

Option 2:
network

Option 3:
string manipulation

Option 4:
binary

Correct Response:
4.0

Explanation:
Buffers in Node.js are designed to handle raw binary data, making them suitable for operations involving binary data, such as reading from or writing to files, handling network protocols, etc.

In Node.js, the _____ module is used to work with file streams.

Option 1:
fs

Option 2:
filestream

Option 3:
stream

Option 4:
file

Correct Response:
3.0

Explanation:
The stream module in Node.js is used to work with file streams. Streams provide an efficient way to read or write data in chunks, making them particularly useful for handling large files or data streams.

The npm command to list the outdated packages in your project is npm _____.

Option 1:
outdated

Option 2:
update

Option 3:
check-update

Option 4:
upgrade

Correct Response:
1.0

Explanation:
The correct option is outdated. The npm outdated command is used to display a list of outdated packages in your project, along with their current and latest versions.

To convert a stream of binary data into a stream of objects in Node.js, you would use a _____ stream.

Option 1:
Buffer

Option 2:
Transform

Option 3:
Readable

Option 4:
Writable

Correct Response:
2.0

Explanation:
The correct option is Transform. Transform streams in Node.js are Duplex streams where the output is computed based on the input. They are often used for tasks like data manipulation during a stream.

When you want to install a package without adding it to your package.json, you use the npm install <package_name> -- _____ command.

Option 1:
save

Option 2:
global

Option 3:
local

Option 4:
production

Correct Response:
2.0

Explanation:
The correct option is global. When you want to install a package

globally without adding it to your project's dependencies, you use the npm install <package_name> --global or npm install -g <package_name> command.

You're working on a Node.js project and need to ensure that the exact versions of all dependencies are installed on every developer's machine. Which npm feature would you use?

Option 1:
npm init

Option 2:
npm install

Option 3:
npm update

Option 4:
npm shrinkwrap

Correct Response:
4.0

Explanation:
The correct option is npm shrinkwrap. This command generates a npm-shrinkwrap.json file that locks down the exact versions of your project's dependencies, ensuring consistent installations across different environments.

A Node.js application is experiencing memory leaks while handling large files. How might buffers and streams be utilized to address this problem?

Option 1:
Buffers provide a fixed-size memory allocation

Option 2:
Buffers allow asynchronous reading of files

Option 3:
Streams enable efficient handling of data

Option 4:
Streams are used for data persistence

Correct Response:
3.0

Explanation:
The correct option is Streams enable efficient handling of data.

Buffers and streams in Node.js can be used to efficiently handle large files by reading and writing data in chunks, reducing memory usage and addressing potential memory leaks.

You are creating a CLI tool and need to use npm to install your tool globally so that it can be used in any terminal session. Which command would you use?

Option 1:
npm install -g

Option 2:
npm install --global

Option 3:
npm global install

Option 4:
npm link

Correct Response:
1.0

Explanation:
The correct option is npm install -g. This command installs the

package globally, making it accessible from any terminal session. The -g flag stands for global installation, ensuring the CLI tool is available system-wide.

In Node.js, which method of the File System module is used to read the contents of a file asynchronously?

Option 1:
fs.readfile()

Option 2:
fs.read()

Option 3:
fs.readFile()

Option 4:
fs.readSync()

Correct Response:
3.0

Explanation:
The correct method to read the contents of a file asynchronously in Node.js is fs.readFile(). This method takes the file path and a callback function as parameters. The callback function is called with the error, if any, and the data read from the file. Using fs.readfile() or fs.read() is incorrect, as they are not valid methods for asynchronous file reading. fs.readSync() is a synchronous version, which is not recommended for asynchronous file operations.

What is the primary method provided by the HTTP module to create an HTTP server?

Option 1:
http.createServer()

Option 2:
http.startServer()

Option 3:
http.newServer()

Option 4:
http.handleServer()

Correct Response:
1.0

Explanation:
The primary method to create an HTTP server in Node.js is http.createServer(). This method takes a callback function as a parameter, which is called for each request received by the server. Options like http.startServer(), http.newServer(), and http.handleServer() are not valid methods for creating an HTTP server in Node.js. They do not exist in the HTTP module.

How can you make an HTTPS GET request using the HTTPS module in Node.js?

Option 1:
https.request('GET', options)

Option 2:
https.get(options, callback)

Option 3:
https.fetch(options)

Option 4:
https.sendRequest('GET', options)

Correct Response:
2.0

Explanation:
To make an HTTPS GET request using the HTTPS module in Node.js, you should use the https.get(options, callback) method. This method takes the options for the request and a callback function that is called when the response is received. The other options provided are incorrect. https.request('GET', options) is not the correct syntax for making a GET request, and https.fetch(options) and https.sendRequest('GET', options) are not valid methods for making HTTPS requests in Node.js.

Which File System module method in Node.js do you use to watch for changes in the filesystem?

Option 1:
fs.watch()

Option 2:
fs.watchFile()

Option 3:
fs.readFile()

Option 4:
fs.createReadStream()

Correct Response:
1.0

Explanation:
The correct option is fs.watch(). It is used to asynchronously watch changes in the file or directory. fs.watchFile() is used for polling, and the other options are unrelated to file watching in real-time.

When using the HTTP module to create a server, how can you listen for POST requests and retrieve the body of the request?

Option 1:
request.on('data')

Option 2:
request.body

Option 3:
request.read()

Option 4:
request.on('end')

Correct Response:
1.0

Explanation:
The correct option is request.on('data'). It allows you to listen for data events and retrieve the body of a POST request in chunks. The other options are either incorrect or not directly related to handling POST requests.

What is the purpose of the ca option when creating an HTTPS server using the HTTPS module?

Option 1:
Certificate Authority (Ccertificate

Option 2:
Client Authentication

Option 3:
Cipher Algorithm

Option 4:
Connection Adapter

Correct Response:
1.0

Explanation:
The correct option is Certificate Authority (CA) certificate. The ca option in the HTTPS module is used to specify the CA certificate that is trusted for verifying the remote peer's certificate. Other options are not relevant to the ca option.

How do you ensure atomicity when renaming a file in Node.js using the File System module?

Option 1:
Use the fs.rename() method

Option 2:
Use fs.unlinkSync() and fs.renameSync()

Option 3:
Use fs.move() from the 'fs-extra' library

Option 4:
Implement a custom atomic rename function

Correct Response:
3.0

Explanation:
When renaming a file atomically in Node.js, one approach is to use the fs-extra library's fs.move() method. This method ensures atomicity by using the underlying rename() system call when available, falling back to a copy and delete strategy in cases where rename is not atomic. This helps prevent race conditions and ensures that the file is either fully moved or not moved at all. The use of

fs.unlinkSync() and fs.renameSync() can lead to non-atomic operations.

What are the differences in handling backpressure between the HTTP and HTTPS modules when streaming large files?

Option 1:
HTTPS handles backpressure better than HTTP

Option 2:
HTTP and HTTPS handle backpressure the same

Option 3:
HTTP handles backpressure better than HTTPS

Option 4:
Backpressure is not relevant in streaming

Correct Response:
1.0

Explanation:
When streaming large files, HTTPS generally handles backpressure better than HTTP. This is because HTTP lacks flow control mechanisms that are present in HTTPS. HTTPS, being a secure protocol, enforces flow control to avoid overwhelming the client with

data. In HTTP, if the client is slower than the server, data may be buffered in memory, leading to potential out-of-memory issues. Understanding these differences is crucial for efficient handling of streaming scenarios.

In Node.js, how can you use the HTTP module to implement a reverse proxy server?

Option 1:
Use the http.createServer() method

Option 2:
Use the http.request() method

Option 3:
Use the http.forward() method

Option 4:
Implementing a reverse proxy is not possible

Correct Response:
2.0

Explanation:
To implement a reverse proxy server in Node.js using the HTTP module, you can use the http.request() method. This method is used to make an outgoing HTTP request to a specified server. By handling incoming requests on your server and forwarding them to another server using http.request(), you can create a reverse proxy. This allows your Node.js server to act as an intermediary between clients and other servers, forwarding requests and responses as needed.

Understanding this process is essential for building scalable and modular systems.

The method to asynchronously write data to a file in the File System module is _____.

Option 1:
fs.writeFile()

Option 2:
fs.appendFile()

Option 3:
fs.write()

Option 4:
fs.writeAsync()

Correct Response:
2.0

Explanation:
In Node.js, the fs.appendFile() method is used to asynchronously append data to a file. It does not replace the file but adds content to the end. This is useful for log files and other scenarios where you want to add data without overwriting existing content.

To parse the URL of an incoming request in an HTTP server, you can use the _____ method from the URL module.

Option 1:
url.parse()

Option 2:
url.stringify()

Option 3:
url.format()

Option 4:
url.extract()

Correct Response:
1.0

Explanation:
In Node.js, the url.parse() method from the URL module is used to parse the URL of an incoming request in an HTTP server. It provides a convenient way to extract various components of the URL, such as the pathname, query parameters, and more.

When creating a server using the HTTPS module, the _____ option is used to provide the private key.

Option 1:
key

Option 2:
privateKey

Option 3:
secureKey

Option 4:
secretKey

Correct Response:
2.0

Explanation:
When creating an HTTPS server in Node.js, the privateKey option is used to provide the private key. This private key is essential for establishing a secure connection over HTTPS. It ensures the confidentiality and integrity of data exchanged between the server and clients.

To handle file paths correctly across different operating systems, you should use the _____ module in Node.js.

Option 1:
fs

Option 2:
path

Option 3:
url

Option 4:
querystring

Correct Response:
2.0

Explanation:
In Node.js, the path module provides a way to work with file paths in a cross-platform manner, handling the differences in file path conventions between operating systems.

The _____ event is emitted when the request has been fully read in an HTTP server.

Option 1:
data

Option 2:
end

Option 3:
request

Option 4:
response

Correct Response:
2.0

Explanation:
The end event in Node.js is emitted when the entire request has been received. It signifies that no more data will be written to the response, indicating the completion of the request processing.

In Node.js, a custom HTTPS agent with specific keep-alive behavior can be created using the _____ option.

Option 1:
agent

Option 2:
keepAlive

Option 3:
httpsAgent

Option 4:
keepAliveAgent

Correct Response:
3.0

Explanation:
The httpsAgent option in Node.js allows you to specify a custom HTTPS agent with specific keep-alive behavior, providing control over how connections are reused and managed in HTTPS requests.

You are tasked with increasing the file upload size limit of your Node.js HTTP server. Which part of the server would you modify to accommodate larger files?

Option 1:
Increase the maxPayload property in the server configuration.

Option 2:
Adjust the maxBodySize setting in the server middleware.

Option 3:
Modify the uploadLimit attribute in the HTML form.

Option 4:
Update the limit option in the body-parser middleware.

Correct Response:
1.0

Explanation:
To accommodate larger file uploads in Node.js, you would increase

the maxPayload property in the server configuration. This property defines the maximum size (in bytes) allowed in the payload of a request. Incorrect options do not directly control the file upload size limit in a Node.js server.

Your Node.js application needs to serve both HTTP and HTTPS traffic. How would you structure your code to handle both protocols efficiently?

Option 1:
Implement separate servers for HTTP and HTTPS using different ports.

Option 2:
Use the express-sslify middleware to enforce HTTPS.

Option 3:
Utilize a reverse proxy like Nginx to handle protocol routing.

Option 4:
Use the built-in http and https modules with a shared codebase.

Correct Response:
4.0

Explanation:
To efficiently handle both HTTP and HTTPS traffic in Node.js, you would use the built-in http and https modules with a shared codebase. These modules allow you to create servers for both protocols using the same application logic. Other options may work but are not as efficient or standard.

During the deployment of a Node.js HTTPS server, you encounter an error stating that the SSL certificate is not trusted. What are the potential causes, and how would you resolve it?

Option 1:
The certificate is self-signed; import it into the trusted store.

Option 2:
Ensure the certificate is issued by a recognized Certificate Authority.

Option 3:
Check if the system date and time are correct.

Option 4:
Disable SSL verification for testing purposes.

Correct Response:
2.0

Explanation:
If the SSL certificate is not trusted, it could be due to it being self-signed or not issued by a recognized Certificate Authority. To resolve, ensure the certificate is valid and issued by a trusted CA. Checking system date/time is essential. Disabling SSL verification should only be temporary for testing.

In Node.js, which object is commonly used as the base for implementing event-driven architecture?

Option 1:
EventEmitter

Option 2:
Buffer

Option 3:
Stream

Option 4:
Module

Correct Response:
1.0

Explanation:
In Node.js, the EventEmitter class is commonly used to implement event-driven architecture. It allows objects to emit and listen for events, facilitating communication between different parts of a program.

What method is used to listen for events emitted by an EventEmitter instance in Node.js?

Option 1:
on()

Option 2:
emit()

Option 3:
listen()

Option 4:
handle()

Correct Response:
1.0

Explanation:
The on() method is used to listen for events emitted by an EventEmitter instance in Node.js. It allows you to register a callback function to be executed when a specific event occurs.

Which Stream class in Node.js is used for reading data from a source in a continuous manner?

Option 1:
Readable

Option 2:
Writable

Option 3:
Duplex

Option 4:
Transform

Correct Response:
1.0

Explanation:
The Readable Stream class in Node.js is used for reading data from a source in a continuous manner. It provides an interface for reading data chunk by chunk, making it suitable for scenarios like reading large files or receiving data over a network.

How can you prevent memory leaks when listening for events in Node.js?

Option 1:
Detach listeners when they are no longer needed

Option 2:
Increase the event listener count

Option 3:
Use arrow functions for event callbacks

Option 4:
Never remove event listeners

Correct Response:
1.0

Explanation:
Memory Management in Node.js is crucial to prevent memory leaks. Detaching listeners when they are no longer needed ensures that resources are freed up properly. Incorrect options may lead to resource retention and potential leaks.

In Node.js, what is the purpose of the pipe() method in the Stream API?

Option 1:
To concatenate two streams

Option 2:
To redirect output to a file

Option 3:
To transfer data from one stream to another

Option 4:
To create a new stream

Correct Response:
3.0

Explanation:
The pipe() method in the Stream API is used to transfer data from one readable stream to a writable stream. It simplifies the process of handling data flow between streams, reducing boilerplate code. Options like concatenation or redirection are not the primary purposes of pipe()

What is the difference between readable.read() and readable.on('data', callback) in the Stream API?

Option 1:
read() reads all available data at once

Option 2:
on('data', callback) is triggered when data is available

Option 3:
read() is synchronous, while on('data', callback) is asynchronous

Option 4:
on('data', callback) reads a specific amount of data

Correct Response:
2.0

Explanation:
The on('data', callback) event is triggered when data is available to be read, providing a way to handle data asynchronously. On the other hand, readable.read() is used to manually read data from the stream, allowing more control over the reading process. It is not event-driven like on('data').

Explain how backpressure is managed in Node.js Streams.

Option 1:
By default, Node.js handles backpressure automatically.

Option 2:
Using the 'drain' event, a writable stream can be paused until the buffer is drained.

Option 3:
Backpressure is not applicable to Node.js Streams.

Option 4:
Implementing a custom backpressure handling mechanism.

Correct Response:
2.0

Explanation:
Node.js Streams and Backpressure

In Node.js, how does the EventEmitter's once() method differ from the on() method?

Option 1:
The once() method listens for an event only once, while on() listens for an event continuously.

Option 2:
on() is for asynchronous events, and once() is for synchronous events.

Option 3:
once() can handle multiple events simultaneously, while on() can handle only one event at a time.

Option 4:
There is no difference; they can be used interchangeably.

Correct Response:
1.0

Explanation:
EventEmitter and Event Handling

Describe a scenario where you would use a Transform stream in Node.js.

Option 1:
When you need to read data from a file.

Option 2:
Transform streams are not used in Node.js.

Option 3:
When you need to modify or manipulate data during streaming.

Option 4:
Only when working with binary data.

Correct Response:
3.0

Explanation:
Transform Streams

To create a custom EventEmitter, you extend the _____ class in Node.js.

Option 1:
EventEmitter

Option 2:
EventSource

Option 3:
EventListener

Option 4:
EventEmit

Correct Response:
1.0

Explanation:
In Node.js, to create a custom EventEmitter, you extend the EventEmitter class. This class provides the ability to emit and listen for events. Extending it allows you to create your own event emitter with custom events and behavior.

The _____ event is emitted when there is no more data to read in a stream.

Option 1:
'end'

Option 2:
'finish'

Option 3:
'complete'

Option 4:
'close'

Correct Response:
1.0

Explanation:
The 'end' event is emitted in a stream when there is no more data to read. It indicates that the entire data has been consumed, and no more 'data' events will be emitted. This event is commonly used to perform cleanup or trigger additional actions after reading all the data.

When handling streams, Node.js can automatically manage the flow of data and the pausing and resuming of streams through the _____ mechanism.

Option 1:
Buffering

Option 2:
FlowControl

Option 3:
Backpressure

Option 4:
Pipelining

Correct Response:
3.0

Explanation:
Node.js manages the flow of data and prevents overwhelming

consumers through the backpressure mechanism. When a readable stream is not being consumed as fast as it's producing data, backpressure is applied, causing the producer to pause until the consumer is ready. This ensures efficient memory usage and prevents potential application issues.

In a Node.js stream, the _____ event indicates that the underlying resource has closed and no more events will be emitted.

Option 1:
end

Option 2:
close

Option 3:
finish

Option 4:
complete

Correct Response:
1.0

Explanation:
The correct event is end. The 'end' event is emitted when there is no more data to be consumed from the stream, indicating the end of the stream. This is different from 'close', which is emitted when the underlying resource is closed.

A _____ stream in Node.js is a type of duplex stream where the output is computed from the input.

Option 1:
Transform

Option 2:
Readable

Option 3:
Writable

Option 4:
Duplex

Correct Response:
1.0

Explanation:
The correct stream type is Transform. A Transform stream is a duplex stream where the output is computed based on the input. It allows for data modification during the read and write operations.

The setImmediate() function within an event handler is often used to defer an action to the next iteration of the _____.

Option 1:
event loop

Option 2:
callback

Option 3:
promise

Option 4:
timer

Correct Response:
1.0

Explanation:
The correct answer is the event loop. The setImmediate() function is used to execute a callback after the current event loop cycle, allowing the event loop to complete its current cycle before executing the deferred action.

You've noticed that large amounts of data are being buffered and not processed quickly enough in your Node.js application. How would you address this issue with regards to the Stream API?

Option 1:
Increase the buffer size to accommodate more data without processing delay.

Option 2:
Implement the 'drain' event to handle backpressure and resume writing when the buffer is empty.

Option 3:
Use a readable stream to handle incoming data more efficiently.

Option 4:
Switch to synchronous processing to ensure real-time data handling.

Correct Response:
2.0

Explanation:
In the context of the Stream API in Node.js, the 'drain' event is crucial for handling backpressure. When a writable stream is buffering large amounts of data, the 'drain' event signals that the buffer is empty, allowing the application to resume writing. Increasing the buffer size (option a) does not solve the backpressure issue, and using a readable stream (option c) is not directly related to handling backpressure. Synchronous processing (option d) can block the event loop, causing performance issues.

A Node.js application you're working on requires the execution of an action only after a specific event has been emitted exactly three times. How do you implement this using the Events module?

Option 1:
Use the 'once' method to listen for the event and execute the action after the third emission.

Option 2:
Maintain a counter variable and execute the action when the counter reaches three after each event emission.

Option 3:
Implement a custom event emitter class with a 'count' property to track the number of emissions.

Option 4:
Utilize the 'twice' method of the Events module to specify the required number of emissions.

Correct Response:
2.0

Explanation:
To implement the desired behavior, option b is correct. By maintaining a counter variable and executing the action when the counter reaches three, you ensure that the action is performed only after the event has been emitted three times. The 'once' method (option a) would execute the action after the first emission, and the 'twice' method (option d) is not a standard method of the Events module. Option c is not necessary as the 'counter' approach is simpler.

You need to implement a logging system where every log message is both saved to a file and printed to the console. Which type of stream would you use and why?

Option 1:
Writable stream, as it allows data to be written to a destination such as a file.

Option 2:
Duplex stream, as it combines both readable and writable functionalities, enabling simultaneous file writing and console printing.

Option 3:
Readable stream, as it efficiently reads data from a source.

Option 4:
Transform stream, as it can modify the log messages before sending them to the file and console.

Correct Response:
2.0

Explanation:
Option b is correct because a Duplex stream allows both reading and writing. In this scenario, it facilitates writing log messages to a file and printing them to the console simultaneously. A Writable stream (option a) only allows writing, and a Readable stream (option c) is used for reading, not writing. Transform streams (option d) are designed for data transformation, which is not required in this case.

Which Node.js module provides utilities to work with the operating system?

Option 1:
fs

Option 2:
os

Option 3:
path

Option 4:
http

Correct Response:
2.0

Explanation:
The correct answer is option b) 'os'. The 'os' module in Node.js provides a set of operating system-related utility methods. These include functions to retrieve information about the operating system, such as free memory, platform, and architecture.

What method in the Node.js 'querystring' module is used to parse a query string into an object?

Option 1:
parse

Option 2:
stringify

Option 3:
decode

Option 4:
encode

Correct Response:
1.0

Explanation:
The correct answer is option a) 'parse'. The 'parse' method in the 'querystring' module is used to convert a URL query string into an object. It parses the query string and returns an object with key-value pairs representing the parameters.

In Node.js, which object provides information about the current Node.js process?

Option 1:
process

Option 2:
global

Option 3:
buffer

Option 4:
console

Correct Response:
1.0

Explanation:
The correct answer is option a) 'process'. The 'process' object in Node.js provides information about the current Node.js process. It includes details such as the environment, command-line arguments, and methods to control the process.

How can you access environment variables in a Node.js application?

Option 1:
process.env.VARIABLE_NAME

Option 2:
NODE_ENV.VARIABLE_NAME

Option 3:
env.VARIABLE_NAME

Option 4:
process.config.VARIABLE_NAME

Correct Response:
1.0

Explanation:
In Node.js, environment variables are accessed through the process.env object. You can use process.env.VARIABLE_NAME to access the value of a specific environment variable, where VARIABLE_NAME is the name of the variable you want to retrieve. This is a crucial aspect for configuration and handling sensitive information in Node.js applications.

What function would you use in Node.js to decode a URL component?

Option 1:
url.decodeComponent()

Option 2:
decodeURIComponent()

Option 3:
url.decodeURI()

Option 4:
uri.decodeComponent()

Correct Response:
2.0

Explanation:
To decode a URL component in Node.js, you would use the decodeURIComponent() function. This function is part of the global decodeURIComponent object and is essential when working with URLs, especially when handling user inputs or dealing with data from external sources. It helps in preventing errors and ensures proper handling of URL-encoded characters.

How do you terminate a Node.js program from within the script?

Option 1:
process.exit()

Option 2:
node.terminate()

Option 3:
exit.script()

Option 4:
process.terminate()

Correct Response:
1.0

Explanation:
To terminate a Node.js program from within the script, you can use the process.exit() method. This function allows you to exit the current Node.js process with an optional exit code. It is essential for managing the lifecycle of a Node.js application, especially in scenarios where you need to gracefully shut down the application or handle specific exit conditions.

Describe how to use the 'os' module in Node.js to monitor system memory usage.

Option 1:
os.memoryUsage()

Option 2:
os.monitorMemory()

Option 3:
os.checkMemory()

Option 4:
os.reportMemory()

Correct Response:
1.0

Explanation:
The correct option is a) os.memoryUsage(). This method returns an object describing the system memory usage, including the total available memory, used memory, and free memory. It is a useful function for monitoring and managing memory resources in Node.js applications.

What Node.js module method would you use to convert a URL object into a formatted URL string?

Option 1:
url.format()

Option 2:
url.stringify()

Option 3:
url.convert()

Option 4:
url.parseString()

Correct Response:
1.0

Explanation:
The correct option is a) url.format(). This method is used to convert a URL object into a formatted URL string. It takes a URL object as an argument and returns the formatted URL string. This is commonly used for creating URLs based on parsed components.

Explain the difference between process.stdout.write() and console.log() in Node.js.

Option 1:
process.stdout.write() writes to stdout without a newline

Option 2:
console.log() writes to stdout with a newline

Option 3:
process.stdout.write() is asynchronous

Option 4:
console.log() is synchronous

Correct Response:
1.0

Explanation:
The correct option is a) process.stdout.write() writes to stdout without a newline. process.stdout.write() is a lower-level method that writes data to the standard output (stdout) without adding a newline. On the other hand, console.log() writes to stdout with a newline character at the end, making it suitable for logging messages with a line break.

The global Node.js object _____ is used to handle events when the process is about to exit.

Option 1:
process.onExit

Option 2:
process.exitHandler

Option 3:
process.on('exit')

Option 4:
process.on('beforeExit')

Correct Response:
3.0

Explanation:
In Node.js, the process.on('exit') event is triggered just before the process is about to exit. Developers can use this event to perform cleanup operations or log messages before the program terminates.

To parse the URL query string into an object in Node.js, the _____ method of the 'url' module is used.

Option 1:
url.parseQuery

Option 2:
url.queryParser

Option 3:
url.parseQueryString

Option 4:
url.parse

Correct Response:
4.0

Explanation:
The url.parse method in the 'url' module is used to parse URL strings. It has an optional second parameter, parseQueryString, which, when set to true, parses the query string into an object.

In Node.js, process._____() is a method that is used to print to stdout with an optional newline character.

Option 1:
write

Option 2:
stdout.print

Option 3:
log

Option 4:
stdout.write

Correct Response:
4.0

Explanation:
The process.stdout.write() method in Node.js is used to write data to the standard output (stdout). It provides more control over the output and does not automatically append a newline character.

Node.js's process._____ method is used to immediately terminate the process with a given code.

Option 1:
exit()

Option 2:
kill()

Option 3:
terminate()

Option 4:
end()

Correct Response:
2.0

Explanation:
The correct method is process.kill(), which is used to immediately terminate the process with a given code. This method sends a signal to the process, allowing it to perform cleanup operations before exiting.

The url._____ method in Node.js is used to resolve a relative URL against an absolute base URL.

Option 1:
join()

Option 2:
resolve()

Option 3:
concatenate()

Option 4:
parse()

Correct Response:
2.0

Explanation:
The correct method is url.resolve(), which resolves a relative URL against an absolute base URL. It returns a formatted URL string.

When managing child processes in Node.js, _____.spawn() can be used to launch a new process with a given command.

Option 1:
process.create

Option 2:
child

Option 3:
child_process

Option 4:
spawn

Correct Response:
3.0

Explanation:
The correct function is child_process.spawn(), which is used to spawn a new process with the given command. It provides more control over the spawned process than child_process.exec().

You're writing a Node.js application that requires detailed information about the network interfaces available on the hosting machine. Which module and methods would you use to retrieve this information?

Option 1:
os.networkInterfaces()

Option 2:
net.getNetworkInterfaces()

Option 3:
network.getInterfaces()

Option 4:
os.getNetworkInterfaces()

Correct Response:
1.0

Explanation:
In Node.js, the os module provides the networkInterfaces() method, which returns an object containing information about the network interfaces on the host machine. It's the appropriate choice for retrieving detailed network interface information.

In your Node.js application, you need to ensure that query strings are correctly encoded and decoded, considering special characters and white spaces. What steps would you take to achieve this?

Option 1:
querystring.encode()

Option 2:
encodeURIComponent()

Option 3:
encodeURI()

Option 4:
queryString.stringify()

Correct Response:
2.0

Explanation:
To correctly encode query strings in Node.js, the encodeURIComponent() function should be used. This function ensures that special characters and white spaces are properly encoded for URL use, making it a suitable choice for this requirement.

You are tasked with writing a script that collects information on CPU and system uptime. Which Node.js module will provide you with the functions needed to accomplish this?

Option 1:
os.cpuInfo()

Option 2:
systemInfo.getCPUInfo()

Option 3:
os.cpus()

Option 4:
process.cpuUsage()

Correct Response:
3.0

Explanation:
The os module in Node.js provides the cpus() method, which returns an array of objects containing information about each CPU/core on the system. This method is suitable for collecting information on CPU and system uptime in a Node.js script.

In Node.js, which module provides cryptographic functionality including a set of wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign, and verify functions?

Option 1:
http

Option 2:
fs

Option 3:
crypto

Option 4:
path

Correct Response:
3.0

Explanation:
The correct answer is option C, crypto. The crypto module in Node.js provides cryptographic functionality, including wrappers for OpenSSL's hash, HMAC, cipher, decipher, sign, and verify functions. This module is commonly used for implementing secure communication and data integrity.

Which function from the Child Processes module in Node.js is used to spawn a new process using a given command?

Option 1:
fork

Option 2:
exec

Option 3:
spawn

Option 4:
child

Correct Response:
2.0

Explanation:
The correct answer is option B, exec. The exec function in the Child Processes module is used to spawn a new process using a given command. It is commonly used for running shell commands in a new child process from within a Node.js application.

When using the Crypto module to generate a hash, which method is used to input data to be hashed?

Option 1:
encrypt

Option 2:
update

Option 3:
digest

Option 4:
finalize

Correct Response:
2.0

Explanation:
The correct answer is option B, update. When using the Crypto module in Node.js to generate a hash, the update method is used to input data to be hashed. This method allows you to feed data to the hash function incrementally, making it useful for processing large datasets without loading them entirely into memory.

How can you ensure that data passed to a child process is safely transmitted without being altered or intercepted?

Option 1:
Encrypt the data using a secure algorithm

Option 2:
Use HTTPS for data transmission

Option 3:
Use JSON.stringify()

Option 4:
Use a secure communication channel like IPC

Correct Response:
1.0

Explanation:
In Node.js, to ensure safe transmission to a child process, encrypting the data with a secure algorithm is a common practice. It helps protect the data from being altered or intercepted. Options like using HTTPS or secure channels like Inter-Process Communication (IPC) are relevant, but encryption adds an extra layer of security. JSON.stringify() is used for serialization and does not inherently ensure data security.

What is the main difference between child_process.spawn() and child_process.exec() in Node.js?

Option 1:
spawn executes a command in a new process

Option 2:
exec spawns a new shell process

Option 3:
spawn is used for long-running processes

Option 4:
exec is used for short-lived processes

Correct Response:
1.0

Explanation:
The key distinction is that child_process.spawn() directly spawns the specified command as a new process, while child_process.exec() spawns a new shell process and runs the specified command within that shell. Additionally, spawn is suitable for long-running processes,

and exec is more appropriate for short-lived processes. Understanding this difference is crucial for selecting the right method based on the specific use case.

In the context of the Crypto module, how does a symmetric key differ from an asymmetric key?

Option 1:
Symmetric key uses a single shared key

Option 2:
Asymmetric key uses two different keys

Option 3:
Symmetric key is faster for encryption

Option 4:
Asymmetric key is more suitable for hashing

Correct Response:
2.0

Explanation:
Symmetric key cryptography employs a single shared key for both encryption and decryption, offering speed advantages. In contrast, asymmetric key cryptography utilizes two different keys (public and private), providing enhanced security but with a trade-off in speed. Understanding the characteristics of both key types is essential for choosing the appropriate cryptographic approach based on the specific security and performance requirements of a given scenario.

Describe how the use of child processes in Node.js can affect the Event Loop and process scheduling.

Option 1:
It improves the Event Loop efficiency by offloading tasks.

Option 2:
It can cause Event Loop delays and impact process scheduling.

Option 3:
It doesn't affect the Event Loop at all.

Option 4:
Child processes have no impact on Event Loop and scheduling.

Correct Response:
2.0

Explanation:
In Node.js, using child processes can introduce delays in the Event Loop due to the communication overhead. Understanding these implications is crucial to managing the performance of applications that rely on child processes for parallel execution.

What security considerations should be taken into account when using the Crypto module for password hashing?

Option 1:
Use of a strong hashing algorithm.

Option 2:
Salting passwords for additional security.

Option 3:
Regularly update Node.js and Crypto module.

Option 4:
Store passwords in plain text for simplicity.

Correct Response:
1.0

Explanation:
When using the Crypto module for password hashing, it's essential to employ strong and proven hashing algorithms. Additionally, salting passwords adds an extra layer of security. Keeping the software and

modules updated is a good practice to address potential vulnerabilities.

Explain the difference between using child_process.fork() and cluster.fork(). In which scenario would each be appropriate?

Option 1:
child_process.fork() is suitable for parallelizing CPU-bound tasks.

Option 2:
cluster.fork() is designed for multi-core systems and load balancing.

Option 3:
They are interchangeable and can be used in any scenario.

Option 4:
cluster.fork() is only for communication between nodes.

Correct Response:
2.0

Explanation:
child_process.fork() is used for creating separate instances of the V8

engine, while cluster.fork() is designed for distributing tasks across multiple processes on a single machine. Understanding their distinct purposes is crucial for efficient utilization in different scenarios, such as parallelizing CPU-bound tasks or achieving load balancing.

The crypto.createCipheriv() method in Node.js requires an algorithm, a key, and an _____ as parameters.

Option 1:
IV (Initialization Vector)

Option 2:
Buffer

Option 3:
Object

Option 4:
Integer

Correct Response:
1.0

Explanation:
The correct option is IV (Initialization Vector). The createCipheriv() method is used for creating a Cipher object with the specified algorithm, key, and initialization vector (IV). The IV is crucial for ensuring the security of the encryption process.

The child_process.execFile() function is a variant of child_process.exec() that executes a file directly without first spawning a _____.

Option 1:
Shell

Option 2:
Child Process

Option 3:
Parent Process

Option 4:
Subprocess

Correct Response:
3.0

Explanation:
The correct option is Parent Process. Unlike exec(), execFile()

directly executes a file without spawning a shell. It doesn't create a new shell process; instead, it runs the specified file in the context of the current process, making the parent process the direct executor.

To generate cryptographically strong random values, use the crypto.randomBytes(size, [callback]) function, where size is the number of _____ to generate.

Option 1:
Bytes

Option 2:
Bits

Option 3:
Characters

Option 4:
Strings

Correct Response:
1.0

Explanation:
The correct option is Bytes. The randomBytes() function is used to generate cryptographically strong random values. The size parameter specifies the number of random bytes to generate, ensuring a secure source of randomness for cryptographic purposes.

Node.js Crypto module's crypto.createSign() method is used to generate a signature for a given set of data using a private key and a specified _____ algorithm.

Option 1:
Hash

Option 2:
Symmetric

Option 3:
Asymmetric

Option 4:
Digest

Correct Response:
3.0

Explanation:
The correct option is (c) Asymmetric. In this context, the crypto.createSign() method in Node.js is used for creating a signature with an asymmetric algorithm, where different keys are used for encryption and decryption. Asymmetric algorithms involve a pair of public and private keys. In this case, the private key is used to sign the data, and the corresponding public key can be used to verify the signature.

In a child process, the global process.send() method is used for sending messages to the parent process when the child is spawned using child_process._____().

Option 1:
fork

Option 2:
spawn

Option 3:
exec

Option 4:
execFile

Correct Response:
1.0

Explanation:
The correct option is (a) fork. When a child process is spawned in

Node.js using the child_process.fork() method, it creates a new Node.js process and uses inter-process communication (IPC) to allow sending and receiving messages between the parent and child processes. The process.send() method is then used to send messages from the child process to the parent process.

When working with child processes, the _____ event is emitted after the child process ends and the operating system has terminated it.

Option 1:
exit

Option 2:
close

Option 3:
disconnect

Option 4:
message

Correct Response:
1.0

Explanation:
The correct option is (a) exit. In Node.js, the 'exit' event is emitted

when a child process ends and the operating system has terminated it. You can listen for this event to perform cleanup tasks or take necessary actions after the child process has completed its execution.

You need to encrypt and decrypt sensitive data in a Node.js application, ensuring that the encryption key is never exposed or hardcoded in the source code. What approach would you use with the Crypto module to accomplish this?

Option 1:
Use the crypto.createCipher method with a passphrase for key derivation.

Option 2:
Utilize the crypto.generateKeyPair method to dynamically generate encryption keys.

Option 3:
Implement the crypto.publicEncrypt and crypto.privateDecrypt methods for secure key handling.

Option 4:
Apply the crypto.scrypt function to derive a key from a password for encryption and decryption.

Correct Response:
1.0

Explanation:
The crypto.createCipher method with a passphrase allows for secure encryption without exposing the key. The passphrase serves as an input for key derivation, ensuring the key is not hardcoded in the source code. This approach enhances security in handling sensitive data.

You're implementing a service that needs to execute several CPU-intensive tasks in parallel without blocking the main application thread. How would you leverage the Child Processes module to achieve optimal performance?

Option 1:
Use the fork method to create child processes, distributing CPU-intensive tasks among them.

Option 2:
Employ the spawn method to execute parallel tasks, ensuring efficient resource utilization.

Option 3:
Utilize the exec method to run multiple processes concurrently for optimal performance.

Option 4:
Implement the cluster module to distribute tasks across multiple instances of the Node.js application.

Correct Response:
1.0

Explanation:
The fork method creates child processes that run independently, allowing for parallel execution of CPU-intensive tasks without blocking the main application thread. This approach maximizes performance by utilizing multiple cores efficiently and is a recommended practice for handling computationally intensive operations.

A Node.js application you are maintaining has to comply with security standards that require the use of hardware security modules (HSMs) for cryptographic operations. What changes or integrations would you consider in the Crypto module to utilize HSMs?

Option 1:
Integrate the crypto.hsm module to enable HSM support for cryptographic operations.

Option 2:
Update the application to use the crypto.hardware flag for enabling

HSM integration.

Option 3:
Utilize the crypto.setHSM method to enable HSM support in the existing Crypto module.

Option 4:
Investigate and integrate a third-party library compatible with HSMs for cryptographic functions.

Correct Response:
4.0

Explanation:
Integrating a third-party library compatible with HSMs is a common approach. Since the Crypto module may not directly support HSMs, using a well-established library designed for HSM integration ensures compliance with security standards. This approach involves selecting a library, understanding its API, and incorporating it into the Node.js application to perform cryptographic operations securely using hardware security modules.

What is the primary use of the Node.js cluster module?

Option 1:
Load balancing

Option 2:
File system manipulation

Option 3:
Database querying

Option 4:
HTTP request handling

Correct Response:
1.0

Explanation:
The Node.js cluster module is primarily used for load balancing. It allows a Node.js application to utilize multiple CPU cores, improving performance by distributing incoming requests across the cores.

In Node.js, what command is used to spawn a child process?

Option 1:
spawnChild()

Option 2:
fork()

Option 3:
createProcess()

Option 4:
executeChild()

Correct Response:
2.0

Explanation:
In Node.js, the fork() method is used to spawn a child process. This method is particularly useful for creating child processes that communicate with the parent process using inter-process communication (IPC).

Which Node.js global function triggers garbage collection manually, assuming the --expose-gc flag is used?

Option 1:
gc()

Option 2:
collectGarbage()

Option 3:
triggerGC()

Option 4:
global.gc()

Correct Response:
4.0

Explanation:
The correct option is D. In Node.js, using global.gc() triggers garbage collection manually when the --expose-gc flag is used. This can be helpful in managing memory resources more efficiently.

How can you share server ports among different worker processes when using Node.js clustering?

Option 1:
Using a load balancer

Option 2:
Using the cluster module

Option 3:
Using IPC (Inter-Process Communication)

Option 4:
Using a proxy server

Correct Response:
2.0

Explanation:
In Node.js clustering, the cluster module allows multiple Node.js processes to share the same port. This is achieved by creating worker processes that can handle incoming requests. Option B is correct because it refers to using the cluster module for this purpose. Options A, C, and D are not direct methods for sharing ports in Node.js clustering.

What is the difference between child_process.spawn() and child_process.fork() in Node.js?

Option 1:
spawn() is used for synchronous processes, while fork() is used for asynchronous processes

Option 2:
spawn() creates a new Node.js process, while fork() creates a new system process

Option 3:
fork() is a special case of spawn() designed for Node.js child processes

Option 4:
spawn() is used for creating additional worker threads, while fork() is used for creating child processes

Correct Response:
3.0

Explanation:
child_process.spawn() is used to launch a new process with the given command. It does not create a new V8 instance, so it is more lightweight. On the other hand, child_process.fork() is a special case of spawn() specifically designed for creating child processes that communicate with each other using Inter-Process Communication (IPC).

What is a potential effect of garbage collection on a Node.js application's performance?

Option 1:
Increased memory consumption

Option 2:
Decreased CPU usage

Option 3:
Temporary freezing of the event loop

Option 4:
Improved application responsiveness

Correct Response:
3.0

Explanation:
Garbage collection in Node.js can lead to a temporary freezing of the event loop, causing delays in processing incoming requests. During garbage collection, the Node.js runtime pauses the execution of the program to reclaim memory occupied by objects that are no longer in use. Options A, B, and D are not direct effects of garbage collection on performance.

How does the master process communicate with worker processes in a clustered Node.js application?

Option 1:
Inter-process communication

Option 2:
TCP/IP communication

Option 3:
EventEmitter

Option 4:
Message passing

Correct Response:
4.0

Explanation:
In a clustered Node.js application, the master process communicates with worker processes through message passing. This involves using the cluster module to send messages between the master and worker processes, enabling coordination and sharing of information. This mechanism is crucial for load balancing and efficient utilization of resources in a clustered environment.

What is the role of the V8 garbage collector in Node.js, and how does it handle memory allocation?

Option 1:
Manual memory management

Option 2:
Automatic memory management

Option 3:
Incremental garbage collection

Option 4:
Reference counting

Correct Response:
2.0

Explanation:
The V8 garbage collector in Node.js is responsible for automatic memory management. It employs a generational garbage collection strategy, dividing objects into young and old generations. The collector uses techniques like scavenging and mark-sweep to efficiently reclaim memory. Understanding V8's garbage collection mechanisms is essential for optimizing memory usage in Node.js applications.

Can you explain the concept of "worker thread isolation" in the context of Node.js clustering and how it impacts the sharing of in-memory data?

Option 1:
Shared memory space

Option 2:
Worker thread encapsulation

Option 3:
Thread-specific memory

Option 4:
Process-level isolation

Correct Response:
4.0

Explanation:
In the context of Node.js clustering, "worker thread isolation" refers

to the process-level isolation of worker threads. Each worker process operates independently, with its own memory space. While this isolation enhances robustness, it also means that in-memory data is not shared directly between worker threads. Developers must use inter-process communication mechanisms to share data between workers. Understanding this concept is crucial for designing scalable and resilient Node.js applications.

To handle load balancing in a clustered Node.js application, the cluster module uses the _____ technique.

Option 1:
Round-robin

Option 2:
Event-driven

Option 3:
Load-shedding

Option 4:
Shared-memory

Correct Response:
1.0

Explanation:
The correct option is (a) Round-robin. The cluster module in Node.js uses a round-robin technique for distributing incoming connections among the various worker processes in a cluster, ensuring balanced load distribution.

When a Node.js process runs out of memory due to a memory leak, it is often due to not properly handling _____.

Option 1:
File I/O

Option 2:
Asynchronous callbacks

Option 3:
Error events

Option 4:
Garbage collection

Correct Response:
2.0

Explanation:
The correct option is (b) Asynchronous callbacks. Memory leaks in Node.js processes often occur when asynchronous callbacks are not handled correctly, leading to the accumulation of unreleased

memory. It's crucial to manage callbacks properly to prevent memory issues.

The child_process module method _____ is best suited for CPU-intensive tasks since it creates a new V8 instance.

Option 1:
execSync

Option 2:
fork

Option 3:
spawn

Option 4:
execFile

Correct Response:
3.0

Explanation:
The correct option is (c) spawn. The spawn method in the child_process module is well-suited for CPU-intensive tasks as it creates a new V8 instance for the child process, allowing it to run independently and efficiently handle CPU-bound operations.

Node.js uses _____ garbage collection strategies to optimize memory use.

Option 1:
Automatic

Option 2:
Manual

Option 3:
Dynamic

Option 4:
Mark-and-Sweep

Correct Response:
4.0

Explanation:
Node.js employs the Mark-and-Sweep garbage collection strategy. In this method, the runtime identifies and collects objects that are no longer reachable by the application, optimizing memory usage.

In a Node.js application, the _____ event is emitted when a worker process dies or is killed.

Option 1:
'exit'

Option 2:
'error'

Option 3:
'disconnect'

Option 4:
'process'

Correct Response:
1.0

Explanation:
The 'exit' event is emitted in a Node.js application when a worker process dies or is explicitly killed. This event allows handling cleanup operations or logging when a process exits.

The command-line flag _____ can be used to trace garbage collection events in Node.js for debugging purposes.

Option 1:
#NAME?

Option 2:
#NAME?

Option 3:
#NAME?

Option 4:
#NAME?

Correct Response:
1.0

Explanation:
The '--trace-gc' command-line flag in Node.js enables tracing of garbage collection events. This is useful for debugging memory-related issues by providing insights into when and how garbage collection occurs.

You're implementing a Node.js application that requires high availability and multi-core scalability. How would you configure your application to restart a worker process if it fails?

Option 1:
PM2 Cluster Mode

Option 2:
Node.js Cluster Module

Option 3:
Forever Module

Option 4:
Node.js Restart Policies

Correct Response:
1.0

Explanation:
In a scenario requiring high availability and multi-core scalability, PM2 Cluster Mode allows automatic restart of failed worker processes. The other options may provide some level of clustering but lack the robust features of PM2.

During peak traffic, your Node.js application experiences a slowdown. How would you determine if this is due to garbage collection pauses?

Option 1:
Use the --trace-gc flag

Option 2:
Analyze the CPU usage during peak traffic

Option 3:
Increase the heap size

Option 4:
Use the Event Loop Lag metric

Correct Response:
1.0

Explanation:
The --trace-gc flag enables detailed garbage collection tracing,

providing insights into when and how long garbage collection pauses occur. Other options may not specifically address garbage collection pauses.

If your Node.js application's child process needs to perform a CPU-bound task, how can you ensure this does not block the main event loop?

Option 1:
Use the child_process.fork() method with IPC

Option 2:
Use the child_process.spawn() method with stdio options

Option 3:
Implement a callback function for the child process

Option 4:
Utilize worker threads in the worker_threads module

Correct Response:
4.0

Explanation:
Worker threads in the worker_threads module enable the execution

of CPU-bound tasks without blocking the main event loop. The other options may introduce some level of asynchrony but not as efficiently.

Which Node.js core module provides utilities to measure the performance of your code?

Option 1:
fs

Option 2:
http

Option 3:
util

Option 4:
net

Correct Response:
3.0

Explanation:
In Node.js, the util module provides utility functions, including the util.promisify method used to convert callback-based functions into Promises. This module also includes functions for measuring the execution time of code using the util.promisify.custom symbol.

What is a common first step in securing a Node.js application?

Option 1:
Use a strong encryption algorithm

Option 2:
Keep dependencies updated

Option 3:
Disable all logging

Option 4:
Remove error messages from the server responses

Correct Response:
2.0

Explanation:
Keeping dependencies updated is a crucial step in securing a Node.js application. Vulnerabilities in dependencies are regularly discovered, and updating ensures that the latest security patches are applied.

In the context of Node.js, what does APM stand for?

Option 1:
Advanced Package Manager

Option 2:
Asynchronous Process Manager

Option 3:
Application Performance Monitoring

Option 4:
Asynchronous Programming Model

Correct Response:
3.0

Explanation:
APM stands for Application Performance Monitoring in the context of Node.js. APM tools help developers monitor the performance of their applications, identify bottlenecks, and optimize code for better efficiency.

What tool would you use to identify memory leaks in a Node.js application?

Option 1:
Chrome DevTools

Option 2:
Postman

Option 3:
Wireshark

Option 4:
Git

Correct Response:
1.0

Explanation:
Memory Leaks can be identified using tools like Chrome DevTools, which includes a "Memory" tab allowing inspection of memory usage over time. It helps to pinpoint memory leaks and optimize the application.

Which HTTP header is important to set correctly to prevent attacks like cross-site scripting (XSS) in a Node.js web application?

Option 1:
Cache-Control

Option 2:
Content-Type

Option 3:
X-Frame-Options

Option 4:
Access-Control-Allow-Origin

Correct Response:
3.0

Explanation:
The correct header is X-Frame-Options. It prevents the web page from being embedded in iframes, reducing the risk of XSS attacks that involve loading the page within a frame.

When monitoring Node.js applications, what is an important metric to watch that could indicate blocking of the event loop?

Option 1:
CPU Usage

Option 2:
Memory Usage

Option 3:
Event Loop Lag

Option 4:
Network Throughput

Correct Response:
3.0

Explanation:
Monitoring the Event Loop Lag is crucial. An increasing lag indicates that the event loop is taking longer to process tasks, potentially leading to performance issues and blocking.

How does the Flamegraph help in Node.js application performance monitoring?

Option 1:
Visualizing function call times

Option 2:
Analyzing CPU usage

Option 3:
Inspecting memory leaks

Option 4:
Profiling network latency

Correct Response:
2.0

Explanation:
The Flamegraph is a visualization tool that helps in analyzing CPU usage. It displays a graphical representation of stack traces, making it easier to identify performance bottlenecks related to function call times and CPU usage.

Explain the role of Content Security Policy (CSP) in Node.js application security.

Option 1:
Restricting database access

Option 2:
Mitigating Cross-Site Scripting (XSS) attacks

Option 3:
Ensuring file integrity

Option 4:
Managing user authentication

Correct Response:
2.0

Explanation:
Content Security Policy (CSP) is a security standard that helps prevent XSS attacks. It defines policies for controlling the sources of content that a web application can load, reducing the risk of malicious script injection.

What is a security risk unique to Node.js due to its event-driven architecture?

Option 1:
SQL injection attacks

Option 2:
Callback hell

Option 3:
Cross-Site Request Forgery (CSRF)

Option 4:
Buffer overflow vulnerabilities

Correct Response:
2.0

Explanation:
Callback hell is a challenge in Node.js where nested callbacks lead to unreadable and error-prone code. While not a direct security risk, it can indirectly contribute to vulnerabilities if not managed properly.

In Node.js, you can use the _____ module to create a simple profiling mechanism which allows you to measure the time taken by functions.

Option 1:
timers

Option 2:
events

Option 3:
profiler

Option 4:
performance

Correct Response:
1.0

Explanation:
The correct module is timers which provides functions for scheduling

code execution after a set time interval. This is useful for profiling to measure the time taken by functions.

A _____ report can be generated to get insights into the operational aspects of a Node.js application, useful in diagnosing problems.

Option 1:
Debugging

Option 2:
Health Check

Option 3:
Performance

Option 4:
Status

Correct Response:
3.0

Explanation:
The correct report is a Performance report, which can be generated to analyze the performance of a Node.js application. It provides insights into the operational aspects, helping in diagnosing performance-related problems.

To guard against a wide range of security threats, it's recommended to use a middleware like _____ in your Node.js application.

Option 1:
helmet

Option 2:
shield

Option 3:
armor

Option 4:
secure-middleware

Correct Response:
1.0

Explanation:
The recommended middleware for enhancing security in a Node.js application is helmet. It helps set various HTTP headers to secure the application against common vulnerabilities.

The _____ tool in Node.js can be used to take a heap snapshot and identify memory leaks.

Option 1:
npm

Option 2:
v8

Option 3:
debugger

Option 4:
inspector

Correct Response:
2.0

Explanation:
The correct option is b) v8. The V8 profiler tool in Node.js can be utilized to capture heap snapshots, enabling developers to analyze memory usage and identify potential memory leaks in their applications.

For security, Node.js applications should be run with least privilege principles, typically using a _____ user.

Option 1:
root

Option 2:
admin

Option 3:
standard

Option 4:
lowprivilege

Correct Response:
4.0

Explanation:
The correct option is d) lowprivilege. Running Node.js applications with the least privilege principle helps minimize the potential impact of security vulnerabilities. Using a low-privilege user enhances the security posture of the application.

To avoid unintentional exposure of sensitive information, environment variables should be accessed through _____ in Node.js.

Option 1:
process.argv

Option 2:
process.env

Option 3:
config

Option 4:
global

Correct Response:
2.0

Explanation:
The correct option is b) process.env. Accessing environment

variables through process.env in Node.js ensures a more secure approach, as it provides a reliable and controlled way to retrieve sensitive information without exposing it unintentionally.

You have deployed a Node.js application into production, and users are reporting slow response times. Which performance monitoring strategy could help you pinpoint the bottleneck?

Option 1:
Heap Dump Analysis

Option 2:
Request Tracing

Option 3:
Event Loop Profiling

Option 4:
Garbage Collection Metrics

Correct Response:
2.0

Explanation:
In the context of slow response times, Request Tracing can help identify the specific requests or operations causing the bottleneck. By analyzing the time taken at each step, you can pinpoint the exact source of the performance issue. Heap Dump Analysis, Event Loop Profiling, and Garbage Collection Metrics are useful but focus on different aspects of performance.

During a security audit of a Node.js API, you discover that sensitive data is being exposed due to an insecure direct object reference (IDOR) vulnerability. What immediate step should be taken to mitigate this?

Option 1:
Implement Access Control Lists (ACLs)

Option 2:
Use JSON Web Tokens (JWT) for Authentication

Option 3:
Encrypt Sensitive Data

Option 4:
Validate and Sanitize User Input

Correct Response:
1.0

Explanation:
To mitigate an insecure direct object reference (IDOR) vulnerability, implementing Access Control Lists (ACLs) is crucial. ACLs restrict access to certain resources based on user roles, preventing unauthorized access. The other options address different aspects of security but may not directly mitigate IDOR.

A Node.js application is experiencing high latency, and CPU profiling indicates that the main thread is frequently blocked. What code pattern might be responsible for this behavior, and how can it be resolved?

Option 1:
Synchronous Blocking Code

Option 2:
Asynchronous Callback Hell

Option 3:
Promise Chaining

Option 4:
Event Loop Starvation

Correct Response:
1.0

Explanation:
High latency and frequent main thread blocking often result from synchronous blocking code. To resolve this, consider refactoring code to be more asynchronous, using features like Promises or async/await. The other options, while relevant to Node.js performance, do not directly address the issue of synchronous blocking.

Which Node.js feature is typically used to render HTML on the server before sending it to the client?

Option 1:
npm

Option 2:
Express

Option 3:
EJS

Option 4:
AJAX

Correct Response:
3.0

Explanation:
In Node.js, the 'EJS' (Embedded JavaScript) templating engine is commonly used to render HTML on the server, allowing dynamic content generation before sending it to the client.

What does API stand for, and what is its primary use in web development with Node.js?

Option 1:
Advanced Programming Interface

Option 2:
Application Programming Interface

Option 3:
Advanced Protocol Interface

Option 4:
Asynchronous Programming Interface

Correct Response:
2.0

Explanation:
API stands for 'Application Programming Interface.' In Node.js, an API is a set of rules that allows one software application to interact with another. It facilitates communication between different parts of a web application.

When building a microservices architecture in Node.js, what is the main advantage over a monolithic architecture?

Option 1:
Scalability

Option 2:
Simplicity

Option 3:
Centralized Management

Option 4:
Ease of Debugging

Correct Response:
1.0

Explanation:
The main advantage of using a microservices architecture in Node.js is scalability. It allows for independent scaling of individual services, enhancing performance and resource utilization compared to a monolithic architecture.

What are some common tools or frameworks used in Node.js for server-side rendering?

Option 1:
Express.js

Option 2:
Next.js

Option 3:
Vue.js

Option 4:
Django

Correct Response:
2.0

Explanation:
Server-side rendering in Node.js is often achieved using frameworks like Next.js, which provides a robust solution for rendering React components on the server. Express.js is a popular Node.js framework, but it is not specifically designed for server-side rendering. Vue.js is a JavaScript framework but is more commonly associated with client-side rendering. Django is a Python web framework and not relevant in the context of Node.js.

How does an API backend in Node.js typically handle database operations differently from server-side rendering?

Option 1:
Using asynchronous operations

Option 2:
Synchronous operations

Option 3:
Both synchronous and asynchronous operations

Option 4:
None of the above

Correct Response:
1.0

Explanation:
In the context of API backends in Node.js, asynchronous operations are commonly used to handle database operations. This allows the server to continue processing other tasks while waiting for the database operations to complete, leading to better performance.

Synchronous operations would block the server, impacting responsiveness.

In the context of microservices with Node.js, what is the role of a service registry?

Option 1:
Managing and registering microservices

Option 2:
Storing user credentials

Option 3:
Load balancing in a microservices architecture

Option 4:
Data caching for microservices

Correct Response:
1.0

Explanation:
A service registry in the context of microservices with Node.js is responsible for managing and registering microservices. It keeps track of available services, their locations, and health, enabling dynamic discovery and communication between microservices. Options 2, 3, and 4 are not the primary roles of a service registry in microservices architecture.

Can you explain how server-side rendering might impact SEO compared to a client-side only API backend?

Option 1:
It improves SEO by pre-rendering content.

Option 2:
It has no impact on SEO.

Option 3:
It worsens SEO due to delayed rendering.

Option 4:
It enhances SEO through client-side scripts.

Correct Response:
1.0

Explanation:
Server-side rendering improves SEO by pre-rendering content on the server, making it more search engine-friendly. Client-side only API backends may face delayed rendering issues, affecting SEO negatively. Understanding the impact of rendering strategies is crucial for optimizing web applications for search engines.

Discuss the challenges of managing state across multiple microservices in a Node.js application.

Option 1:
Centralized state management

Option 2:
Database synchronization

Option 3:
Event sourcing

Option 4:
Stateless microservices

Correct Response:
3.0

Explanation:
Managing state across multiple microservices poses challenges. Event sourcing is a strategy where state changes are captured as events, enabling synchronization. Understanding challenges and adopting suitable patterns, like event sourcing, is essential for effective state management in a microservices architecture.

What patterns or strategies are commonly employed to facilitate communication between microservices in a Node.js ecosystem?

Option 1:
RESTful APIs

Option 2:
Message queues

Option 3:
GraphQL

Option 4:
Shared databases

Correct Response:
2.0

Explanation:
Facilitating communication between microservices involves choosing effective patterns. Message queues provide asynchronous communication, enhancing scalability. Understanding various communication strategies, such as RESTful APIs and message

queues, is vital for designing resilient and scalable Node.js microservices ecosystems.

Server-side rendering in Node.js can be efficiently handled using the _____ template engine.

Option 1:
EJS

Option 2:
Mustache

Option 3:
Jade

Option 4:
Handlebars

Correct Response:
1.0

Explanation:
Server-side rendering is a technique where the server generates HTML pages and sends them to the client. In Node.js, the EJS (Embedded JavaScript) template engine is commonly used for server-side rendering. EJS allows embedding JavaScript code directly within HTML, making it easy to generate dynamic content on the server side.

An API backend in Node.js often communicates with the front end using the _____ format.

Option 1:
JSON

Option 2:
XML

Option 3:
CSV

Option 4:
YAML

Correct Response:
1.0

Explanation:
JSON (JavaScript Object Notation) is a lightweight data interchange format commonly used in API communication. It is easy for humans to read and write and easy for machines to parse and generate. In Node.js, the JSON format is frequently used for exchanging data between the server and the client in API communications.

Microservices in Node.js can enhance scalability by being deployed as _____, which can be managed by orchestration tools like Kubernetes.

Option 1:
Containers

Option 2:
Virtual Machines

Option 3:
Monoliths

Option 4:
Modules

Correct Response:
1.0

Explanation:
Microservices architecture involves breaking down an application

into small, independent services that can be deployed and scaled individually. Containers, such as those provided by Docker, are a common choice for deploying microservices. Orchestration tools like Kubernetes help manage and scale these containers efficiently.

For SEO purposes, server-side rendering might be preferred over an API backend because search engines can index content that is _____.

Option 1:
dynamically generated

Option 2:
static

Option 3:
client-side rendered

Option 4:
encrypted

Correct Response:
1.0

Explanation:
Search engines rely on static content for better indexing. Server-side

rendering provides pre-rendered content, making it more SEO-friendly.

A _____ is a design approach in microservices that allows individual services to be developed, deployed, and scaled independently in a Node.js application.

Option 1:
Monolithic architecture

Option 2:
Singleton pattern

Option 3:
Microservices architecture

Option 4:
Facade pattern

Correct Response:
3.0

Explanation:
Microservices architecture allows independent development and scaling of services, contributing to better flexibility and scalability.

_____ is a term used to describe the technique of breaking down a monolithic Node.js application into smaller, independently deployable services.

Option 1:
Modularization

Option 2:
Monetization

Option 3:
Decentralization

Option 4:
Microservices migration

Correct Response:
4.0

Explanation:
Breaking down a monolithic application into smaller, independently

deployable services is known as microservices migration.

What is a common tool used in Node.js to check for memory leaks within an application?

Option 1:
Node Inspector

Option 2:
Nodemon

Option 3:
PM2

Option 4:
Valgrind

Correct Response:
1.0

Explanation:
In Node.js, Node Inspector is a popular tool for debugging and profiling applications, including identifying and fixing memory leaks. It allows developers to inspect memory usage and performance.

Which Node.js global function can you use to inspect the memory usage of your process?

Option 1:
process.memoryUsage()

Option 2:
global.inspectMemory()

Option 3:
node.inspect()

Option 4:
memory.inspectProcess()

Correct Response:
1.0

Explanation:
The process.memoryUsage() function in Node.js provides information about the memory usage of the current process, including details about heap and RSS (resident set size).

When integrating native addons in Node.js, which of the following file extensions typically represents a compiled addon?

Option 1:
.json

Option 2:
.js

Option 3:
.node

Option 4:
.addon

Correct Response:
3.0

Explanation:
In Node.js, native addons are often compiled binaries with the extension .node. This allows them to be loaded by the Node.js runtime for enhanced performance and interaction with native code.

Which npm package provides utilities to monitor and manage memory leaks in a Node.js application?

Option 1:
heapdump

Option 2:
memory-leak-detector

Option 3:
leak-monitor

Option 4:
node-memwatch

Correct Response:
4.0

Explanation:
The correct option is node-memwatch. This npm package is widely used for monitoring and managing memory leaks in Node.js applications. It provides tools to detect and analyze memory usage, making it easier to identify and address memory-related issues. Utilizing such packages is crucial for maintaining the performance and stability of Node.js applications.

How can memory leaks in Node.js affect the event loop and overall application performance?

Option 1:
Memory leaks have no impact on the event loop.

Option 2:
Memory leaks can lead to increased CPU usage.

Option 3:
Memory leaks can cause the event loop to become inefficient.

Option 4:
Memory leaks only affect asynchronous operations.

Correct Response:
3.0

Explanation:
Memory leaks in Node.js can significantly impact the event loop by causing it to become inefficient. As memory leaks accumulate, the application may experience degraded performance, increased response times, and even crashes. Developers should actively monitor and address memory leaks to ensure the smooth operation of the event loop and overall application performance.

What is the standard tool used to compile and install native addons in Node.js from npm?

Option 1:
node-build

Option 2:
npm-native

Option 3:
node-addon-gyp

Option 4:
addon-compiler

Correct Response:
3.0

Explanation:
The correct option is node-addon-gyp. This tool is the standard for compiling and installing native addons in Node.js from npm. It simplifies the process of building addons that include C++ code and ensures compatibility with different platforms. Understanding and using node-addon-gyp is essential for developers working on projects that require native addons in Node.js.

In the context of memory leaks, what is the significance of the V8 garbage collector in Node.js?

Option 1:
Efficient handling of circular references

Option 2:
Automatic detection and reclamation of unused memory

Option 3:
Priority-based allocation of memory

Option 4:
Manual garbage collection through JavaScript

Correct Response:
2.0

Explanation:
The V8 garbage collector in Node.js automatically detects and reclaims memory that is no longer in use, preventing memory leaks. It uses a tracing garbage collection algorithm to efficiently manage memory, especially beneficial in handling circular references.

How do native addons in Node.js handle memory allocation differently from JavaScript code, and what implications does this have for memory leaks?

Option 1:
Native addons use automatic memory management

Option 2:
Native addons rely on JavaScript's garbage collector

Option 3:
Native addons manage memory manually

Option 4:
Native addons have no impact on memory allocation

Correct Response:
3.0

Explanation:
Native addons in Node.js manage memory manually, which can lead

to memory leaks if not handled properly. Unlike JavaScript's automatic memory management, developers need to be cautious when working with native addons to avoid memory-related issues.

Describe the process and significance of using memory profiling tools in identifying leaks in a Node.js application.

Option 1:
Memory profiling helps identify CPU bottlenecks

Option 2:
Memory profiling tools analyze memory usage patterns

Option 3:
Memory profiling is only useful for frontend code

Option 4:
Memory profiling is irrelevant in Node.js

Correct Response:
2.0

Explanation:
Memory profiling tools in Node.js analyze memory usage patterns, helping developers identify and address potential memory leaks. By examining memory allocations and deallocations, developers can optimize their code and improve overall application performance.

To detect a memory leak, developers often take heap snapshots and compare them using the _____ tool in Node.js.

Option 1:
Node Inspector

Option 2:
Memory Analyzer

Option 3:
Chrome Developer Tools

Option 4:
Node.js Profiler

Correct Response:
2.0

Explanation:
The correct option is B. Memory Analyzer. Memory leaks can be identified by taking heap snapshots and analyzing them using tools like the Memory Analyzer, which helps developers pinpoint memory usage issues.

A native addon in Node.js is usually written in _____ and compiled into a binary format.

Option 1:
JavaScript

Option 2:
TypeScript

Option 3:
C++

Option 4:
Java

Correct Response:
3.0

Explanation:
The correct option is C. C++. Native addons in Node.js are typically written in C++ and then compiled into a binary format that can be loaded into Node.js using the require function.

When tracking down memory leaks, the _____ module can be used to log memory usage over time.

Option 1:
fs

Option 2:
util

Option 3:
v8

Option 4:
events

Correct Response:
3.0

Explanation:
The correct option is C. v8. The v8 module in Node.js provides methods to access the V8 JavaScript engine, including the ability to log memory usage over time, which is useful for tracking down memory leaks.

Native addons in Node.js can be linked to external libraries using _____, which defines build rules for native code.

Option 1:
CMake

Option 2:
GYP (Generate Your Projects)

Option 3:
Babel

Option 4:
Webpack

Correct Response:
2.0

Explanation:
In Node.js, GYP is a build configuration tool that can be used to link native addons to external libraries. GYP stands for "Generate Your Projects" and is commonly used for this purpose.

The _____ garbage collector strategy in V8 is instrumental in managing memory for short-lived objects in Node.js.

Option 1:
Mark-and-Sweep

Option 2:
Generational

Option 3:
Reference Counting

Option 4:
Stop-the-world

Correct Response:
2.0

Explanation:
V8, the JavaScript engine used by Node.js, employs a generational garbage collection strategy. This strategy is effective in handling short-lived objects, improving memory management in Node.js applications.

Advanced detection of memory leaks in Node.js applications can be achieved through _____ tracing, which records allocation and deallocation events.

Option 1:
CPU

Option 2:
Heap

Option 3:
Event

Option 4:
Garbage

Correct Response:
2.0

Explanation:
Heap tracing is a method for advanced detection of memory leaks in Node.js. It involves recording allocation and deallocation events in the heap, providing insights into memory usage and potential leaks in the application.

You have implemented a native addon for image processing in Node.js. After deployment, users report increasing latency over time. How would you approach diagnosing and fixing this issue?

Option 1:
Use the Node.js built-in 'profiler' module

Option 2:
Employ performance monitoring tools like 'New Relic'

Option 3:
Analyze system resource usage with 'pm2'

Option 4:
Inspect code with 'console.time' and 'console.timeEnd'

Correct Response:
2.0

Explanation:
The correct option is to employ performance monitoring tools like 'New Relic.' Profiling tools such as 'New Relic' can provide insights into the application's performance and help identify bottlenecks causing latency.

During the development of a Node.js application, you notice that the heap size grows significantly after several hours of stress testing. Which profiling tools or techniques would you use to identify potential memory leaks?

Option 1:
Utilize the 'memwatch-next' module

Option 2:
Leverage the 'heapdump' library

Option 3:
Analyze memory usage with 'ndb' debugger

Option 4:
Monitor garbage collection patterns with 'node-gc-profiler'

Correct Response:
1.0

Explanation:
The correct option is to utilize the 'memwatch-next' module. It provides tools for detecting and analyzing memory leaks in Node.js applications by monitoring object allocations and garbage collection events.

You are optimizing a computational-heavy Node.js application that uses native addons for performance-critical operations. How would you ensure that the native addons do not contribute to memory leaks?

Option 1:
Implement proper error handling in native addon code

Option 2:
Regularly update native addon dependencies

Option 3:
Use tools like 'Valgrind' for memory analysis

Option 4:
Leverage 'NAN' (Native Abstractions for Node.js)

Correct Response:
4.0

Explanation:
The correct option is to leverage 'NAN' (Native Abstractions for Node.js). 'NAN' provides a stable API for native addons, ensuring compatibility across different Node.js versions and reducing the risk of memory leaks.

In Express.js, which method is typically used to handle GET requests to the server?

Option 1:
app.get()

Option 2:
app.post()

Option 3:
app.route()

Option 4:
app.use()

Correct Response:
1.0

Explanation:
In Express.js, app.get() is used to handle HTTP GET requests. It specifies a callback function that will be invoked when a GET request with the specified route is received.

What command is used to install Express.js in a Node.js project?

Option 1:
npm install express

Option 2:
npm init express

Option 3:
npm install express-generator -g

Option 4:
node install express

Correct Response:
1.0

Explanation:
To install Express.js in a Node.js project, the correct command is npm install express.

Can you name the middleware that is used in Express.js to parse the body of incoming requests?

Option 1:
body-parser

Option 2:
request-parser

Option 3:
data-parser

Option 4:
express-parser

Correct Response:
1.0

Explanation:
The middleware used in Express.js to parse the body of incoming requests is body-parser.

How does Koa.js differ from Express.js in terms of middleware execution?

Option 1:
Koa.js executes middleware sequentially

Option 2:
Express.js executes middleware sequentially

Option 3:
Koa.js executes middleware concurrently

Option 4:
Express.js executes middleware concurrently

Correct Response:
4.0

Explanation:
In Koa.js, middleware is executed in a more modern and efficient manner, using async/await, allowing for better control flow. Express.js, on the other hand, follows a traditional sequential execution model. This difference can impact how middleware functions interact with each other and handle asynchronous operations in the context of request processing.

In Express.js, what object is commonly used to define the routes of an application?

Option 1:
app.Routes

Option 2:
app.Router

Option 3:
app.Route

Option 4:
app.Mapping

Correct Response:
2.0

Explanation:
Express.js commonly uses the Router object to define routes. The Router allows developers to modularize and organize routes efficiently. By using routers, you can create more maintainable and scalable applications as routes can be grouped and organized based on functionality, making the codebase more readable and easier to manage.

What would be a reason to choose Koa.js over Express.js for a new project?

Option 1:
Better performance with asynchronous operations

Option 2:
Larger community support

Option 3:
More built-in features than Express.js

Option 4:
Simplicity and lightweight design

Correct Response:
1.0

Explanation:
One reason to choose Koa.js over Express.js for a new project is its superior performance when dealing with asynchronous operations. Koa.js uses async/await for middleware, providing a cleaner and more concise syntax, making it easier to handle asynchronous code. This can be advantageous for projects that heavily rely on asynchronous operations, such as I/O-bound tasks or real-time applications.

Explain how context (ctx) in Koa.js improves upon the request (req) and response (res) objects in Express.js.

Option 1:
ctx encapsulates both the request and response objects

Option 2:
ctx provides additional utilities for request and response

Option 3:
ctx simplifies error handling in Express.js

Option 4:
ctx is identical to req and res in Express.js

Correct Response:
2.0

Explanation:
In Koa.js, the context object (ctx) encapsulates both the request and response objects, providing additional utilities and simplifying the handling of request and response. This abstraction allows for more flexibility and cleaner code compared to the separate req and res objects in Express.js.

Discuss the impact of Express.js's middleware stacking order on request and response handling.

Option 1:
Middlewares are executed in the order they are defined

Option 2:
Middlewares are executed concurrently in Express.js

Option 3:
Middleware order does not affect request handling

Option 4:
Middlewares are executed randomly in Express.js

Correct Response:
1.0

Explanation:
In Express.js, middlewares are executed in the order they are defined. This stacking order is crucial, as it determines the sequence of operations on the request and response. Understanding the middleware order is essential for proper request handling and allows developers to control the flow of data through the middleware stack.

Compare the error handling approaches between Koa.js and Express.js.

Option 1:
Koa.js uses try-catch blocks for error handling

Option 2:
Express.js relies on promises for error handling

Option 3:
Koa.js uses a centralized error handling middleware

Option 4:
Express.js uses synchronous error handling

Correct Response:
3.0

Explanation:
Koa.js and Express.js differ in their error handling approaches. Koa.js uses a centralized error handling middleware, making it easy to manage errors in one place. In contrast, Express.js relies on try-catch blocks and asynchronous error handling. Understanding these differences is crucial for developers transitioning between the two frameworks and choosing the right approach for robust error handling.

In Express.js, the _____ object represents the HTTP request and has properties for the request query string, parameters, body, and HTTP headers.

Option 1:
req

Option 2:
res

Option 3:
next

Option 4:
app

Correct Response:
1.0

Explanation:
In Express.js, the req object is the request object and represents the HTTP request. It contains properties for the request query string, parameters, body, and HTTP headers, allowing you to access and manipulate various aspects of the incoming request.

To mount middleware at a specific path in Express.js, you use _____ method.

Option 1:
use

Option 2:
mount

Option 3:
middleware

Option 4:
route

Correct Response:
4.0

Explanation:
In Express.js, the route method is used to mount middleware at a specific path. It allows you to define middleware that will only be executed for requests that match the specified path, providing a way to modularize and organize your application's middleware.

Koa.js is built by the same team that created Express.js and aims to be a smaller, more _____, and more robust foundation for web applications and APIs.

Option 1:
lightweight

Option 2:
complex

Option 3:
feature-rich

Option 4:
deprecated

Correct Response:
1.0

Explanation:
Koa.js is designed to be a smaller and more lightweight alternative to

Express.js. It offers a more streamlined and expressive syntax for building web applications and APIs, emphasizing a minimalistic and modular approach.

In Koa.js, the _____ method is used to encapsulate middleware functions.

Option 1:
use

Option 2:
middleware

Option 3:
route

Option 4:
compose

Correct Response:
4.0

Explanation:
In Koa.js, the compose method is used to encapsulate middleware functions. It efficiently combines middleware, allowing better control over asynchronous code execution.

Express.js uses _____ routing, which is a way of organizing routes based on their functionality.

Option 1:
functional

Option 2:
path

Option 3:
layered

Option 4:
app

Correct Response:
3.0

Explanation:
Express.js uses layered routing, organizing routes based on functionality and creating a layered structure for better code organization and readability.

One of the key differences in middleware handling between Koa.js and Express.js is that Koa.js supports _____ out of the box, allowing better error handling and cleaner asynchronous code.

Option 1:
async/await

Option 2:
error-first callback

Option 3:
promises

Option 4:
sync/await

Correct Response:
1.0

Explanation:
Koa.js supports async/await out of the box, providing a cleaner way to handle asynchronous code compared to Express.js, which relies on error-first callbacks and promises.

When scaling a web application to handle a large number of routes and requests, what features of Express.js would you leverage?

Option 1:
Middleware for routing and request handling

Option 2:
Cluster module for process management

Option 3:
Template engines for view rendering

Option 4:
Built-in WebSocket support for real-time communication

Correct Response:
1.0

Explanation:
In a large-scale application, middleware is crucial for handling routes

and requests efficiently. Express.js provides a robust middleware system, making it a suitable choice for scaling. Cluster module is used for process management, not route handling. Template engines and WebSocket support are relevant but may not be the primary focus in this context.

If a project requires extensive use of asynchronous functions and you're choosing between Koa.js and Express.js, what factors would influence your decision?

Option 1:
Async/await support and lightweight syntax

Option 2:
Extensive middleware ecosystem

Option 3:
Built-in support for generators

Option 4:
Strong emphasis on callbacks and event-driven architecture

Correct Response:
1.0

Explanation:
Koa.js is known for its native support for async/await, providing a cleaner syntax for handling asynchronous operations. Its lightweight nature also makes it suitable for projects requiring extensive use of asynchronous functions. While Express.js has adopted async/await, Koa.js has a more lightweight syntax. Extensive middleware and generator support are not the primary factors in this decision.

A developer is transitioning from Express.js to Koa.js and is concerned about the learning curve. What aspects of Koa.js could be presented as advantages in terms of application design and development?

Option 1:
Improved error handling and simpler middleware

Option 2:
Robust middleware system and wide adoption

Option 3:
Integration with third-party libraries

Option 4:
Compatibility with Express.js middleware

Correct Response:
1.0

Explanation:
Koa.js is designed with a simpler middleware system and improved error handling, making it more approachable for developers. While Express.js has a robust middleware system, Koa.js provides a cleaner and more modular approach. The compatibility with Express.js middleware can ease the transition for developers. Koa.js is not known for its wide adoption or integration with third-party libraries compared to Express.js.

Which feature of Hapi.js allows you to validate request payload, query, and parameters easily?

Option 1:
Joi validation

Option 2:
Express validation

Option 3:
Koa validation

Option 4:
Meteor validation

Correct Response:
1.0

Explanation:
Hapi.js utilizes Joi for validation, making it easy to validate various parts of the request, including payload, query, and parameters. Joi provides a rich set of validation rules and is widely used in the Hapi.js ecosystem. This feature enhances the overall reliability of the application by ensuring that incoming data meets the expected criteria.

Sails.js is built on top of which popular Node.js framework that makes it ideal for configuring middleware?

Option 1:
Koa.js

Option 2:
Express.js

Option 3:
Meteor.js

Option 4:
Hapi.js

Correct Response:
2.0

Explanation:
Sails.js is built on top of the Express.js framework, which is a widely adopted and versatile Node.js framework. Leveraging Express.js as its foundation, Sails.js inherits its powerful middleware capabilities,

making it well-suited for configuring and customizing middleware functions to enhance the functionality of Sails.js applications.

Hapi.js provides a robust plugin system. What is a common use case for plugins in Hapi.js?

Option 1:
Handling authentication

Option 2:
Modularizing functionality

Option 3:
Templating engines

Option 4:
Database connections

Correct Response:
2.0

Explanation:
A common use case for plugins in Hapi.js is to modularize functionality. Plugins allow developers to organize and encapsulate features, making the codebase more maintainable and extensible. This modular approach facilitates the development process, as each plugin can focus on a specific aspect of the application, promoting code reusability and maintainability.

How does Hapi.js facilitate the management of application state through its lifecycle?

Option 1:
Using request lifecycle events

Option 2:
Through built-in state management features

Option 3:
Managing state manually using JavaScript

Option 4:
State management is not supported in Hapi.js

Correct Response:
1.0

Explanation:
Hapi.js provides request lifecycle events, allowing developers to manage application state by attaching handlers to specific events.

What is the default templating engine provided by Sails.js for server-side rendering, and how can it be changed?

Option 1:
EJS (Embedded JavaScript)

Option 2:
Handlebars

Option 3:
Jade/Pug

Option 4:
Sails.js doesn't support server-side rendering

Correct Response:
1.0

Explanation:
Sails.js uses EJS (Embedded JavaScript) as the default templating engine for server-side rendering. It can be changed by configuring the views setting in the config folder.

In Hapi.js, how are authentication strategies used to secure routes?

Option 1:
Configuring route-specific authentication

Option 2:
Defining authentication in each route handler

Option 3:
Using a global configuration for all routes

Option 4:
Authentication is not supported in Hapi.js

Correct Response:
1.0

Explanation:
Authentication strategies in Hapi.js are applied by configuring route-specific authentication using the config.auth.strategy property in the route options.

Explain how Sails.js enhances real-time application features with WebSockets integration.

Option 1:
Sails.js uses WebSockets for bidirectional communication, allowing instant data updates.

Option 2:
Sails.js relies on HTTP polling for real-time features, ensuring compatibility with all browsers.

Option 3:
Sails.js employs MQTT for real-time functionality, ensuring low-latency communication.

Option 4:
Sails.js uses AJAX calls for real-time updates, ensuring broad cross-browser compatibility.

Correct Response:
1.0

Explanation:
Sails.js leverages WebSockets to establish persistent connections, enabling real-time communication between the server and clients.

This approach reduces latency and allows for instant data updates in applications.

What are the advantages of using Hapi.js's server methods for caching over direct plugin caching mechanisms?

Option 1:
Hapi.js server methods offer better performance and lower latency.

Option 2:
Hapi.js server methods allow for more fine-grained control over caching strategies

Option 3:
Direct plugin caching mechanisms are more efficient than Hapi.js server methods.

Option 4:
Hapi.js server methods lack support for caching in production environments.

Correct Response:
2.0

Explanation:
Hapi.js's server methods provide advantages in terms of

performance and flexibility for caching. They allow developers to implement custom caching strategies and offer better control over caching mechanisms compared to direct plugin caching in Hapi.js.

Describe the role of Waterline in Sails.js and how it contributes to the framework's ability to handle real-time data.

Option 1:
Waterline is a template engine in Sails.js, used for rendering real-time data on the client-side.

Option 2:
Waterline is a database adapter in Sails.js, providing an abstraction layer for handling different databases.

Option 3:
Waterline is a real-time communication library used in Sails.js applications.

Option 4:
Sails.js does not use Waterline for real-time data handling.

Correct Response:
2.0

Explanation:
Waterline in Sails.js serves as a powerful database adapter, offering

an abstraction layer that simplifies database interactions. It contributes to the framework's ability to handle real-time data by providing a consistent interface for working with various databases.

In Hapi.js, _____ extensions can be used to execute custom logic before a request is routed.

Option 1:
pre-handler

Option 2:
middleware

Option 3:
route

Option 4:
post-handler

Correct Response:
2.0

Explanation:
In Hapi.js, middleware extensions allow developers to execute custom logic before a request is routed. These are functions that run in sequence, allowing manipulation of the request or response before reaching the route handler.

Sails.js uses _____ to automatically generate REST APIs for models, simplifying backend development for real-time applications.

Option 1:
Blueprints

Option 2:
Routes

Option 3:
Controllers

Option 4:
Policies

Correct Response:
1.0

Explanation:
Sails.js uses Blueprints to automatically generate REST APIs for models. Blueprints are a set of predefined actions that can be performed on models, such as creating, finding, updating, and

deleting records. This simplifies backend development, especially for real-time applications.

The _____ feature in Hapi.js allows for fine-grained request lifecycle management and the ability to intercept and alter request flow.

Option 1:
Lifecycle

Option 2:
Events

Option 3:
Plugins

Option 4:
Decorators

Correct Response:
1.0

Explanation:
The Lifecycle feature in Hapi.js allows for fine-grained request

lifecycle management. Developers can intercept and alter the request flow at different stages of processing, providing flexibility and control over the request handling process.

Hapi.js's _____ functionality provides a formal way for services to communicate with each other and share processing tasks.

Option 1:
Pub/Sub

Option 2:
WebSocket

Option 3:
IPC (Inter-Process Communication)

Option 4:
Event Emitter

Correct Response:
1.0

Explanation:
Hapi.js provides Pub/Sub functionality through channels, enabling services to communicate and share tasks.

In Sails.js, the use of _____ along with adapters supports the integration of different websocket subprotocols.

Option 1:
Blueprint WebSocket

Option 2:
Policies

Option 3:
Socket.IO

Option 4:
WebSocket Adapter

Correct Response:
3.0

Explanation:
Sails.js leverages Socket.IO to support the integration of different websocket subprotocols through its adapters.

To enhance real-time capabilities in Sails.js, you would utilize _____ to allow users to subscribe to model changes.

Option 1:
Sails.js Hooks

Option 2:
WebSocket

Option 3:
Sails.js Services

Option 4:
Blueprint Pub/Sub

Correct Response:
4.0

Explanation:
Sails.js uses Blueprint Pub/Sub to enhance real-time capabilities, allowing users to subscribe to model changes efficiently.

A developer is designing a Sails.js application that requires real-time messaging. They need to decide between using native WebSockets and Socket.io. What considerations should they make in terms of scalability and browser compatibility?

Option 1:
Scalability may be a concern with native WebSockets, and compatibility might be an issue with certain browsers.

Option 2:
Native WebSockets may be more scalable, but Socket.io ensures

better compatibility with various browsers.

Option 3:
Scalability is not a concern with either option, but browser compatibility might be better with native WebSockets.

Option 4:
Socket.io is more scalable, and native WebSockets offer better browser compatibility.

Correct Response:
2.0

Explanation:
When choosing between native WebSockets and Socket.io, consider that native WebSockets are more scalable, but Socket.io provides better browser compatibility, handling fallbacks for environments that do not support WebSockets.

You are tasked with building a Hapi.js application that requires dynamic route configuration based on external data. How would you leverage Hapi.js features to implement this?

Option 1:
Use Hapi.js plugins to dynamically configure routes based on external data.

Option 2:
Hapi.js does not support dynamic route configuration based on external data.

Option 3:
Utilize Hapi.js middleware to manually configure routes dynamically.

Option 4:
Use Hapi.js route templates to dynamically generate routes from external data.

Correct Response:
1.0

Explanation:
In Hapi.js, dynamic route configuration based on external data can be achieved by using plugins to dynamically configure routes. This allows flexibility and adaptability based on external changes.

In a Sails.js application, you are observing delays in real-time updates when scaling to multiple server instances. What Sails.js feature can you use to synchronize sessions across server instances?

Option 1:
Utilize the Sails.js session store to synchronize sessions across server instances.

Option 2:
Sails.js does not provide features to synchronize sessions across server instances.

Option 3:
Enable the "sticky sessions" feature in the load balancer to address session synchronization issues.

Option 4:
Use Redis as a session store in Sails.js to synchronize sessions across server instances.

Correct Response:
4.0

Explanation:
To address delays in real-time updates when scaling Sails.js to multiple server instances, you can use Redis as a session store. This allows for synchronized sessions across different server instances, enhancing real-time functionality.

In a Node.js application, what is the primary purpose of using GraphQL over a traditional REST API?

Option 1:
Better performance

Option 2:
Strongly-typed queries

Option 3:
Simplicity and ease of use

Option 4:
Asynchronous communication

Correct Response:
2.0

Explanation:
GraphQL provides strongly-typed queries, enabling better validation and optimization. It offers simplicity and ease of use compared to REST.

How do you define a schema in GraphQL for a Node.js application?

Option 1:
Using JavaScript objects

Option 2:
With SQL statements

Option 3:
Using HTML tags

Option 4:
Using regular expressions

Correct Response:
1.0

Explanation:
Schemas in GraphQL are defined using JavaScript objects, outlining the types and their relationships.

Which package is commonly used to implement a WebSocket server in Node.js applications?

Option 1:
http-server

Option 2:
ws

Option 3:
express-ws

Option 4:
socket.io

Correct Response:
2.0

Explanation:
The 'ws' package is commonly used to implement WebSocket servers in Node.js applications.

What is the difference between a query and a mutation in GraphQL when used in a Node.js application?

Option 1:
Queries are for reading data, and mutations are for modifying data.

Option 2:
Queries are only for read operations, and mutations are for write operations.

Option 3:
Mutations are for reading data, and queries are for modifying data.

Option 4:
Both queries and mutations are used for reading and writing data.

Correct Response:
1.0

Explanation:
In GraphQL, queries are used to read data, and mutations are used to modify data. The correct option highlights this fundamental

distinction, ensuring the candidate understands the basic usage of queries and mutations.

How can you secure a WebSocket connection using Socket.io in a Node.js application?

Option 1:
Use secure WebSocket protocol (wss) and configure server-side security measures.

Option 2:
WebSocket connections are inherently secure, so no additional security measures are needed.

Option 3:
Encrypt data on the client-side before sending it over the WebSocket connection.

Option 4:
Rely on HTTP security measures, as WebSocket connections use the same security protocols.

Correct Response:
1.0

Explanation:
Securing a WebSocket connection in a Node.js application involves using the wss protocol and implementing server-side security

measures. This ensures a secure communication channel for real-time data exchange.

Explain how to use GraphQL subscriptions in Node.js to handle real-time data.

Option 1:
Use the PubSub mechanism in GraphQL to facilitate real-time data updates.

Option 2:
GraphQL subscriptions are not suitable for handling real-time data.

Option 3:
Poll the server at regular intervals to check for updates instead of using subscriptions.

Option 4:
Implement WebSocket connections independently of GraphQL for real-time data handling.

Correct Response:
1.0

Explanation:
To handle real-time data in Node.js with GraphQL, one can utilize the PubSub mechanism provided by GraphQL subscriptions. This enables efficient communication and updates between the server and clients in real-time.

Describe the role of resolvers in GraphQL and how they interact with a Node.js backend.

Option 1:
Resolvers are responsible for fetching data from the database.

Option 2:
Resolvers define the data structure of GraphQL queries.

Option 3:
Resolvers handle the business logic and data retrieval for GraphQL queries.

Option 4:
Resolvers are used to manage client-side state in a GraphQL application.

Correct Response:
3.0

Explanation:
In GraphQL, resolvers are functions that determine how data is retrieved or mutated. They are crucial for handling the business logic and interacting with the backend to fetch the required data.

In Socket.io with Node.js, how would you handle the scaling of WebSocket connections across multiple servers or instances?

Option 1:
Use Redis to store and share WebSocket connections.

Option 2:
Utilize the cluster module to distribute connections across multiple Node.js processes.

Option 3:
Employ a load balancer to evenly distribute WebSocket connections.

Option 4:
Use JWT for authentication to ensure secure WebSocket scaling.

Correct Response:
1.0

Explanation:
Scaling WebSocket connections in Socket.io can be achieved by using Redis to share connections among multiple instances, ensuring seamless communication.

Discuss how the N+1 problem can occur in a GraphQL Node.js application and the strategies to mitigate it.

Option 1:
The N+1 problem arises when there is only one Node.js server.

Option 2:
Use batching and caching to reduce the number of database queries.

Option 3:
Increase the server capacity to handle more simultaneous requests.

Option 4:
Avoid using GraphQL for data retrieval to eliminate the N+1 problem.

Correct Response:
2.0

Explanation:
The N+1 problem in GraphQL occurs when multiple database queries are made for each item in a list. Mitigation strategies involve batching and caching to optimize data retrieval and minimize unnecessary queries.

To set up a GraphQL API in a Node.js server, you must define a _____ which specifies how clients can fetch and change data.

Option 1:
Schema

Option 2:
Resolver

Option 3:
Endpoint

Option 4:
Middleware

Correct Response:
1.0

Explanation:
In GraphQL, the schema defines the types and relationships in your API. It includes Query and Mutation types, which represent the read and write operations. The schema acts as a contract between the server and clients.

In Socket.io, the _____ event is emitted when the client successfully completes the handshake and establishes a connection.

Option 1:
connect

Option 2:
disconnect

Option 3:
handshake

Option 4:
connection

Correct Response:
3.0

Explanation:
The 'handshake' event in Socket.io is emitted when a client

successfully completes the handshake process and establishes a connection with the server. This event is useful for handling connection-related tasks.

A _____ in GraphQL is used to fetch multiple resources in a single query from a Node.js backend.

Option 1:
Batch Resolve

Option 2:
DataLoader

Option 3:
Aggregate Query

Option 4:
Batch Fetch

Correct Response:
2.0

Explanation:
DataLoader in GraphQL is used to efficiently batch and cache multiple resource requests into a single query. It helps in avoiding the N+1 query problem and improves the performance of fetching multiple resources.

The Node.js library _____ can be used to automatically generate a GraphQL schema from a Sequelize model.

Option 1:
GraphQLgen

Option 2:
Sequelize-GraphQL

Option 3:
GraphQLize

Option 4:
Apollo-Server-Express

Correct Response:
3.0

Explanation:
In this question, the correct answer is GraphQLize. GraphQLize is a library that helps in generating GraphQL schemas automatically from Sequelize models.

Socket.io uses _____ to handle the fallback of WebSocket connections to long-polling in environments where WebSockets are not supported.

Option 1:
WebSockets

Option 2:
Engine.io

Option 3:
SockJS

Option 4:
WebSocket.io

Correct Response:
2.0

Explanation:
The correct answer is Engine.io. Socket.io uses Engine.io as its

transport layer, which provides fallback mechanisms, such as long-polling, in non-WebSocket environments.

In a Node.js application, the _____ pattern can be used to resolve the N+1 query problem in GraphQL.

Option 1:
DataLoader

Option 2:
BatchLoader

Option 3:
QueryResolver

Option 4:
GraphQLCache

Correct Response:
1.0

Explanation:
The correct answer is DataLoader. DataLoader is a utility that helps in batching and caching GraphQL queries, addressing the N+1 query problem and improving performance.

You need to implement a feature in your Node.js application where changes in the database should be pushed to the client in real-time. Which technology would you use, and how would you implement it?

Option 1:
WebSocket with Socket.io

Option 2:
HTTP Long Polling

Option 3:
Server-Sent Events (SSE)

Option 4:
WebRTC

Correct Response:
1.0

Explanation:
In a real-time scenario, WebSocket with Socket.io is a common choice due to its bidirectional communication capabilities. It allows the server to push updates to the client instantly, providing real-time updates. WebRTC is more suitable for peer-to-peer communication. HTTP Long Polling and SSE are alternatives but may not be as efficient for real-time updates.

If a Node.js server using Socket.io needs to be refactored to support horizontal scaling, what key considerations would you take into account for WebSocket communication?

Option 1:
Implement a centralized message broker (e.g., Redis) for WebSocket communication

Option 2:
Use sticky sessions to ensure WebSocket connections go to the same server in a cluster

Option 3:
Implement WebSocket connection draining for smooth transitions during scaling

Option 4:
Use WebSocket compression to minimize bandwidth usage

Correct Response:
1.0

Explanation:
When scaling horizontally, a centralized message broker like Redis helps manage WebSocket connections across multiple instances. Sticky sessions ensure the same client is always directed to the same server. Connection draining ensures graceful scaling. WebSocket compression reduces bandwidth usage.

A Node.js GraphQL service is facing performance issues due to complex queries fetching too much unnecessary data. How would you optimize the queries?

Option 1:
Use GraphQL query batching to combine multiple queries into a single request

Option 2:
Implement data loader to efficiently batch and cache database queries

Option 3:
Apply depth-limiting and complexity analysis to restrict query depth and complexity

Option 4:
Use persisted queries to save and reuse frequently executed queries

Correct Response:
2.0

Explanation:
GraphQL query batching combines multiple queries into a single request, reducing overhead. Data loader efficiently batches and caches database queries. Depth-limiting and complexity analysis prevent overly complex queries. Persisted queries optimize by saving and reusing frequently executed queries.

What is the primary use of testing frameworks like Mocha and Jest in Node.js development?

Option 1:
Managing package dependencies

Option 2:
Handling asynchronous code

Option 3:
Styling web pages

Option 4:
Creating RESTful APIs

Correct Response:
2.0

Explanation:
Testing frameworks like Mocha and Jest are primarily used for handling asynchronous code in Node.js. They provide tools for writing and running tests, making it easier to ensure the correctness of asynchronous code.

Which templating engine uses a syntax similar to HTML and is known for its simplicity and minimalism?

Option 1:
EJS

Option 2:
Pug (formerly Jade)

Option 3:
Handlebars

Option 4:
Mustache

Correct Response:
3.0

Explanation:
Pug (formerly Jade) is a templating engine for Node.js that uses a syntax similar to HTML. It is known for its simplicity and minimalism, making it easy to write and understand templates.

In Jest, what global function is used to define a block of tests?

Option 1:
describe

Option 2:
it

Option 3:
test

Option 4:
beforeAll

Correct Response:
1.0

Explanation:
In Jest, the global function describe is used to define a block of tests. The describe function groups related tests together, providing a way to structure and organize test suites in Jest.

How do you simulate a function to test its behavior in isolation with Jest?

Option 1:
jest.mock()

Option 2:
jest.spyOn()

Option 3:
jest.setup()

Option 4:
jest.test()

Correct Response:
1.0

Explanation:
In Jest, the jest.mock() function is used to replace a module dependency with a mock implementation, allowing isolated testing of the function's behavior.

When integrating Pug as a templating engine in an Express.js application, which method do you use to render the templates?

Option 1:
res.pug()

Option 2:
res.renderPug()

Option 3:
res.render()

Option 4:
res.template()

Correct Response:
3.0

Explanation:
In Express.js, the correct method to render Pug templates is res.render(). It associates the specified template engine with the given file extension and renders the template.

Explain how you can use Mocha's done callback in asynchronous test cases.

Option 1:
The done callback is not used in Mocha.

Option 2:
Call done() within the test case body.

Option 3:
Pass done as an argument to the test case

Option 4:
done is automatically invoked by Mocha

Correct Response:
2.0

Explanation:
In Mocha, for asynchronous test cases, you can use the done callback. Call done() within the test case body to signal the completion of the test once the asynchronous operations are finished.

What Jest feature would you use to run some code before each test in a suite?

Option 1:
beforeEach()

Option 2:
beforeAll()

Option 3:
setupTestFrameworkScriptFile

Option 4:
beforeTest()

Correct Response:
1.0

Explanation:
Jest provides the beforeEach() function to run code before each test in a suite. This is useful for setting up common test conditions or state. beforeAll() runs code only once before all the tests. setupTestFrameworkScriptFile is deprecated in favor of setupFilesAfterEnv. beforeTest() is not a valid Jest function.

Describe how EJS partials can be utilized to create reusable template fragments.

Option 1:
EJS partials are used to include external CSS files.

Option 2:
EJS partials enable the creation of custom HTML elements.

Option 3:
EJS partials allow the inclusion of other EJS files within a template.

Option 4:
EJS partials provide a way to import JavaScript functions into templates.

Correct Response:
3.0

Explanation:
EJS partials are used to include other EJS files within a template, promoting code reusability. They are not specifically designed for CSS inclusion or creating custom HTML elements. Importing JavaScript functions into templates is typically done using other mechanisms.

Discuss the differences in the assertion libraries that are typically used with Mocha and Jest.

Option 1:
Mocha uses Chai for assertions, while Jest has its built-in expect library.

Option 2:
Both Mocha and Jest use Chai for assertions.

Option 3:
Mocha has no built-in assertion library, while Jest uses Jasmine for assertions.

Option 4:
Jest relies on Chai for assertions, and Mocha has its built-in expect library.

Correct Response:
1.0

Explanation:
Mocha relies on external libraries like Chai for assertions, giving developers flexibility. Jest, on the other hand, comes with its built-in expect library, streamlining the testing setup. The combination of

Mocha and Chai or Jest and expect is a common choice in Node.js testing.

In Mocha, the _____ function is used for grouping related tests together.

Option 1:
describe

Option 2:
context

Option 3:
group

Option 4:
suite

Correct Response:
1.0

Explanation:
In Mocha, the describe function is used to create a test suite and group related tests. It provides a way to structure your tests and add clarity to the test reports.

Jest uses the _____ global function to track the number of times a simulated function is called.

Option 1:
mock

Option 2:
spyOn

Option 3:
trackCalls

Option 4:
jest.mock

Correct Response:
4.0

Explanation:
Jest uses the jest.mock function to mock a module and track the number of times a simulated function is called. This is commonly used for testing functions with external dependencies.

To insert a variable into a Pug template, you would use the #{_____} syntax.

Option 1:
insert

Option 2:
var

Option 3:
include

Option 4:
interpolate

Correct Response:
4.0

Explanation:
In Pug, to insert a variable into a template, you use the #{} syntax for interpolation. This allows dynamic content to be added to the template based on the value of the variable.

The Jest configuration option _____ can be used to collect coverage information.

Option 1:
collectCoverage

Option 2:
coverageOption

Option 3:
testCoverage

Option 4:
coverageCollect

Correct Response:
1.0

Explanation:
In Jest, the correct option is a. collectCoverage. This configuration option allows Jest to collect coverage information during the test execution, helping you analyze code coverage for your project.

To include a mixin in a Pug template, you first have to _____ it.

Option 1:
extend

Option 2:
include

Option 3:
mixin

Option 4:
import

Correct Response:
3.0

Explanation:
In Pug templates, you use the c. mixin keyword to include a mixin. Mixins in Pug are reusable blocks of code that you can include in different parts of your templates.

In an EJS template, the delimiter <%_____%> is used for executing JavaScript code.

Option 1:
js

Option 2:
code

Option 3:
script

Option 4:
ejs

Correct Response:
2.0

Explanation:
In EJS templates, the correct option is b. code. The delimiter <% %> is used to execute JavaScript code within the template, allowing dynamic content to be generated based on server-side logic.

You have a suite of tests written in Mocha that occasionally fail due to timing issues. How would you refactor these tests to improve reliability?

Option 1:
Increase timeouts in problematic tests

Option 2:
Use Mocha's retry mechanism

Option 3:
Implement explicit waits with setTimeout

Option 4:
Adjust the order of test execution

Correct Response:
2.0

Explanation:
Mocha's retry mechanism helps handle intermittent timing issues by

rerunning failed tests.

While rendering a website with EJS, you notice that the HTML output is escaping special characters, causing formatting issues. What EJS feature can you use to prevent this?

Option 1:
<%- ... %>

Option 2:
<%= ... %>

Option 3:
<%# ... %>

Option 4:
<%== ... %>

Correct Response:
1.0

Explanation:
Use <%- ... %> in EJS to unescape and render HTML content, preventing special character issues in the output.

A complex Node.js application you're testing requires a different environment setup for different test scenarios. How would you configure Jest to handle this?

Option 1:
Use Jest configuration files to set environment variables for each scenario

Option 2:
Utilize Jest's test setup and teardown functions for specific environment setups

Option 3:
Create separate Jest configuration files for each test scenario

Option 4:
Implement custom logic within each test to set up the required environment

Correct Response:
2.0

Explanation:
Jest's setup and teardown functions allow for specific environment setups based on the test scenario, improving configurability.

What is a callback in Node.js?

Option 1:
A function passed as an argument to another function to be executed later

Option 2:
A built-in function in Node.js for handling asynchronous tasks

Option 3:
A data type in Node.js used for storing callback functions

Option 4:
A mechanism to handle errors in Node.js

Correct Response:
1.0

Explanation:
In Node.js, a callback is a function passed as an argument to another function, which will be invoked later in the program execution. It is commonly used to handle asynchronous operations, such as reading files or making network requests.

How do promises in Node.js help in handling asynchronous operations?

Option 1:
By providing a way to work with asynchronous code in a more synchronous manner

Option 2:
By blocking the event loop until the asynchronous operation is complete

Option 3:
By eliminating the need for callbacks

Option 4:
By enforcing strict sequential execution of code

Correct Response:
3.0

Explanation:
Promises in Node.js provide a cleaner and more readable syntax for handling asynchronous operations. They eliminate callback hell and allow chaining of asynchronous tasks in a more sequential manner.

What will happen if a promise is rejected and there is no .catch() block attached?

Option 1:
The program will terminate with an unhandled promise rejection warning

Option 2:
The promise will automatically retry the operation

Option 3:
The rejected promise will be silently ignored

Option 4:
The event loop will be blocked indefinitely

Correct Response:
1.0

Explanation:
If a promise is rejected, and there is no .catch() block attached to handle the rejection, it will result in an unhandled promise rejection warning. It's essential to handle promise rejections to prevent unexpected issues in the application.

When converting a callback-based asynchronous function to return a promise, which Node.js utility function can be used?

Option 1:
util.promisify

Option 2:
async.promisify

Option 3:
node.promisify

Option 4:
callback.promisify

Correct Response:
1.0

Explanation:
In Node.js, the util.promisify function is used to convert callback-based asynchronous functions into functions that return promises. This allows for better handling of asynchronous operations using the Promise API.

What is the correct order of states that a promise can transition through?

Option 1:
Pending, Fulfilled, Rejected

Option 2:
Fulfilled, Pending, Rejected

Option 3:
Rejected, Pending, Fulfilled

Option 4:
Fulfilled, Rejected, Pending

Correct Response:
1.0

Explanation:
The correct order of states for a promise is Pending, Fulfilled, and Rejected. A promise starts in the pending state and can transition to either fulfilled or rejected based on the outcome of the asynchronous operation it represents.

How does Node.js handle unhandled promise rejections?

Option 1:
It automatically logs the rejection stack trace to the console.

Option 2:
It ignores unhandled rejections.

Option 3:
It throws an error.

Option 4:
It triggers the unhandledRejection event.

Correct Response:
4.0

Explanation:
Node.js emits the unhandledRejection event when a promise rejection is not handled. It provides an opportunity to log the error or take appropriate action. Ignoring unhandled rejections can lead to unexpected behavior in the application.

What is the potential pitfall of using Promise.all() with an array of promises in Node.js?

Option 1:
Overlooking individual promise rejections

Option 2:
Ignoring the order of promises in the array

Option 3:
Failing to handle resolved values correctly

Option 4:
Using it only for asynchronous operations

Correct Response:
1.0

Explanation:
When using Promise.all(), if any of the promises in the array is rejected, the entire Promise.all() is rejected. It's important to handle individual promise rejections to identify which promise failed. This is crucial for error handling and debugging.

How does the event loop prioritize promise callbacks (microtasks) compared to other callbacks like setImmediate() or setTimeout() (macrotasks)?

Option 1:
Microtasks have higher priority and are executed before macrotasks

Option 2:
Macrotasks are prioritized, and microtasks are executed later

Option 3:
Microtasks and macrotasks are executed concurrently

Option 4:
Microtasks are executed only after all macrotasks are completed

Correct Response:
1.0

Explanation:
Microtasks, including promise callbacks, are executed before

macrotasks in the event loop. Understanding this priority is essential for managing the order of execution in asynchronous code.

In Node.js, how can you avoid callback hell when dealing with nested asynchronous operations?

Option 1:
Use Promises and async/await

Option 2:
Increase the nesting depth to make the code more readable

Option 3:
Utilize only traditional callback functions

Option 4:
Mix synchronous and asynchronous code to simplify nesting

Correct Response:
1.0

Explanation:
Callback hell, or the pyramid of doom, can be avoided by using Promises and async/await. This improves code readability and maintainability by handling asynchronous operations in a more structured and linear fashion.

To handle errors in a promise chain, you attach a _____ block at the end of the chain.

Option 1:
catch

Option 2:
finally

Option 3:
resolve

Option 4:
reject

Correct Response:
1.0

Explanation:
In a promise chain, the catch block is used to handle errors that occur in any of the preceding promises. It allows you to gracefully handle and process errors at the end of the chain.

A _____ is a function that allows you to attach success and failure handlers to an asynchronous operation in Node.js.

Option 1:
Promise

Option 2:
Callback

Option 3:
Async

Option 4:
Await

Correct Response:
1.0

Explanation:
The Promise is a constructor function in Node.js that allows you to work with asynchronous operations more easily. It allows you to attach handlers for success and failure, making it easier to manage asynchronous code.

The method _____ is used to wait for all promises in an iterable to either resolve or for one to reject.

Option 1:
Promise.race

Option 2:
Promise.all

Option 3:
Promise.any

Option 4:
Promise.resolve

Correct Response:
2.0

Explanation:
The Promise.all method is used to wait for all promises in an iterable (e.g., an array of promises) to either resolve or for one to reject. It returns a single promise that resolves with an array of the results when all the input promises have resolved.

In a promise, the executor function takes two functions as arguments, commonly referred to as _____ and _____.

Option 1:
Resolve, Reject

Option 2:
Done, Fail

Option 3:
Success, Error

Option 4:
Then, Catch

Correct Response:
1.0

Explanation:
In a promise, the executor function typically receives two parameters: a resolve function and a reject function. The resolve function is called when the promise is fulfilled, and the reject function is called when the promise is rejected.

The static method _____ on the Promise object is used to handle multiple promises that may not be related and don't depend on each other's completion.

Option 1:
Promise.all

Option 2:
Promise.race

Option 3:
Promise.sequence

Option 4:
Promise.combine

Correct Response:
1.0

Explanation:
Promise.all is used to handle an array of promises and is fulfilled only if all the input promises are fulfilled. It's useful for scenarios where multiple asynchronous operations need to complete before proceeding.

To convert an array of callback-based functions to promises, you might use _____ from Node.js utilities.

Option 1:
util.promisify

Option 2:
async.promisify

Option 3:
callback.promisify

Option 4:
convert.promisify

Correct Response:
1.0

Explanation:
util.promisify is a utility function in Node.js that converts a callback-based function to a promise-based one. It simplifies working with asynchronous code by allowing the use of promises instead of callbacks.

You have a list of URLs to fetch data from in your Node.js application. How would you ensure that you proceed only after all the fetch operations have completed successfully, or handle the case where any one of them fails?

Option 1:
Using async/await with Promise.all

Option 2:
Using callbacks

Option 3:
Using setInterval with a counter for successful completions

Option 4:
Using the 'async' library for parallel execution of fetch operations

Correct Response:
1.0

Explanation:
Async/Await with Promise.all ensures that all promises are settled, either resolved or rejected, before proceeding.

A Node.js service you are developing needs to process tasks in sequence, where each task depends on the output of the previous one. What approach would you take to structure the code for these asynchronous tasks?

Option 1:
Using the 'async' library with the waterfall function

Option 2:
Using Promises with 'then' chaining

Option 3:
Using nested callbacks

Option 4:
Using the 'async' library with the parallel function

Correct Response:
2.0

Explanation:
Using Promises with 'then' chaining ensures sequential execution, as each 'then' waits for the previous one to complete.

When refactoring legacy Node.js code that heavily uses nested callbacks, you decide to implement promises. What should you be careful about during this refactoring process?

Option 1:
Handling error propagation by chaining '.catch' to each promise in the chain

Option 2:
Replacing all callbacks with promises

Option 3:
Ignoring error handling, as promises handle errors automatically

Option 4:
Using the 'async' library instead of promises for backward compatibility

Correct Response:
1.0

Explanation:
Carefully chaining '.catch' ensures proper error handling throughout the promise chain, preventing unhandled promise rejections.

What must precede the function keyword to use async/await syntax inside a function in Node.js?

Option 1:
async

Option 2:
await

Option 3:
promise

Option 4:
callback

Correct Response:
1.0

Explanation:
In Node.js, the async keyword must precede the function keyword to enable the use of await inside the function. It indicates that the function will return a promise. The await keyword is used to pause the execution until the promise is resolved.

Which Node.js core module is primarily used for creating custom event emitters?

Option 1:
fs

Option 2:
events

Option 3:
http

Option 4:
path

Correct Response:
2.0

Explanation:
The events module in Node.js is primarily used for creating custom event emitters. Event emitters allow objects to emit and listen for custom events, providing a way to implement the observer pattern in Node.js applications.

What is the output of an async function in Node.js?

Option 1:
undefined

Option 2:
Promise

Option 3:
Callback

Option 4:
AsyncFunction

Correct Response:
2.0

Explanation:
An async function in Node.js returns a Promise. When called, it executes asynchronously and returns a Promise that is either resolved with the function's return value or rejected with an exception thrown by the function.

How do you properly handle errors when using async/await in Node.js?

Option 1:
Use try-catch block

Option 2:
Use if-else statements

Option 3:
Use switch-case statements

Option 4:
Use for loop

Correct Response:
1.0

Explanation:
When using async/await in Node.js, errors can be handled using a try-catch block. This allows you to catch and handle any exceptions that might occur during the asynchronous operation. If an error occurs within the try block, it can be caught in the corresponding catch block.

What method is used to emit an event from an instance of an event emitter in Node.js?

Option 1:
dispatchEvent

Option 2:
triggerEvent

Option 3:
emitEvent

Option 4:
emit

Correct Response:
4.0

Explanation:
The emit method is used to emit an event from an instance of an event emitter in Node.js. This method takes the event name as its first argument and additional arguments that will be passed to the event listeners.

What is a possible consequence of not handling the 'error' event on an EventEmitter instance in Node.js?

Option 1:
Silent failures

Option 2:
Automatic error handling

Option 3:
Graceful degradation

Option 4:
Improved performance

Correct Response:
1.0

Explanation:
Not handling the 'error' event on an EventEmitter instance in Node.js can lead to silent failures. If an error occurs and there's no listener for the 'error' event, the error will be uncaught, potentially

causing the application to behave unexpectedly or crash without providing any indication of the issue.

How does Node.js handle uncaught exceptions from async functions when using the async/await syntax?

Option 1:
Promise rejection is treated as an unhandled exception and terminates the process.

Option 2:
Node.js provides the process.on('unhandledRejection') event to handle uncaught exceptions.

Option 3:
Unhandled promise rejections are automatically caught by the event loop.

Option 4:
Node.js ignores unhandled promise rejections.

Correct Response:
2.0

Explanation:
In Node.js, unhandled promise rejections lead to the termination of the process. To handle such exceptions, you can use the process.on('unhandledRejection') event, which allows you to log the

error or perform any necessary cleanup. This is crucial for preventing unexpected process termination.

When using event emitters, how can you ensure a listener only executes once?

Option 1:
Using the 'once' method to register the listener.

Option 2:
Manually tracking executions within the listener.

Option 3:
Setting a flag within the listener to check execution status.

Option 4:
Removing the listener after its first execution.

Correct Response:
1.0

Explanation:
To ensure a listener executes only once in Node.js event emitters, you can use the 'once' method when registering the listener. This automatically removes the listener after its first execution, preventing subsequent invocations.

In what scenario might you prefer to use async/await over traditional event emitters in Node.js?

Option 1:
Handling sequential asynchronous operations.

Option 2:
Managing concurrent tasks with fine-grained control.

Option 3:
Dealing with real-time events and callbacks.

Option 4:
Optimizing for maximum parallelism.

Correct Response:
1.0

Explanation:
Async/await is suitable for handling sequential asynchronous operations where one operation depends on the result of the previous one. It simplifies code readability and structure, making it preferable in scenarios where tasks need to be executed sequentially.

To wait for a set of parallel async operations to complete, you can use Promise.all with the _____ syntax.

Option 1:
allSettled

Option 2:
all

Option 3:
race

Option 4:
resolve

Correct Response:
2.0

Explanation:
In Node.js, when you want to wait for multiple promises to complete, you use Promise.all along with an array of promises. The correct syntax is Promise.all(iterable), where the iterable is an array of promises. So, the correct option is all.

When creating a custom event emitter, you must extend the _____ class from the 'events' module.

Option 1:
EventEmitter

Option 2:
EventDispatcher

Option 3:
Emitter

Option 4:
EventSource

Correct Response:
1.0

Explanation:
The 'events' module in Node.js provides an EventEmitter class. When creating a custom event emitter, you should extend the EventEmitter class to inherit its functionality. Therefore, the correct option is EventEmitter.

An async function in Node.js returns a _____, which can be awaited or used with .then() and .catch().

Option 1:
Promise

Option 2:
Callback

Option 3:
Async

Option 4:
Function

Correct Response:
1.0

Explanation:
In Node.js, an async function returns a Promise. This allows you to use the await keyword to wait for the asynchronous operation to complete or use the traditional promise methods like .then() and .catch(). Therefore, the correct option is Promise.

If an async function throws an error and there is no catch block, Node.js will emit an 'unhandledRejection' event on the _____ object.

Option 1:
process

Option 2:
global

Option 3:
EventEmitter

Option 4:
Promise

Correct Response:
2.0

Explanation:
When an unhandled promise rejection occurs, Node.js emits an 'unhandledRejection' event on the global object, which is accessible through the global variable or globalThis in modern JavaScript.

The _____ pattern can be used to convert callback-based functions to return a promise for use with async/await.

Option 1:
Callback Hell

Option 2:
Promisification

Option 3:
Event Emitter

Option 4:
Observer

Correct Response:
2.0

Explanation:
Promisification is the pattern used to convert callback-based functions to return a promise. It simplifies the usage of asynchronous code with async/await syntax, making it more readable and maintainable.

To ensure a piece of code is executed after an async operation, despite the outcome, you should use a _____ block in an async function.

Option 1:
finally

Option 2:
catch

Option 3:
then

Option 4:
reject

Correct Response:
1.0

Explanation:
The 'finally' block is used in an async function to ensure a piece of

code is executed after the completion of the async operation, regardless of whether it resolves or rejects.

You have an async function that processes a request, but you need to ensure that it times out if the operation takes too long. How do you implement this using async/await?

Option 1:
Use Promise.race with a promise for the timeout

Option 2:
Use setTimeout with a callback function inside the async function

Option 3:
Use Promise.timeout method

Option 4:
Implement a custom timeout mechanism using async/await

Correct Response:
1.0

Explanation:
The correct option is to use Promise.race with a promise for the timeout. This allows you to race between the main asynchronous operation and a timeout promise, ensuring that the function times out if the operation takes too long.

You're designing a chat application using Node.js and need to implement a feature where a user's status is broadcasted when they come online. Which pattern involving event emitters would be most suitable for this?

Option 1:
Observer Pattern

Option 2:
Mediator Pattern

Option 3:
Publisher-Subscriber Pattern

Option 4:
Singleton Pattern

Correct Response:
3.0

Explanation:
The correct option is the Publisher-Subscriber Pattern. This pattern involves a central event bus that allows components to subscribe and publish events. In the context of a chat application, the user's status change can be an event broadcasted to subscribers.

A Node.js service you're maintaining is facing issues due to unhandled promise rejections. What changes would you make to gracefully handle these rejections in async functions?

Option 1:
Use process.on('unhandledRejection') to handle unhandled promise rejections globally

Option 2:
Add a .catch block to every async function to handle promise rejections

Option 3:
Use the --unhandled-rejections flag when starting the Node.js process

Option 4:
Set NODE_ENV to 'production' to automatically handle unhandled promise rejections

Correct Response:
2.0

Explanation:
The correct option is to add a .catch block to every async function to handle promise rejections. This ensures that any unhandled promise rejections within the async function are caught and handled gracefully.

When an error occurs within a callback function in Node.js, how should it typically be handled?

Option 1:
Use try-catch block

Option 2:
Return an error object to the callback

Option 3:
Log the error to the console

Option 4:
Ignore the error and continue execution

Correct Response:
1.0

Explanation:
In Node.js, errors within a callback should be handled using a try-catch block. This allows you to catch and handle exceptions within the asynchronous code. Returning an error to the callback, logging it, or ignoring it may lead to unhandled exceptions.

What Node.js global function is used to schedule the execution of a callback after a set number of milliseconds?

Option 1:
setTimeout

Option 2:
setInterval

Option 3:
setImmediate

Option 4:
process.nextTick

Correct Response:
1.0

Explanation:
The setTimeout function is used to schedule the execution of a callback after a specified number of milliseconds. It is commonly used for asynchronous operations in Node.js to introduce delays or schedule code to run in the future.

Which object is provided by Node.js to handle a series of asynchronous operations in a sequence?

Option 1:
Promise

Option 2:
EventEmitter

Option 3:
Buffer

Option 4:
Stream

Correct Response:
1.0

Explanation:
The Promise object is used in Node.js to handle a series of asynchronous operations in a sequence. It represents a value that may be available now, or in the future, or never. It allows for more structured and readable asynchronous code using then and catch.

In Node.js, what is the standard pattern for propagating errors in asynchronous callbacks?

Option 1:
Callback function

Option 2:
Error-first Callback

Option 3:
Promise

Option 4:
Event Emitters

Correct Response:
2.0

Explanation:
The standard pattern for propagating errors in asynchronous callbacks in Node.js is the "Error-first Callback" pattern. In this pattern, the first parameter of a callback function is reserved for an error object. If an error occurs during the asynchronous operation, it is passed as the first argument to the callback. This allows the developer to check for errors and handle them appropriately in the

callback function. This pattern is widely used in Node.js for handling asynchronous operations.

How can you use the uncaughtException event to handle exceptions that were not caught within asynchronous operations?

Option 1:
Register a callback function using process.on()

Option 2:
Use try...catch blocks

Option 3:
Attach the callback function to the EventEmitter

Option 4:
Set the uncaughtException property

Correct Response:
1.0

Explanation:
To handle exceptions that were not caught within asynchronous operations using the uncaughtException event in Node.js, you can register a callback function using process.on('uncaughtException', callback). This callback will be invoked whenever an unhandled

exception occurs, allowing you to log the error or perform any necessary cleanup before the process exits. However, it's important to note that using uncaughtException is not a recommended practice due to its global nature and potential for unintended consequences.

What is the main difference between setTimeout() and setInterval() functions in Node.js's timer functions?

Option 1:
setTimeout() triggers the callback function once

Option 2:
setTimeout() is used for delaying a function call

Option 3:
setInterval() triggers the callback function repeatedly

Option 4:
setInterval() is used for delaying a function call

Correct Response:
3.0

Explanation:
The main difference between setTimeout() and setInterval() functions in Node.js's timer functions lies in how they handle the execution of the callback function. setTimeout() triggers the callback function once after the specified delay, while setInterval() triggers the callback function repeatedly at intervals specified by the delay. Developers need to consider this difference when choosing between the two functions based on the desired behavior in their applications.

Explain how the process.nextTick() function can impact error handling within Node.js applications.

Option 1:
It allows immediate execution of a callback after the current event loop cycle.

Option 2:
It delays the execution of a callback until the next event loop cycle.

Option 3:
It is used for handling synchronous code only.

Option 4:
It is equivalent to setTimeout(callback, 0) for asynchronous code.

Correct Response:
1.0

Explanation:
The process.nextTick() function is used to schedule a callback to be invoked in the next iteration of the event loop, before I/O events. This allows for immediate execution, impacting error handling by enabling quick response to errors in the current cycle.

What are the implications of using try/catch blocks within asynchronous code in Node.js?

Option 1:
It ensures synchronous execution of asynchronous code.

Option 2:
It helps in catching errors from asynchronous operations.

Option 3:
It has no impact on error handling in asynchronous code.

Option 4:
It can lead to unhandled exceptions and silent failures.

Correct Response:
4.0

Explanation:
Using try/catch blocks in asynchronous code can lead to unhandled exceptions and silent failures because they may not capture errors thrown in asynchronous operations. It's crucial to use alternative error-handling mechanisms, such as Promises or the error event, in asynchronous code.

How does the Node.js event loop handle timer functions with zero delay, and what could be a potential pitfall of using setTimeout(callback, 0)?

Option 1:
It executes the callback immediately, bypassing the event loop.

Option 2:
It schedules the callback to be executed in the next iteration of the event loop.

Option 3:
It delays the execution until all other tasks in the event loop are complete.

Option 4:
It triggers an error as zero delay is not allowed.

Correct Response:
2.0

Explanation:
The event loop in Node.js handles timer functions with zero delay by scheduling the callback to be executed in the next iteration of the event loop. A potential pitfall of using setTimeout(callback, 0) is that it doesn't guarantee immediate execution and can be influenced by other tasks in the event loop.

To handle errors in a promise-based workflow, you should use the .catch() method or a(n) _____ block.

Option 1:
error

Option 2:
except

Option 3:
finally

Option 4:
reject

Correct Response:
3.0

Explanation:
In a promise-based workflow, the .catch() method or a finally block can be used to handle errors. The finally block is executed regardless of whether the promise is resolved or rejected, making it suitable for cleanup tasks.

Node.js's _____ function can be used to execute a block of code only once after a specified delay.

Option 1:
setInterval

Option 2:
setImmediate

Option 3:
setTimeout

Option 4:
delay

Correct Response:
3.0

Explanation:
The setTimeout function in Node.js is used to execute a block of code once after a specified delay. It allows you to schedule the execution of a function after a given time, making it useful for tasks like delayed execution or timeouts.

The _____ event can be used to capture asynchronous errors that are not caught by any other means.

Option 1:
error

Option 2:
unhandledRejection

Option 3:
exception

Option 4:
uncaughtException

Correct Response:
2.0

Explanation:
The unhandledRejection event in Node.js can be used to capture asynchronous errors that are not caught by any other means, such as unhandled promise rejections. It provides a way to handle and log such errors in the application.

Node.js executes the timers in the _____ phase of the event loop.

Option 1:
Callback

Option 2:
Timer

Option 3:
Poll

Option 4:
Check

Correct Response:
2.0

Explanation:
In Node.js, timers are executed in the Timer phase of the event loop. This phase handles callbacks scheduled by setTimeout and setInterval.

Asynchronous errors that occur inside a(n) _____ callback are not caught by a surrounding try/catch block because the try/catch block is synchronous.

Option 1:
Promises

Option 2:
Asynchronous

Option 3:
Error

Option 4:
Callback

Correct Response:
4.0

Explanation:
Asynchronous errors inside a callback are not caught by a surrounding try/catch block because the try/catch block is synchronous and won't capture asynchronous errors.

Using the Node.js _____ utility module, you can wrap asynchronous functions and ensure errors are passed to a callback.

Option 1:
Async

Option 2:
Util

Option 3:
Promise

Option 4:
Callback

Correct Response:
1.0

Explanation:
The util module in Node.js provides the promisify function, allowing you to convert callback-based functions into Promises. This helps in handling errors in a more readable and consistent manner.

You're reviewing a Node.js application where some asynchronous errors are not being logged. What changes would you make to ensure that all errors are properly caught and logged?

Option 1:
Utilize a global unhandledRejection event listener

Option 2:
Use try-catch blocks around asynchronous code

Option 3:
Implement a custom error handler for asynchronous operations

Option 4:
Set up a process.on('uncaughtException') event listener

Correct Response:
1.0

Explanation:
In Node.js, the 'unhandledRejection' event is emitted whenever a Promise is rejected but no error handler is attached. Implementing a global unhandledRejection listener can help catch and log such errors.

A developer has used setTimeout() with a delay of zero milliseconds in an attempt to defer an operation to the next tick of the Event Loop. Explain why this might not always result in the operation being executed on the next tick.

Option 1:
setTimeout with zero delay does not guarantee immediate execution due to the minimum timer resolution in browsers and Node.js

Option 2:
setTimeout with zero delay may result in execution in the current tick if the event loop is not busy

Option 3:
The use of setTimeout for deferred execution is deprecated, and setImmediate should be used instead

Option 4:
Using setInterval with zero delay is a more reliable way to ensure immediate execution

Correct Response:
1.0

Explanation:
setTimeout with zero delay does not guarantee immediate execution due to the minimum timer resolution in browsers and Node.js. It may be delayed to the next tick, but it's not a reliable way to schedule code for the next tick.

How would you handle a scenario where an asynchronous operation might throw an error that is not passed to the callback function?

Option 1:
Attach an error event listener to the asynchronous operation

Option 2:
Wrap the asynchronous operation in a try-catch block

Option 3:
Use the process.on('uncaughtException') event to catch unhandled errors

Option 4:
Utilize a Promise-based approach and handle the rejection using .catch()

Correct Response:
3.0

Explanation:
The 'uncaughtException' event can be used to catch unhandled errors globally. However, it is considered a last resort and should be used cautiously as it can lead to an inconsistent state in the application.

What is a stream in Node.js and how is it different from traditional data handling methods?

Option 1:
Flow of data in chunks, enhancing performance

Option 2:
Synchronous processing, blocking the event loop

Option 3:
Event-driven, handling data in chunks asynchronously

Option 4:
Linear processing, reading data sequentially

Correct Response:
3.0

Explanation:
In Node.js, a stream is a flow of data in chunks, providing efficient handling of large amounts of data. Unlike traditional methods that may block the event loop, streams are event-driven, allowing asynchronous handling of data in chunks, enhancing performance.

What is the primary use case for using async hooks in Node.js?

Option 1:
Tracing and debugging asynchronous operations

Option 2:
Synchronous operations tracking

Option 3:
Enhancing code readability and organization

Option 4:
Performance optimization

Correct Response:
1.0

Explanation:
Async hooks in Node.js are primarily used for tracing and debugging asynchronous operations. They allow developers to monitor and trace asynchronous events, aiding in the identification and resolution of issues related to asynchronous code.

How can you listen for data chunks in a readable stream in Node.js?

Option 1:
Using the 'data' event

Option 2:
Utilizing 'read' method

Option 3:
Listening for 'onData' event

Option 4:
Subscribing to 'chunk' event

Correct Response:
1.0

Explanation:
You can listen for data chunks in a readable stream in Node.js by using the 'data' event. This event is emitted whenever data is available to be read, allowing you to handle data chunks as they become available in the stream.

application?

Option 1:
It prevents data loss by slowing down the data flow

Option 2:
It increases memory usage for better performance

Option 3:
It doesn't affect memory efficiency

Option 4:
It causes crashes in the application

Correct Response:
1.0

Explanation:
Backpressure in streams is a mechanism that allows a slower consumer to control the rate of data flow, preventing overwhelming the faster producer. It helps maintain memory efficiency by avoiding the accumulation of excessive data in memory, reducing the risk of out-of-memory errors. The correct option, (a), highlights the purpose of backpressure.

What is the purpose of the init async hook in Node.js?

Option 1:
To initialize variables before the event loop starts

Option 2:
To perform asynchronous initialization tasks

Option 3:
To handle errors during initialization

Option 4:
To manage global event handlers

Correct Response:
2.0

Explanation:
The init async hook in Node.js is used for performing asynchronous initialization tasks before the event loop begins. It is particularly useful for setting up resources or configurations that are needed before the application starts processing events. Option (b) correctly captures the purpose of the init async hook.

Describe how you would use a Transform stream in a Node.js application.

Option 1:
To read data from a file

Option 2:
To modify or manipulate data during streaming

Option 3:
To create a readable stream

Option 4:
To handle HTTP requests

Correct Response:
2.0

Explanation:
A Transform stream in Node.js is used to modify or manipulate data during streaming. It reads input, performs transformations, and produces output. This is useful for tasks like data encryption, compression, or formatting. Option (b) accurately describes the purpose of a Transform stream.

Explain how you would implement a custom readable stream in Node.js.

Option 1:
Implementing a readable stream involves extending the 'Readable' class, defining a '_read' method, and pushing data using 'this.push'.

Option 2:
Creating a custom readable stream is achieved by using the 'readable' event and emitting it when data is available to be read.

Option 3:
Utilizing the 'Transform' stream class is the recommended way to create custom readable streams, as it provides flexibility and ease of use.

Option 4:
Custom readable streams can be achieved using 'fs.createReadStream' with additional customization using the 'transform' function.

Correct Response:
1.0

Explanation:
In Node.js, a custom readable stream can be implemented by extending the 'Readable' class, defining the '_read' method to push data, and handling events such as 'end' and 'error' for proper stream

management. This approach provides a clean and efficient way to create custom readable streams for various data sources.

How does the async_hooks module's executionAsyncId function aid in tracking asynchronous resources?

Option 1:
The 'executionAsyncId' function in the 'async_hooks' module returns the ID of the current execution context, facilitating the tracking of asynchronous resources across asynchronous operations.

Option 2:
By using 'executionAsyncId,' the async_hooks module allows developers to correlate asynchronous operations and trace their lifecycle across different resource types.

Option 3:
The 'executionAsyncId' function is not directly related to tracking asynchronous resources; instead, it is used for debugging asynchronous code.

Option 4:
'executionAsyncId' retrieves the asynchronous ID, which can be utilized to identify the current asynchronous context in a given execution flow.

Correct Response:
2.0

Explanation:
The 'executionAsyncId' function in the 'async_hooks' module plays a crucial role in tracking asynchronous resources in Node.js. It returns the ID of the current execution context, allowing developers to correlate and trace asynchronous operations across various resource types, enabling better debugging and understanding of asynchronous code execution.

Discuss how stream pipelines can be used to efficiently compose multiple streams together in Node.js.

Option 1:
Stream pipelines in Node.js enable the seamless composition of multiple streams by using the 'pipeline' function, ensuring efficient data transfer and handling.

Option 2:
Connecting streams using the 'pipe' method is the recommended approach for stream composition, providing a straightforward way to link multiple streams together.

Option 3:
Stream composition involves manually managing data flow between streams, and the 'pipeline' function is not essential for this purpose.

Option 4:
Stream pipelines are not a recommended practice, as they may introduce complexity and performance issues in data processing.

Correct Response:
1.0

Explanation:
In Node.js, stream pipelines offer an efficient way to compose

multiple streams together. The 'pipeline' function simplifies the process by handling error propagation, ensuring proper stream closure, and providing a clean syntax for connecting streams. This approach enhances code readability and maintainability when dealing with complex data processing scenarios using streams.

To handle stream errors in Node.js, you should listen to the _____ event.

Option 1:
errorEvent

Option 2:
streamError

Option 3:
error

Option 4:
onError

Correct Response:
3.0

Explanation:
In Node.js, the error event is emitted when an error occurs in a stream. It's essential to listen for this event to handle and log errors effectively.

The _____ method in a writable stream is used when the stream needs to process the buffered data.

Option 1:
flush

Option 2:
drain

Option 3:
process

Option 4:
clear

Correct Response:
2.0

Explanation:
The drain method is invoked when it's appropriate to resume writing data to a writable stream. It signals that the buffer is empty and ready to receive more data.

The async hook's _____ callback is invoked when an asynchronous operation is completed.

Option 1:
before

Option 2:
after

Option 3:
complete

Option 4:
finish

Correct Response:
2.0

Explanation:
The after callback in async hooks is called when an asynchronous operation initiated by a hook is completed. It allows you to perform actions after the asynchronous operation finishes.

Node.js _____ function is a utility that helps to pipe a series of streams together while handling the errors.

Option 1:
pipeline

Option 2:
stream

Option 3:
connect

Option 4:
concat

Correct Response:
1.0

Explanation:
In Node.js, the pipeline function is used to pipe a series of streams together, providing a convenient way to handle errors during the process. It simplifies the chaining of streams and ensures proper error propagation.

In a custom stream implementation, the _____ function must be provided to supply the stream with data.

Option 1:
write

Option 2:
end

Option 3:
read

Option 4:
push

Correct Response:
4.0

Explanation:
In a custom stream implementation, the push function is used to supply the stream with data. This function is called to push chunks of data into the stream, making it available for further processing.

Async hooks are initialized within an async function by invoking _____.

Option 1:
initHooks

Option 2:
registerHooks

Option 3:
enableHooks

Option 4:
createHooks

Correct Response:
3.0

Explanation:
Async hooks in Node.js are initialized within an async function by invoking the enableHooks function. This sets up the environment for utilizing async hooks, allowing developers to track asynchronous operations in their applications.

You need to process a large CSV file and convert each row into an object. How would you implement this using Node.js streams to ensure low memory usage?

Option 1:
Utilize the 'csv-parser' library with the 'stream' module to create a readable stream and parse each row into an object.

Option 2:
Use the 'fs' module to read the CSV file synchronously and then process each row to create an object.

Option 3:
Implement a simple 'for' loop to read the CSV file line by line, converting each row into an object.

Option 4:
Use the 'Buffer' module to load the entire CSV file into memory and then convert each row into an object.

Correct Response:
1.0

Explanation:
The 'csv-parser' library with the 'stream' module is efficient for large files. It creates a readable stream, ensuring low memory usage by processing rows individually. Synchronous methods and loading the entire file into memory can lead to high memory usage, especially with large files. Using streams allows for better scalability and resource management.

During debugging, you find that asynchronous operations are not being tracked as expected. What async hook lifecycle events would you inspect?

Option 1:
'init', 'before', 'after', and 'destroy' events.

Option 2:
'before', 'after', 'promiseResolve', and 'promiseReject' events.

Option 3:
'init', 'before', 'after', and 'promiseResolve' events.

Option 4:
'init', 'before', 'after', and 'promiseError' events.

Correct Response:
3.0

Explanation:
Inspecting 'init', 'before', 'after', and 'promiseResolve' events can help

identify the flow of asynchronous operations. The 'promiseError' event is not a part of the async hook lifecycle. It's important to track initialization, before and after execution, and resolution of promises to understand the asynchronous flow during debugging.

Your Node.js application requires the integration of several data processing steps that need to be executed in sequence. How would you organize these steps using Node.js streams to ensure optimal performance?

Option 1:
Create separate readable and writable streams for each processing step and pipe them together to form a pipeline.

Option 2:
Use 'async/await' with 'Promise.all()' to execute processing steps concurrently.

Option 3:
Implement a single stream to process all steps sequentially.

Option 4:
Use the 'cluster' module to create child processes for each processing step and run them concurrently.

Correct Response:
1.0

Explanation:
Creating separate streams for each processing step and piping them together ensures a modular and efficient pipeline. It allows for parallel execution of each step, maximizing performance. Concurrent processing with 'async/await' and 'Promise.all()' may not guarantee optimal performance as it depends on the nature of the processing steps. The 'cluster' module is more suitable for parallelizing CPU-bound tasks, not sequential data processing.

What is the command to build a Docker image for a Node.js application?

Option 1:
docker build -t my-node-app .

Option 2:
docker create -t my-node-app .

Option 3:
docker run -t my-node-app .

Option 4:
docker image create my-node-app .

Correct Response:
1.0

Explanation:
The correct command to build a Docker image for a Node.js application is docker build -t my-node-app .. This command uses the Dockerfile in the current directory (.) to build an image with the specified tag (-t my-node-app).

Which file is typically used to define a Node.js application's Docker container configuration?

Option 1:
docker-config.json

Option 2:
node-docker.yml

Option 3:
Dockerfile

Option 4:
package.json

Correct Response:
3.0

Explanation:
The Dockerfile is commonly used to define a Node.js application's Docker container configuration. It contains instructions for building the Docker image, such as specifying the base image, setting up the working directory, and copying necessary files.

Name a popular service that can be used to implement CI/CD for Node.js applications.

Option 1:
Jenkins

Option 2:
Travis CI

Option 3:
CircleCI

Option 4:
Docker Hub

Correct Response:
3.0

Explanation:
CircleCI is a popular CI/CD service that can be used to implement continuous integration and continuous deployment for Node.js applications. It automates the build, test, and deployment processes, providing a streamlined workflow.

How do you specify the Node.js version you want to use in your Dockerfile?

Option 1:
FROM node:10

Option 2:
ARG NODE_VERSION=12

Option 3:
ENV NODE_VERSION=14

Option 4:
RUN apt-get install -y nodejs

Correct Response:
3.0

Explanation:
In a Dockerfile, you can specify the Node.js version using the FROM instruction followed by the desired version tag. This sets the base image for the Docker container. An alternative approach is to use the ENV instruction to set the Node.js version. The example provided shows setting the Node.js version to 14 using the ENV instruction.

What is a Docker Compose file and how is it used in the context of Node.js applications?

Option 1:
A Docker Compose file is used to define multi-container Docker applications. It specifies services, networks, and volumes.

Option 2:
It is a configuration file used to define the structure of a Node.js application.

Option 3:
It is a script for building Docker images for Node.js applications.

Option 4:
It is a file used to configure Node.js modules for Docker deployment.

Correct Response:
1.0

Explanation:
A Docker Compose file is utilized to define the services, networks, and volumes for a multi-container Docker application. It allows you to manage the configuration in a single file, facilitating the orchestration of multiple containers for a Node.js application.

Describe a strategy to manage environment-specific configurations in CI/CD pipelines for Node.js.

Option 1:
Use environment variables to customize configurations based on the target environment.

Option 2:
Embed configuration files directly into the source code.

Option 3:
Maintain separate branches for each environment configuration.

Option 4:
Store configurations in a centralized database and fetch them at runtime.

Correct Response:
1.0

Explanation:
In CI/CD pipelines for Node.js, managing environment-specific configurations involves using strategies like leveraging environment variables. This allows you to customize configurations dynamically based on the target environment, ensuring consistency across different stages of the pipeline.

Explain how Docker's layered filesystem benefits Node.js application deployment and scaling.

Option 1:
Efficient use of layers to cache dependencies and improve build

Option 2:
Reduction of image size using multi-stage builds

Option 3:
Utilizing Docker Compose for layer management

Option 4:
Employing Kubernetes for filesystem optimization

Correct Response:
2.0

Explanation:
Docker uses a layered filesystem, enabling the reuse of intermediate layers, reducing image size. Multi-stage builds help discard unnecessary dependencies, improving efficiency. Docker Compose is a tool for defining and managing multi-container Docker applications. Kubernetes is an orchestration platform, not directly related to Docker's layered filesystem.

What are some best practices for minimizing the size of a Node.js Docker image?

Option 1:
Removing unnecessary dependencies

Option 2:
Utilizing a slim base image

Option 3:
Combining multiple layers into a single layer

Option 4:
Minimizing the use of environment variables

Correct Response:
2.0

Explanation:
Minimizing a Node.js Docker image involves removing unnecessary dependencies, using a slim base image, and combining multiple layers. While environment variables are important, they don't directly impact image size in the context of minimizing it.

Discuss how you would integrate end-to-end tests into a CI/CD pipeline for a Node.js application.

Option 1:
Running tests in parallel for faster feedback

Option 2:
Utilizing a separate pipeline for tests

Option 3:
Incorporating test coverage analysis

Option 4:
Integrating tests at multiple stages of the pipeline

Correct Response:
1.0

Explanation:
Integrating end-to-end tests in a CI/CD pipeline involves running tests in parallel for quicker feedback, ensuring faster delivery. A separate pipeline for tests can be counterproductive. Coverage analysis is valuable but not the primary focus of integration. Integrating tests at various stages ensures comprehensive validation.

The command docker run -p 80:3000 node_app maps port 3000 in the container to port _____ on the Docker host.

Option 1:
80

Option 2:
3000

Option 3:
8080

Option 4:
4000

Correct Response:
1.0

Explanation:
The correct option is 80. This command binds port 80 on the Docker host to port 3000 in the container. The syntax is -p hostPort:containerPort, so in this case, it exposes port 3000 in the container to port 80 on the Docker host. This is useful for accessing

the Node.js application running in the Docker container from port 80 on the host machine.

In a Node.js project, a _____ file is used to specify the commands that should be executed automatically on a build server when the repository is updated.

Option 1:
.travis.yml

Option 2:
package.json

Option 3:
Dockerfile

Option 4:
.gitignore

Correct Response:
1.0

Explanation:
The correct option is .travis.yml. This file is used for configuring Travis CI, a continuous integration service. It contains instructions for running tests and other tasks when changes are pushed to the repository. In the context of a Node.js project, it ensures that the specified commands are executed automatically on a build server, promoting continuous integration.

To handle secret keys and credentials in a Node.js application Docker container, it is recommended to use Docker _____.

Option 1:
Secrets

Option 2:
Volumes

Option 3:
Compose

Option 4:
Env Variables

Correct Response:
4.0

Explanation:
The correct option is Env Variables. Using environment variables in Docker helps manage sensitive information like secret keys and credentials. By passing these values as environment variables, you can keep them separate from the codebase and easily update them

when needed. It enhances security and flexibility in handling confidential information within a Node.js application.

When setting up a Node.js application for continuous deployment, the use of _____ in Docker can help you run database migrations during the deployment process.

Option 1:
Multi-Stage Builds

Option 2:
Docker Compose

Option 3:
Docker Swarm

Option 4:
Docker Volumes

Correct Response:
1.0

Explanation:
In a continuous deployment setup, leveraging Multi-Stage Builds in Docker allows for a clean separation of build and runtime environments. This helps in running database migrations during the deployment process. Multi-Stage Builds enable the creation of lightweight and efficient Docker images by building and copying only the necessary artifacts for runtime.

To optimize the build time of a Node.js Docker image in a CI/CD pipeline, leveraging Docker's _____ cache is essential.

Option 1:
Layer

Option 2:
Container

Option 3:
Volume

Option 4:
Snapshot

Correct Response:
1.0

Explanation:
Docker uses layers to store intermediate build artifacts. Leveraging Docker's layer caching mechanism is crucial for optimizing build times in CI/CD pipelines. Layers are cached during successive builds,

and if there are no changes to a particular layer, Docker can reuse the cached layer, significantly reducing build times.

In a CI/CD pipeline, a _____ can be used to automatically revert a Node.js deployment in Docker if health checks fail after a new image is deployed.

Option 1:
Canary Deployment

Option 2:
Rollback Mechanism

Option 3:
Blue-Green Deployment

Option 4:
Feature Toggle

Correct Response:
2.0

Explanation:
Implementing a Rollback Mechanism in a CI/CD pipeline allows for automatic reversion of a Node.js deployment in Docker if health checks fail after deploying a new image. This ensures that in case of issues, the system can quickly revert to the previous, stable state, maintaining reliability and minimizing downtime.

You have a Node.js application containerized with Docker. During deployment, the application fails to start, and you suspect it's a port conflict issue. How would you investigate and resolve this in the Docker setup?

Option 1:
Check for conflicting ports on the host machine, inspect Docker container logs for error messages, update Docker Compose file if needed, restart Docker daemon.

Option 2:
Review Node.js application code for port configurations, modify Dockerfile to expose the correct port, ensure the host machine's

firewall allows the specified port, update CI/CD pipeline for port consistency.

Option 3:
Examine Docker container's resource allocation, review network settings, inspect system-wide port usage, check for Docker daemon restart issues.

Option 4:
Rebuild the Docker image with a different base image, modify the application's entry point script, adjust Docker Compose service name, update the CI/CD pipeline for port resolution.

Correct Response:
1.0

Explanation:
When troubleshooting port conflicts in a Node.js Docker container, it's crucial to inspect both the host machine and container configurations. Checking logs, updating necessary files, and ensuring consistency across the pipeline are essential steps.

A Node.js Docker container works well locally but fails when deployed through the CI/CD pipeline. What steps would you take to diagnose the issue, considering the container's environment consistency?

Option 1:
Review CI/CD pipeline configuration for environment variables, check for dependency mismatches between local and production environments, inspect the container's environment variables during CI/CD execution, ensure consistent Node.js versions.

Option 2:
Update Docker Compose file to match the production environment, review CI/CD pipeline logs for error messages, check for missing dependencies in the production environment, verify that required environment variables are correctly set.

Option 3:
Review Node.js application code for environment-specific configurations, use Docker Inspect to examine container details, update CI/CD pipeline with additional environment checks, rebuild Docker image with production-ready configurations.

Option 4:
Debug the CI/CD pipeline script for inconsistencies, use Docker Compose override files for different environments, review Node.js application logs in both local and production environments, ensure that the production database is reachable.

Correct Response:
2.0

Explanation:
Diagnosing discrepancies between local and production environments involves checking configurations at various levels, including CI/CD settings, Docker Compose files, and Node.js application code.

Describe how you would set up a CI/CD pipeline to support blue-green deployments for a Node.js application using Docker.

Option 1:
Configure two separate environments (blue and green) in the CI/CD pipeline, use Docker Compose override files for each environment, implement a load balancer to switch between blue and green deployments, update database migrations for each version.

Option 2:
Utilize Docker Compose for defining blue and green environments, incorporate a reverse proxy for load balancing, automate database schema migrations with each deployment, integrate rollback mechanisms in the CI/CD pipeline.

Option 3:
Implement a custom script for managing blue-green deployments, use environment variables to toggle between deployments, ensure zero-downtime database migrations, incorporate health checks for automatic rollback.

Option 4:
Set up separate Docker Compose files for blue and green

environments, use environment-specific configuration files, automate the switching of Docker Compose files during deployments, update DNS settings for smooth transitions.

Correct Response:
2.0

Explanation:
Blue-green deployments with Docker involve careful orchestration of environments, load balancing, and database migrations. CI/CD pipelines play a key role in automating and ensuring the success of such deployment strategies.

What is the primary purpose of logging in a Node.js application?

Option 1:
Debugging

Option 2:
Monitoring

Option 3:
Authentication

Option 4:
Performance Improvement

Correct Response:
2.0

Explanation:
Logging in a Node.js application is primarily used for monitoring the application's behavior, identifying errors, and gaining insights into its performance. It helps developers understand how the application is functioning in real-time.

Which Node.js module is commonly used to log requests and errors in an application?

Option 1:
fs

Option 2:
http

Option 3:
path

Option 4:
log4js

Correct Response:
4.0

Explanation:
The 'log4js' module is commonly used for logging requests and errors in a Node.js application. It provides a flexible and configurable logging framework that supports various output formats and log levels.

What is load balancing in the context of web application deployment?

Option 1:
Distributing network traffic across multiple servers

Option 2:
Minimizing the size of application code

Option 3:
Optimizing database queries

Option 4:
Managing client-side resources

Correct Response:
1.0

Explanation:
Load balancing involves distributing network traffic across multiple servers to ensure no single server is overwhelmed. This improves the application's reliability, availability, and scalability by preventing any single server from becoming a bottleneck.

How can you enable automatic logging of HTTP requests in a Node.js application using a middleware?

Option 1:
Use morgan middleware

Option 2:
Implement custom logging middleware

Option 3:
Utilize built-in console module

Option 4:
Use express-logger middleware

Correct Response:
1.0

Explanation:
In Node.js, morgan is a popular middleware that provides automatic logging of HTTP requests. It can be easily integrated into an Express.js application to log details like request method, status code,

and response time. This helps in monitoring and debugging the application's HTTP traffic.

Which tool can be used for real-time monitoring of a Node.js application's performance?

Option 1:
New Relic

Option 2:
Nodemon

Option 3:
PM2

Option 4:
JSHint

Correct Response:
1.0

Explanation:
New Relic is a performance monitoring tool that allows real-time monitoring of Node.js applications. It provides insights into various aspects such as response time, throughput, and error rates, helping developers identify and resolve performance issues.

Describe how the Cluster module in Node.js can be used to implement load balancing across multiple CPU cores.

Option 1:
Fork child processes with cluster.fork()

Option 2:
Utilize a reverse proxy like Nginx

Option 3:
Use a single process to handle all requests

Option 4:
Implement load balancing through DNS

Correct Response:
1.0

Explanation:
The Cluster module in Node.js allows developers to take advantage of multiple CPU cores by creating child processes with cluster.fork(). Each child process can handle incoming requests, distributing the

load and improving application performance. It simplifies the implementation of load balancing in a multi-core environment.

In the context of centralized logging for multiple Node.js services, what is the advantage of using a log aggregation tool?

Option 1:
Improved Debugging

Option 2:
Scalability

Option 3:
Real-time Analysis

Option 4:
Streamlined Maintenance

Correct Response:
2.0

Explanation:
Log aggregation tools, such as Elasticsearch and Logstash (ELK stack), offer scalability by consolidating logs from various services, aiding in real-time analysis for rapid issue identification. This leads to improved debugging and streamlined maintenance processes.

What are the key metrics to monitor in a Node.js application to ensure its health and performance?

Option 1:
Event Loop Latency

Option 2:
Memory Usage

Option 3:
CPU Utilization

Option 4:
Network Throughput

Correct Response:
4.0

Explanation:
Monitoring network throughput, CPU utilization, memory usage, and event loop latency are crucial to assess a Node.js application's health and performance. High throughput, optimal CPU usage, and efficient memory management contribute to a robust and responsive application.

How does a reverse proxy server facilitate load balancing for Node.js applications?

Option 1:
Round Robin Scheduling

Option 2:
IP Hashing

Option 3:
Least Connections

Option 4:
Server Weight-based Balancing

Correct Response:
1.0

Explanation:
A reverse proxy server, through methods like round-robin scheduling, IP hashing, or least connections, facilitates load balancing for Node.js applications by distributing incoming requests among multiple servers. This improves overall application performance, prevents overloading, and enhances fault tolerance by efficiently utilizing server resources.

To track down application errors in production, it's essential to have a comprehensive _____ strategy in place.

Option 1:
Logging

Option 2:
Debugging

Option 3:
Monitoring

Option 4:
Profiling

Correct Response:
3.0

Explanation:
In a production environment, monitoring strategies help identify issues, bottlenecks, and errors in real-time, allowing for quick response and resolution. Monitoring provides a holistic view of the application's performance and health.

Node.js applications can be monitored using _____, which provides insights into runtime performance and resource usage.

Option 1:
New Relic

Option 2:
PM2

Option 3:
Winston

Option 4:
Dynatrace

Correct Response:
2.0

Explanation:
PM2 is a popular process manager for Node.js applications that also

offers monitoring capabilities. It provides real-time insights into the application's performance, resource usage, and other relevant metrics.

A _____ can distribute incoming network traffic across several servers to optimize resource use, reduce response times, and avoid overload of any single resource.

Option 1:
Reverse Proxy

Option 2:
Load Balancer

Option 3:
Gateway

Option 4:
Firewall

Correct Response:
2.0

Explanation:
A load balancer distributes incoming network traffic across multiple servers, ensuring even resource utilization, improved response times, and high availability. It helps prevent overloading a single server and enhances the application's scalability.

When setting up a centralized logging system, _____ can be used to transport logs from various services to a central server.

Option 1:
Logstash

Option 2:
Fluentd

Option 3:
Winston

Option 4:
Bunyan

Correct Response:
2.0

Explanation:
In the context of Node.js, Fluentd is a robust and versatile open-source log collector that can efficiently transport logs from different services to a central server. It provides flexibility in log forwarding and is often used in conjunction with other tools.

_____ is a Node.js module that allows you to spin up multiple worker processes to handle incoming network traffic efficiently.

Option 1:
Cluster

Option 2:
Express

Option 3:
PM2

Option 4:
Axios

Correct Response:
1.0

Explanation:
Node.js Cluster module allows the creation of multiple worker processes, enabling the efficient handling of network traffic. It's particularly useful for scaling applications and taking advantage of multi-core systems to improve performance.

To minimize downtime and provide seamless updates, _____ enables traffic to be routed away from servers under maintenance.

Option 1:
Nginx

Option 2:
Docker

Option 3:
Kubernetes

Option 4:
Apache

Correct Response:
3.0

Explanation:
Kubernetes is a container orchestration platform that facilitates seamless updates and minimizes downtime by intelligently routing traffic away from servers undergoing maintenance. Its robust

features make it a popular choice for deploying and managing containerized applications.

A Node.js application is experiencing intermittent slowdowns. How would you approach setting up monitoring to identify potential bottlenecks?

Option 1:
Implementing performance profiling tools

Option 2:
Analyzing server logs

Option 3:
Utilizing APM (Application Performance Monitoring) tools

Option 4:
Setting up custom metrics using Node.js APIs

Correct Response:
3.0

Explanation:
APM tools provide real-time insights into application performance,

allowing identification of bottlenecks. Analyzing logs may not be as immediate, and custom metrics can be more specific than general profiling tools.

You are tasked with designing a load balancing solution for a high-traffic Node.js web application. What factors would you consider when choosing between different load balancing algorithms?

Option 1:
Session persistence requirements

Option 2:
Server health monitoring

Option 3:
Latency-based routing

Option 4:
Round Robin scheduling

Correct Response:
2.0

Explanation:
Server health monitoring ensures even distribution, latency-based routing optimizes based on response times, and round robin ensures fairness. Session persistence depends on application needs.

If a Node.js application's logging system is generating too much data and causing I/O bottlenecks, what strategies could be implemented to handle this?

Option 1:
Implement log rotation

Option 2:
Use a centralized logging system

Option 3:
Apply log level filtering

Option 4:
Utilize asynchronous logging

Correct Response:
2.0

Explanation:
Centralized logging reduces I/O on individual servers. Log rotation

helps manage file size. Filtering based on log levels and asynchronous logging improve performance.

Which of the following is a common use of environment variables in a Node.js application?

Option 1:
Configuring application settings

Option 2:
Declaring constants

Option 3:
Defining function parameters

Option 4:
Storing data in a database

Correct Response:
1.0

Explanation:
Environment variables are commonly used for configuring application settings such as database connection strings, API keys, and other configurations that may vary between development, testing, and production environments.

What file is typically used to define scripts and dependencies for a Node.js project?

Option 1:
package.json

Option 2:
index.js

Option 3:
app.js

Option 4:
node_modules

Correct Response:
1.0

Explanation:
The package.json file is used to define scripts, dependencies, and other metadata about a Node.js project. It provides a central location to manage project configuration and dependencies, making it a key file in Node.js development.

How can environment variables be accessed within a Node.js application?

Option 1:
process.env

Option 2:
window.env

Option 3:
global.env

Option 4:
app.config

Correct Response:
1.0

Explanation:
Environment variables can be accessed in Node.js using process.env. This object contains key-value pairs of the user environment and is commonly used to retrieve configuration values set outside the application code.

What tool can be used to manage environment configurations for different deployment environments in Node.js?

Option 1:
dotenv

Option 2:
config

Option 3:
nconf

Option 4:
yenv

Correct Response:
2.0

Explanation:
In Node.js, developers commonly use tools like 'dotenv,' 'config,' 'nconf,' or 'yenv' to manage environment configurations. These tools help in handling different settings for various deployment

environments, making it easier to switch configurations without modifying the code.

In the context of Node.js, what is the benefit of using a process manager like PM2?

Option 1:
Improved Fault Tolerance

Option 2:
Load Balancing

Option 3:
Hot Reloading

Option 4:
Memory Management

Correct Response:
1.0

Explanation:
Using a process manager like PM2 in Node.js provides improved fault tolerance. PM2 can automatically restart crashed processes, enhancing the overall reliability of the application. This is crucial for ensuring continuous availability and minimizing downtime.

What is the primary purpose of using the cluster module in Node.js?

Option 1:
Load Balancing

Option 2:
Concurrency

Option 3:
Code Organization

Option 4:
Memory Management

Correct Response:
2.0

Explanation:
The primary purpose of using the cluster module in Node.js is to achieve better concurrency by creating child processes that share the same server port. This enables the application to handle multiple requests concurrently, enhancing performance and utilizing hardware resources efficiently.

How do environment variables contribute to the Twelve-Factor App methodology in Node.js application deployments?

Option 1:
They provide a way to store sensitive information securely, such as API keys and database credentials.

Option 2:
They enable configuration flexibility, allowing developers to adjust application behavior without code changes.

Option 3:
They enhance portability by making it easy to move applications between environments.

Option 4:
They support collaboration by separating configuration from code, making it easier to share codebases.

Correct Response:
2.0

Explanation:
Environment variables in Node.js applications are crucial for maintaining the principles of the Twelve-Factor App methodology. They allow developers to separate configuration from code, providing flexibility and security. This separation enhances collaboration and portability, supporting the best practices outlined in Twelve-Factor App.

Describe the impact of Node.js application scaling on database connections and how connection pooling can be beneficial.

Option 1:
Increased Node.js application scaling can lead to a higher demand for database connections, potentially causing performance bottlenecks.

Option 2:
Connection pooling helps address this challenge by efficiently managing and reusing database connections, reducing the overhead of establishing new connections.

Option 3:
Connection pooling ensures that the database server is not overwhelmed with a large number of simultaneous connections.

Option 4:
Connection pooling improves fault tolerance by automatically handling connection failures and retries.

Correct Response:
2.0

Explanation:
Scaling Node.js applications can strain database connections. Connection pooling mitigates this by efficiently managing connections, reducing overhead. It improves performance and fault tolerance, making it a key strategy in handling increased demand during application scaling.

Explain the role of a reverse proxy when scaling Node.js applications across multiple servers or instances.

Option 1:
A reverse proxy serves as an intermediary between client requests and multiple Node.js servers, distributing incoming traffic.

Option 2:
It helps with load balancing by distributing requests evenly among the backend servers, preventing individual servers from becoming overwhelmed.

Option 3:
A reverse proxy provides an additional layer of security by hiding the internal structure of the application servers from external clients.

Option 4:
It enables features like SSL termination, centralizing SSL certificate management for all backend servers.

Correct Response:
1.0

Explanation:
In a scaled Node.js application, a reverse proxy plays a crucial role by distributing traffic, ensuring load balancing, enhancing security, and

centralizing SSL management. It acts as an intermediary, improving performance and simplifying SSL certificate handling across multiple servers.

Node.js applications are scaled horizontally by increasing the number of _____ running the application.

Option 1:
instances

Option 2:
threads

Option 3:
modules

Option 4:
processes

Correct Response:
4.0

Explanation:
In Node.js, horizontal scaling involves running multiple processes to handle requests concurrently. Therefore, to scale a Node.js application horizontally, developers increase the number of

processes. This approach allows the application to take advantage of multi-core systems and distribute the load effectively.

The command process.env.NODE_ENV in Node.js is typically used to determine the _____ environment of the application.

Option 1:
execution

Option 2:
development

Option 3:
server

Option 4:
runtime

Correct Response:
2.0

Explanation:
The process.env.NODE_ENV command is commonly used to

identify the development or production environment in which the Node.js application is running. Developers can use this information to configure settings, such as enabling debugging features during development and optimizing performance in production.

To scale a Node.js application, developers can use the _____ module to fork multiple worker processes.

Option 1:
cluster

Option 2:
fork

Option 3:
child

Option 4:
parallel

Correct Response:
1.0

Explanation:
The cluster module in Node.js allows developers to create a cluster of worker processes. By forking multiple worker processes, each process can handle a portion of the incoming requests. This

approach enhances the application's scalability and takes advantage of multi-core systems for improved performance.

For complex configuration management in Node.js, the _____ package allows you to store configuration in hierarchical files.

Option 1:
dotenv

Option 2:
config

Option 3:
yargs

Option 4:
nconf

Correct Response:
4.0

Explanation:
The correct option is nconf. nconf is a powerful configuration management package in Node.js that enables storing configuration data in hierarchical files, making it suitable for complex setups where different configuration levels are needed.

When scaling Node.js applications, it's important to consider the _____ pattern to manage session state across multiple instances.

Option 1:
Singleton

Option 2:
Observer

Option 3:
Stateless

Option 4:
sticky-session

Correct Response:
4.0

Explanation:
The correct option is sticky-session. In a scaled Node.js application,

the sticky-session pattern is used to ensure that requests from the same client are always directed to the same server instance, helping to manage session state effectively.

A Node.js application's throughput can be improved by offloading tasks to worker processes using the _____ module.

Option 1:
cluster

Option 2:
child_process

Option 3:
async

Option 4:
pm2

Correct Response:
1.0

Explanation:
The correct option is cluster. The cluster module in Node.js allows for the creation of worker processes, distributing the load and improving throughput by taking advantage of multiple CPU cores.

You are configuring a Node.js application for different environments (development, staging, production). What strategy would you use to manage the environment-specific variables securely?

Option 1:
Use environment variables with a configuration module

Option 2:
Embed environment variables directly in the code

Option 3:
Store environment-specific data in a configuration file

Option 4:
Use a version control system for environment configurations

Correct Response:
1.0

Explanation:
Environment variables are a secure way to manage configuration settings in Node.js applications. Storing them in a configuration module helps maintain consistency and avoids exposing sensitive data.

A Node.js application is experiencing increased traffic and the single instance is not able to handle the load. What scaling strategies would you consider implementing?

Option 1:
Horizontal scaling by adding more instances

Option 2:
Vertical scaling by upgrading the server's hardware

Option 3:
Load balancing across multiple servers

Option 4:
Caching frequently accessed data

Correct Response:
1.0

Explanation:
Horizontal scaling involves adding more instances to distribute the load, making it a suitable strategy for handling increased traffic. Load balancing ensures even distribution, improving overall performance.

After scaling out your Node.js application to multiple instances, you notice session-related issues. What could be the cause and how might you resolve it?

Option 1:
Sticky sessions to tie a user's session to a specific instance

Option 2:
Use a centralized session store (e.g., Redis)

Option 3:
Implement JWT (JSON Web Tokens) for session management

Option 4:
Increase the session timeout duration

Correct Response:
2.0

Explanation:
Scaling to multiple instances can lead to session-related issues.

Using a centralized session store like Redis ensures that sessions are shared and accessible across all instances, resolving potential conflicts.

Which built-in Node.js module can be used to start a simple profiling session to monitor performance?

Option 1:
fs

Option 2:
path

Option 3:
util

Option 4:
v8

Correct Response:
4.0

Explanation:
In Node.js, the v8 module provides the v8.profiler object, which can be used to start a profiling session for monitoring performance. This allows developers to identify and optimize areas of their code that may be causing performance bottlenecks.

Name one popular Node.js command-line tool that helps in profiling and understanding event loop latency.

Option 1:
npm

Option 2:
nodemon

Option 3:
clinic

Option 4:
mocha

Correct Response:
3.0

Explanation:
The clinic is a popular Node.js command-line tool that helps in profiling and understanding event loop latency. It provides insights into the performance of your application and aids in identifying and resolving bottlenecks.

What is a common practice when preparing a Node.js application for deployment to a cloud platform like AWS or Azure?

Option 1:
Minify all JavaScript files

Option 2:
Use a process manager like PM2

Option 3:
Embed API keys directly in the code

Option 4:
Hardcode configuration values

Correct Response:
2.0

Explanation:
When preparing a Node.js application for deployment to a cloud platform, using a process manager like PM2 is a common practice. PM2 ensures that your Node.js application runs continuously, can be easily managed, and is scalable in a cloud environment.

Which built-in Node.js module can be used to start a simple profiling session to monitor performance?

Option 1:
debug

Option 2:
profiler

Option 3:
v8

Option 4:
performance

Correct Response:
3.0

Explanation:
The correct option is c) v8. The v8 module in Node.js provides the v8-profiler module, which allows you to start a profiling session and analyze the performance of your application. This is useful for identifying bottlenecks and optimizing code.

Name one popular Node.js command-line tool that helps in profiling and understanding event loop latency.

Option 1:
trace

Option 2:
nodemon

Option 3:
clinic

Option 4:
debugger

Correct Response:
3.0

Explanation:
The correct option is c) clinic. The clinic command-line tool is commonly used for profiling and diagnosing performance issues in Node.js applications. It provides insights into event loop latency, CPU usage, and other metrics.

What is a common practice when preparing a Node.js application for deployment to a cloud platform like AWS or Azure?

Option 1:
Bundle dependencies with the application

Option 2:
Include unnecessary files for reference

Option 3:
Use a monolithic structure

Option 4:
Set environment-specific configurations

Correct Response:
1.0

Explanation:
The correct option is a) Bundle dependencies with the application. It is a good practice to package your Node.js application with its dependencies to ensure consistency and avoid potential issues with

missing modules during deployment. This is often achieved using tools like npm pack or Docker containers.

The Node.js _____ tool can be used to monitor and analyze the runtime performance of an application.

Option 1:
Profiling

Option 2:
Debugging

Option 3:
Tracing

Option 4:
Testing

Correct Response:
1.0

Explanation:
In Node.js, profiling tools like V8 profiler can be used to gather information about the runtime performance of an application, helping identify performance bottlenecks.

In Node.js, the _____ command is used to generate a CPU profile in a cloud deployment environment.

Option 1:
npm profile

Option 2:
node --prof

Option 3:
deploy --cpu

Option 4:
cloud-cpu-profile

Correct Response:
2.0

Explanation:
The node --prof command in Node.js is used to generate a CPU profile, which can be crucial for understanding and optimizing the performance of a Node.js application in a cloud environment.

Deploying a Node.js application to Azure commonly involves the use of _____ Services.

Option 1:
Azure App

Option 2:
Azure Cloud

Option 3:
Azure Web

Option 4:
Azure Platform

Correct Response:
3.0

Explanation:
When deploying a Node.js application to Azure, Azure App Services are commonly used to host and manage the application in a scalable and efficient manner.

Advanced Node.js profiling may involve analyzing _____, which can indicate memory leaks when they grow unexpectedly over time.

Option 1:
Event loop

Option 2:
Garbage collection

Option 3:
Heap snapshots

Option 4:
Asynchronous functions

Correct Response:
3.0

Explanation:
In Node.js, advanced profiling often involves analyzing Heap

snapshots. If the heap size grows unexpectedly over time, it may indicate memory leaks. Memory leaks can lead to performance issues and application crashes. Understanding heap usage is crucial for optimizing Node.js applications.

The _____ feature in cloud platforms can help in rolling back Node.js applications to a previous state in case of deployment failure.

Option 1:
Auto Scaling

Option 2:
Blue-Green Deployment

Option 3:
Load Balancing

Option 4:
Serverless Computing

Correct Response:
2.0

Explanation:
Blue-Green Deployment is a technique used in cloud platforms to

deploy applications. It involves running two separate environments, allowing for easy rollback in case of deployment failure. This helps maintain application availability and reliability during the deployment process.

Node.js applications that require scaling due to high traffic are often deployed using _____ in AWS to manage the distribution of incoming traffic.

Option 1:
Amazon RDS

Option 2:
Amazon S3

Option 3:
Amazon EC2

Option 4:
Amazon Elastic Load Balancing (ELB)

Correct Response:
4.0

Explanation:
Amazon Elastic Load Balancing (ELB) is a service in AWS that

automatically distributes incoming application traffic across multiple targets, such as EC2 instances. It ensures high availability and fault tolerance for Node.js applications experiencing high traffic.

A Node.js application is experiencing memory leaks in production. You need to capture and analyze heap dumps without stopping the application. What tools or techniques would you use?

Option 1:
Heapdump

Option 2:
Node.js Inspector

Option 3:
PM2

Option 4:
V8 Profiler

Correct Response:
1.0

Explanation:
Heapdump is a popular tool for capturing and analyzing heap dumps in a Node.js application without stopping it. It provides valuable insights into memory usage and helps identify and troubleshoot memory leaks. Node.js Inspector, PM2, and V8 Profiler are not primarily used for heap dump analysis.

You are tasked with automating the deployment of a Node.js application across multiple cloud environments for testing, staging, and production. What strategy or tool would you recommend?

Option 1:
Ansible

Option 2:
Jenkins

Option 3:
Docker Compose

Option 4:
Kubernetes

Correct Response:
4.0

Explanation:
Kubernetes is a powerful container orchestration tool suitable for automating the deployment of Node.js applications across multiple cloud environments. While Ansible, Jenkins, and Docker Compose are useful for automation, Kubernetes excels in managing containerized applications in a scalable and efficient manner.

After deploying a Node.js application on Azure, you need to ensure that it can scale automatically based on CPU usage. Which Azure service will you configure to achieve this?

Option 1:
Azure App Service

Option 2:
Azure Functions

Option 3:
Azure Kubernetes Service (AKS)

Option 4:
Azure Virtual Machines

Correct Response:
3.0

Explanation:
Azure Kubernetes Service (AKS) allows automatic scaling based on CPU usage for Node.js applications. Azure App Service is a platform-as-a-service (PaaS) offering suitable for web apps but may not provide fine-grained control over scaling. Azure Functions are more suitable for event-driven workloads. Azure Virtual Machines require manual scaling.