100 Must-Know Node.js Interview Questions in 2025

1. What is Node.js and why is it used?

Node.js is an open-source, cross-platform JavaScript runtime environment that executes code outside of a web browser. It is built on V8, the same JavaScript engine within Chrome, and optimized for high performance. This environment, coupled with an event-driven, non-blocking I/O framework, is tailored for server-side web development and more.

Key Features

- **Asynchronous & Non-Blocking**: Ideal for handling a myriad of concurrent connections with efficiency.
- **V8 Engine**: Powered by Google's V8, Node.js boasts top-tier JavaScript execution.
- **Libuv Library**: Ensures consistent performance across platforms and assists in managing I/O operations.
- **NPM**: A vast package ecosystem simplifies module management and deployment.
- **Full-Stack JavaScript**: Allows for unified server and client-side code in JavaScript.

Use Cases

- **Data Streaming**: Suited for real-time streaming of audio, video, and lightweight data.
- **API Servers**: Ideal for building fast, scalable, and data-intensive applications.
- **Microservices**: Its module-oriented design facilitates the development of decoupled, independently scalable services.
- **Single Page Applications**: Often used with frameworks like Angular, React, or Vue to craft robust, server-side backends.
- **Chat Applications**: Its real-time capabilities are advantageous in building instant messaging systems.
- **Internet of Things (IoT)**: Provides a lightweight environment for running applications on constrained devices like Raspberry Pi.

Why Node.js?

- **Unified Language**: Utilizing JavaScript both on the frontend and backend brings coherence to development efforts, potentially reducing debugging time and enabling shared libraries.
- **NPM Ecosystem**: The NPM repository offers myriad open-source packages, empowering rapid development and feature expansion.
- **Rapid Prototyping**: Express, a minimalist web framework for Node.js, and NPM's wealth of modules expedite early application development and testing.
- **Scalability**: Cluster modules, load balancers, and Microservice Architecture aid in linear, on-demand scaling for both simple and intricate applications.
- **Real-Time Power**: With built-in WebSockets and event-based architecture, Node.js excels in constructing real-time applications such as multiplayer games, stock trading platforms, and chat applications.
- **Open Source**: Being an open-source technology, Node.js continuously benefits from community contributions, updates, and enhanced packages.

2. How does Node.js handle child threads?

Node.js employs event-driven architecture and non-blocking I/O for efficiency.

While Node.js **operates off a single main thread**, it can harness the full power of multi-core systems by launching child threads for specific tasks, such as file compression or image processing.

Thread Pool and Worker Threads

To manage these child threads, Node.js uses a combination of:

- A **thread pool**, powered by the libuv library.
- **Worker threads** for dedicated, offloaded computation.

Node.js Event Loop

When a task in Node.js is designated to operate on a child thread, the main event loop hands it over to the thread pool. This setup allows Node.js to **stay responsive to incoming requests**, benefiting from asynchronous I/O.

The main event loop regains control once the task on the child thread is completed, and results are ready.

Advantages

- **Boosted Efficiency**: Offloading certain tasks to worker threads prevents I/O or computation-heavy jobs from blocking the event loop.
- **Convenient Multi-Threading**: Node.js enables multi-threading without the complexities of managing threads directly.

Code Example: Basic Multi-Threading Task

Here is the JavaScript code:

```
// Import the built-in 'worker_threads' module
const { Worker, isMainThread, parentPort } = require('worker_threads');

// Check if it's the main module
if (isMainThread) {
  // Create a new child worker
  const worker = new Worker(__filename);

  // Listen for messages from the worker
  worker.on('message', message => console.log('Received:', message));

  // Send a message to the worker
  worker.postMessage('Hello from the main thread!');
} else {
  // Listen for messages from the main thread
  parentPort.on('message', message => {
    console.log('Received in the worker:', message);
    // Send a message back to the main thread
    parentPort.postMessage('Hello from the worker thread!');
  });
}
```

3. Describe the event-driven programming in Node.js.

Event-driven programming, a hallmark of Node.js, uses an **event, listener,** and **emitter** architecture to handle asynchronous tasks. This design centers around events and how they trigger actions in the attached listeners.

Core Components

- **Event Emitter**: Acts as the event registry and dispatcher, letting objects register interest in particular events and emit these events when they occur.
- **Event Handler (Listener)**: Associates with a particular event through registration. These callback functions will be asynchronously carried out when a matching event is emitted.

Code Example: Event Emitter and Handlers

Here is the Node.js code:

```
const { EventEmitter } = require('events');
const emitter = new EventEmitter();

emitter.on('event-name', (eventArgs) => {
    console.log(`Event-name was emitted with arguments: ${eventArgs}`);
});

emitter.emit('event-name', 'Some Payload');
```

In this code, when `emit` is called, the `on` method's callback is executed asynchronously.

Event Loop Mechanism in Node.js

- **Call Stack**: Maintains the call order of the functions and methods being executed.
- **Node APIs** and **Callbacks Queue**: Handle I/O tasks and timers.
- **Event Loop**: Constantly watches the execution stack and checks whether it's clear to execute pending tasks from the Callback Queue.

Practical Applications in Node.js

- **HTTP Server**: Listens for and serves requests.
- **File System Operations**: Execute I/O tasks.
- **Database Operations**: Such as data retrieval.

4. What is the event loop in Node.js?

The **event loop** is a fundamental concept in Node.js for managing asynchronous operations. Its efficiency is a key reason behind Node.js's high performance.

How Does the Event Loop Work?

1. **Initialization**: When Node.js starts, it initializes the **event loop** to watch for I/O operations and other asynchronous tasks.
2. **Queueing**: Any task or I/O operation is added to a **queue**, which can be either the `microtask queue` or the `macrotask/Callback queue`.
3. **Polling**: The event loop iteratively checks for tasks in the queue while also **waiting** for I/O and timers.
4. **Execution Phases**: When the event loop detects tasks in the queue, it executes them in specific phases, ensuring order efficiency.

Task Scheduler Zones: microtask and Callback Queue

- **Microtask Queue**: This is a highly prioritized queue, usually acting over tasks in the **Callback Queue**. Useful for tasks that require immediate attention.
- **Callback Queue (Macrotask Queue)**: Also known as the 'Task Queue,' it manages events and I/O operations.

Event Loop Phases

- **Timers**: Manages timer events for scheduled tasks.
- **Pending callbacks**: Handles system events such as I/O, which are typically queued by the kernel.
- **Idle / prepare**: Ensures internal actions are managed before I/O events handling.
- **Poll**: Retrieves New I/O events.
- **Check**: Executes 'setImmediate' functions.
- **Close**: Handles close events, such as 'socket.close'.

Task Scheduling: microtasks and macrotasks

- **Microtasks (process.nextTick and Promises)**: Executed after each task.
- **Macrotasks**: Executed after the poll phase when the event loop is not behind any file I/O or scheduled time. This includes timers, setImmediate, and I/O events.

Code Example: Timers and Task Queues

Here is the JavaScript code:

```
// Code Example
console.log('Start');

setTimeout(() => {
  console.log('Set Timeout - 1');

  Promise.resolve().then(() => {
    console.log('Promise - 1');
  }).then(() => {
    console.log('Promise - 2');
  });

}, 0);

setImmediate(() => {
  console.log('Set Immediate');
});

process.nextTick(() => {
  console.log('Next Tick');
  // It's like an infinite loop point for microtask queue
  process.nextTick(() => console.log('Next Tick - nested'));
});

fs.readFile(file, 'utf-8', (err, data) => {
  if (err) throw err;
  console.log('File Read');
});

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

5. What is the difference between Node.js and traditional web server technologies?

Node.js revolutionized server-side development with its non-blocking, event-driven architecture. Let's look at how it differs from traditional web servers and how it leverages a **Single Input-Output (I/O)** model.

Key Distinctions

Multi-threading (Traditional Servers) vs. Event Loop (Node.js)

- **Traditional Servers**: Employ multi-threading. Each client request spawns a new thread, requiring resources even when idle.
- **Node.js**: Utilizes a single-thread with non-blocking, asynchronous functions for I/O tasks. This makes it exceptionally suitable for scenarios like real-time updates and microservices.

Blocking vs. Non-blocking I/O

- **Traditional Servers**: Primarily rely on blocking I/O, meaning that the server waits for each I/O operation to finish before moving on to the next task.
- **Node.js**: Leverages non-blocking I/O, allowing the server to continue handling other tasks while waiting for I/O operations. Callbacks, Promises, and async/await support this approach.

Language Consistency

- **Traditional Servers**: Often pair with languages like Java, C#, or PHP for server-side logic. Front-end developers might need to be proficient in both the server language and client-side technologies like JavaScript.
- **Node.js**: Employs JavaScript both client-side and server-side, fostering full-stack developer coherence and code reusability.

Code Execution

- **Traditional Servers**: Generally compile and execute code. Alterations might necessitate recompilation and possible downtime.
- **Node.js**: Facilitates a "write, save, and run" approach, without the need for recompilation.

Package Management

- **Traditional Servers**: Rely on package managers like Maven or NuGet, with each language typically having its own package dependency system.
- **Node.js**: Centralizes dependency management via npm, simplifying the sharing and integration of libraries.

Deployment

- **Traditional Servers**: Often necessitate coordination with systems, database administrators, and IT teams for deployment.
- **Node.js**: Offers flexible, straightforward deployments. It's especially suited for cloud-native applications.

Use Cases

- **Traditional Servers**: Ideal for enterprise systems, legacy applications, or when extensive computational tasks are required.
- **Node.js**: Well-suited for data-intensive, real-time applications like collaborative tools, gaming, or social media platforms. Its lightweight, scalable nature also complements cloud deployments.

6. Explain what "non-blocking" means in Node.js.

Node.js leverages non-blocking I/O to handle multiple operations without waiting for each to complete separately.

This particular I/O model, coupled with the event-driven paradigm of Node.js, is key to its high performance and scalability, making it **ideal** for tasks such as data streaming, background tasks, and concurrent operations.

Non-Blocking I/O

With non-blocking I/O, an application **doesn't halt** or wait for a resource to become available. Instead, it goes on executing other tasks that don't depend on that resource.

For instance, if a file operation is in progress, Node.js doesn't pause the entire application until the file is read or written. This allows for a more responsive and efficient system, especially when handling multiple, concurrent I/O operations.

Event Loop

Node.js constantly monitors tasks and I/O operations. When a task or operation is ready, it triggers an event. This mechanism is referred to as the **event loop**.

When an event fires, a corresponding event handler or callback function is executed.

Concurrency Without Threads

Traditionally, concurrency can be achieved in languages that support multithreading (e.g., Java). However, managing and coordinating multiple threads can be challenging and is a common source of bugs.

Node.js, on the other hand, provides a simplified yet effective concurrency model using non-blocking I/O and the event loop. It achieves parallelism through mechanisms such as **callbacks**, **Promises**, and

async/await.

By not using threads, Node.js eliminates many of the complexities associated with traditional multithreaded architectures, making it easier to **develop** and **maintain** applications, particularly those requiring high concurrency.

Code Example: File I/O

Here is the JavaScript code:

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

// Perform non-blocking file read operation
fs.readFile('path/to/file', (err, data) => {
    if (err) throw err;
    console.log(data);
});

// Other non-blocking operations continue without waiting for file read
console.log('This message is displayed immediately.');
```

In this example, the file read operation is non-blocking. Node.js does not halt the thread of execution to wait for the file read to complete. Instead, the supplied callback function is invoked when the read operation finishes.

7. How do you update Node.js to the latest version?

Regular updates ensure that your **Node.js** setup is secure, efficient, and equipped with the latest features. Here's how to keep it up-to-date.

Using Package Managers

- **NPM**: Run the following commands to find and install the latest stable version of Node.js:

```
npm cache clean -f
```

```
npm install -g n
```

- **Yarn**: Execute the following command that fetches the latest version and updates Node.js in your system:

```
yarn global add n
```

Using the Official Installer

You can use the official installer to upgrade to the latest stable version.

Version Management Tools

Tools like **nvm** (Node Version Manager), **n** (Node Version Manager) and **nvs** (Node Version Switcher) can be convenient for managing multiple Node.js versions and performing updates.

Windows via Scoop

On Windows, **Scoop** simplifies the task of updating:

```
scoop update nodejs-lts
```

Check the Updated Version

Verify that the update was successful by checking the version number:

```
node -v
```

8. What is "npm" and what is it used for?

npm (Node Package Manager) is a powerful and highly popular package manager that is focused on the Node.js environment. Its primary purpose is to simplify the installation, management, and sharing of libraries or tools written in Node.js.

npm is more than just a package manager: It's also a thriving ecosystem, offering a plethora of ready-to-use modules and tools, thereby making the development workflow for Node.js even more efficient.

Key Functions

- **Package Installation**: npm makes it easy to install and specify dependencies for Node.js applications. Developers can simply define required packages in a `package.json` file, and npm resolves and installs all dependencies.
- **Dependency Management**: npm establishes a tiered dependency system, effectively managing the versions and interdependencies of various packages.
- **Registry Access**: It acts as a central repository for Node.js packages, where developers can host, discover, and access modules.
- **Version Control**: npm enables version control to ensure consistent and predictable package installations. It supports features such as semantic versioning and lock files.
- **Lifecycle Scripts**: It allows developers to define custom scripts for tasks like application start or build, making it convenient to execute routine operations.
- **Packaging and Publication**: Developers can use npm to bundle their applications and publish them, ready for use by others.

npm Client and Registry

- The **npm client** is the command-line tool that developers interact with locally. It provides a set of commands to manage a project's packages, scripts, and configuration.
- The **npm registry** is a global, central database of published Node.js packages. It's where modules and libraries are made available to the Node.js community. The official, public registry is managed by npm, Inc.

npm vs yarn

- **yarn** is another popular package manager, introduced by Facebook. Like npm, it's designed for Node.js and excels in areas like performance and determinism. However, both npm and yarn are continuously evolving, and their differences are becoming more nuanced.

Famous Commands

- **install**: This command downloads and installs the specified packages and their dependencies.
- **init**: This command initializes a `package.json` file for the project.
- **start**: This command typically begins the execution of a Node.js application, as specified in the `scripts` section of `package.json`.
- **publish**: This command is used to publish the package to the npm registry.

npm Scripts

One of the key features of npm is the ability to define scripts in the `package.json` file, executing them with the `npm run` command. This allows for automation of tasks such as testing, building, and starting the application.

These scripts have access to a variety of built-in and environment-specific variables, helping you to customize the script's behavior.

For example:

In `package.json`:

```
{
  "scripts": {
    "start": "node server.js"
  }
}
```

You can then execute:

```
npm start
```

to start the server.

npm Web Interface

While most developers interact with npm via the command line, it also offers a web interface called `npmjs.com`. The website allows users to search for packages, view documentation, and explore related modules. It is also where developers publish and manage their packages.

9. How do you manage packages in a Node.js project?

Node.js utilizes **npm** (Node Package Manager) or yarn for **package management**.

npm vs. Yarn

Both tools create a `node_modules` folder, but they have subtle differences:

- **Yarn's** `yarn.lock` provides deterministic package versions, while npm uses `package-lock.json`.

- npm uses `npm install` while Yarn uses `yarn add` to install a package.

Yarn also has advanced features like parallel package installations and a lockfile ensuring consistent installations across machines.

Core npm Commands

- **npm init**: Initializes a new project and creates a `package.json` file.
- **npm install [package] (-D)**: Installs a package and updates the `package.json` file. The `-D` flag indicates a devDependency.
- **npm update [package]**: Updates installed packages to their latest versions.

Using npm Scripts

The `package.json` can include custom scripts for tasks like testing, building, and deployment, opening up the terminal from the current project directory and running `npm run SCRIPT_NAME`.

CLI Examples

- **Install lodash**: `npm install lodash`
- **Install express and save as a devDependency**: `npm install express --save-dev`
- **Update all packages**: `npm update`

10. What is a package.json file?

The **package.json** file in Node.js projects contains valuable information, such as project metadata and dependencies. This file is essential for managing project modules, scripts, and version control and helps ensure the consistency and stability of your project.

Key Elements

The `package.json` file consists of several essential sections:

1. **Name and Version**: Required elements that identify the project and its version.
2. **Dependencies**: Separated into `dependencies`, `devDependencies`, and `optionalDependencies` which list package dependencies needed for development, production, or as optional features, respectively.

3. **Scripts**: Encompasses a series of custom commands, managed by npm or yarn, that can be executed to perform various tasks.
4. **Git Repository Information**: Optional but helpful for version control.
5. **Project Metadata**: Such as the description and the author-related details.
6. **Peer Dependencies**: A list of dependencies that must be installed alongside the module but are not bundled with it.
7. **Private/Public Status**: Indicates whether the package is publicly available.

Creating `package.json`

You can **generate** the initial `package.json` file by running `npm init` or `yarn init` in the project directory. This command will guide you through a set of interactive prompts to configure your project.

Managing Dependencies

Adding Packages

To add a package to your project, use `npm install package-name` or `yarn add package-name`. This will also automatically update your `package.json` file.

Removing Packages

Remove a package from the project and update the `package.json` file by running `npm uninstall package-name` or `yarn remove package-name`.

Scripts

The `scripts` section allows you to define **task shortcuts**. Each entry is a command or group of sub-commands that can be invoked via `npm run` or `yarn run`.

For example, the following `scripts` section would enable the executing of `babel src -d lib` by running `npm run build`.

```
{
  "scripts": {
    "build": "babel src -d lib"
```

```
    }
  }
```

Using `package.json` in CI/CD Pipelines

When using services like Travis CI, the `package.json` file is crucial for both setting the project environment and defining any required test steps and deployment commands.

For instance, you might use the `scripts` section to specify the test command:

```
{
  "scripts": {
    "test": "mocha"
  }
}
```

During the Travis CI build, you can run `npm test` to execute Mocha tests as per the `package.json` configuration.

Best Practices

- **Regular Updates**: Keep your dependencies up to date, especially any security patches or bug fixes.
- **Conservative Versioning**: Use `^` for minor upgrades and `~` for patch upgrades to maximize stability and compatibility.
- **Try out 'npm' & 'yarn'**: Both are reliable package managers, so pick one that best suits your workflow.

11. Describe some of the core modules of Node.js.

Node.js offers a host of inbuilt modules that cover diverse functionalities, ranging from file system handling to HTTP server management. These modules expedite development and allow for more streamlined application building.

Core Modules Overview

Major Categories

- **Basic/System Control**: Modules optimized for system interaction, diagnostics, and error handling.
- **File System Handling**: Offers a range of file operations.
- **Networking**: Specialized for data communication over various network protocols.
- **Utility Modules**: Miscellaneous tools for data analysis, task scheduling, etc.

Key Modules

Basic/System Control

- `os` : Provides system-related utility functions. Example: `os.freemem()` , `os.totalmem()` .
- `util` : General utility functions primarily used for debugging. Example: `util.inspect()` .

File System Handling

- `fs` : Offers extensive file system capabilities. Commonly used methods include `fs.readFile()` and `fs.writeFile()` .

Networking

- `http` / `https` : Implements web server and client. Example: `http.createServer()` .
- `net` : Facilitates low-level networking tasks. Example: `net.createServer()` .
- `dgram` : Delivers UDP Datagram Socket support for messaging.

Utility Modules

- `crypto` : Encompasses cryptographic operations. Common methods include `crypto.createHash()` and `crypto.createHmac()` .
- `zlib` : Offers data compression capabilities integrated with various modules like `http` .
- `stream` : Facilitates event-based data stream processing.

Others

- `path` : Aids in file path string manipulation.
- `url` : Parses and formats URL strings, especially beneficial in web applications and server operations.

Code Example: Using Core Modules

Here is the node.js code:

```
const os = require('os');
const fs = require('fs');
const http = require('http');
const path = require('path');
const url = require('url');
const zlib = require('zlib');

// Module: os
console.log('Free memory:', os.freemem());
console.log('Total memory:', os.totalmem());

// Module: fs
fs.readFile('input.txt', 'utf8', (err, data) => {
  if (err) throw err;
  console.log(data);
});

// Module: http
http.createServer((req, res) => {
  const reqPath = url.parse(req.url).pathname;
  const file = path.join(__dirname, reqPath);

  const readStream = fs.createReadStream(file);
  readStream.pipe(zlib.createGzip()).pipe(res);
}).listen(8080);
```

12. How do you create a simple server in Node.js using the HTTP module?

Let's look at how to create a simple server in Node.js using the built-in `http` module.

Server Setup

First, a few steps are necessary.

1. **Import the Module**: Use `require` to load the `http` module.
2. **Define Callback Function**: For each request, the server will execute a specific callback function. This function takes two parameters:

- `request` : Represents the HTTP request, from which you can extract any necessary data.
- `response` : Use this parameter to define what the server sends back to the client.

3. **Server Initialization**: Use the `http.createServer` method to set up the server and define the callback function.
4. **Listen on a Port**: Use the `.listen` method to specify the port the server should "listen" on, waiting for incoming requests.

Code Example: Server Setup

Here is the Node.js code:

```
// Import the http module
const http = require('http');

// Define the callback function
const requestListener = (req, res) => {
  res.writeHead(200);
  res.end('Hello, World!');
};

// Server initialization
const server = http.createServer(requestListener);

// Listen on port 8080
server.listen(8080);
```

Request Handler

The **Request listener** is the main entry to the server. This callback function handles the incoming client request and sends a response back to the client.

The `req` object represents the HTTP request that the server receives. It provides all the details about the request, such as the request URL, request headers, request method, and more.

The `res` object is the server's response to the client. You can use methods on this object, like `res.write()` and `res.end()` , to send data back to the client. In most cases, you'll use `res.end()` to send a response.

Code Example: Request Listener with More Capabilities

Here is the Node.js code:

```
const requestListener = (req, res) => {
  if(req.url === '/profile') {
    res.writeHead(200);
    res.end('Welcome to your profile!');
  } else {
    res.writeHead(200);
    res.end('Hello, World!');
  }
};
```

In this example, we're checking the request URL. If it's `/profile`, the server will respond with a "Welcome!" message; otherwise, it will respond with "Hello, World!".

This server is basic yet powerful. With this foundational understanding, you can extend the server's behavior in numerous ways, such as by serving dynamic content or handling different HTTP methods like `POST` and `PUT`.

13. Explain the purpose of the File System (fs) module.

The **File System (fs)** module in Node.js facilitates file operations such as reading, writing, and manipulation. It's a core module, meaning it's available without needing 3rd-party installations.

Key Methods of the `fs` Module

- **Asynchronous Methods**: Ideal for non-blocking file I/O operations. Their function names end with `File`.
- **Synchronous Methods**: Best suited for simpler scripts and robustness is needed.
- **File Names**: As a convention, file and folder names in the Node.js `fs` module that correspond to methods end with `Sync` to indicate synchronous operations (e.g., `renameSync`).

The Synchronous Approach

Though the synchronous file methods can make scripting simpler, their use should be limited in web servers as they can block the event loop, reducing scalability and performance.

Synchronous operations in Node's `fs` module are best avoided in server-side applications that must manage many connections.

Supported Operations

The `fs` module covers a wide array of file-handling tasks, including:

- **I/O Operations**: Read or write files using streams or high-level functions.
- **File Metadata**: Obtain attributes such as size or timestamps.
- **Directories**: Manage folders and the files within them, including sync and async variants for listing.
- **File Types**: Distinguish between files and directories.
- **Links**: Create and manage hard or symbolic links.
- **Permissions and Ownership**: Integrate with operating systems' security systems.

Code Example: File Reading

Here is the Node.js code:

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

// Asynchronous read
fs.readFile('input.txt', (err, data) => {
  if (err) {
    return console.error(err);
  }
  console.log('Asynchronous read: ' + data.toString());
});

// Synchronous read
const data = fs.readFileSync('input.txt');
console.log('Synchronous read: ' + data.toString());
```

In the above code, both asynchronous and synchronous methods are demonstrated for file reading.

Considerations for the Web

When working with HTTP connections or in web applications, the **synchronous methods may block other requests**. Always favor their asynchronous counterparts, especially in web applications.

14. What is the Buffer class in Node.js?

In **Node.js**, the `Buffer` class is a core module that provides a way to **read**, **manipulate**, and **allocate** binary data, which primarily represents a sequence of bytes (octets).

Key Features

- **Backbone of I/O Operations**: Buffers serve as the primary data structure for handling I/O in Node.js, acting as a transient container for data being read from or written to streams and files.
- **Raw Binary Data**: Buffers are used for handling raw binary data, which is particularly useful for tasks like cryptography, network protocols, and WebGL operations.
- **Unmodifiable Size**: Buffers are fixed in size after allocation. To resize a buffer, you'd need to create a new buffer with the necessary size and optionally copy over the original data.
- **Shared Memory**: Buffers provide a mechanism for sharing memory between Node.js instances or between Node.js and C++ Addons, offering enhanced performance in certain scenarios.

Common Use Cases

- **File and Network Operations**: Buffers are leveraged for reading and writing data from files, sockets, and other sources/sinks.
- **Data Conversion**: For example, converting text to binary data or vice versa using character encodings such as UTF-8.
- **Binary Calculations**: Buffers make binary manipulations more manageable, such as computing checksums or parsing binary file formats.

Code Example: Buffer Use

Here is the JavaScript code:

```
let bufTemp = Buffer.from('Hey!');
console.log(bufTemp.toString()); // Output: Hey!

let bufAlloc = Buffer.alloc(5, 'a');
console.log(bufAlloc.toString()); // Output: aaaaa

bufAlloc.write('Hello');
console.log(bufAlloc.toString()); // Output: Hello

let bufSlice = bufAlloc.slice(0, 3);  // Slice the buffer
console.log(bufSlice.toString());  // Output: Hel
```

15. What are streams in Node.js and what types are available?

Node.js utilizes **streams** for efficient handling of input/output data, offering two main varieties: readable and writable.

Categories of Streams

1. **Standard Streams**: Represent standard input, output, and error. These are instances of Readable or Writable streams.
2. **Duplex Streams**: Facilitate both reading and writing. They can be connected to processes or handling pipelines.
3. **Transform Streams**: A special type that acts as an intermediary, modifying the data as it passes through.

Practical Implementations

- **HTTP Transactions**: HTTP clients use readable and writable streams for sending requests and receiving responses. HTTP servers also apply these streams for similar actions in the opposite direction.
- **File System**: Reading and writing files in Node.js utilizes these streams. For instance, the `fs.createReadStream()` method generates a readable stream whereas `fs.createWriteStream()` creates a writable one.

Workflows

1. **Standard I/O Streams**: These support interactivity between a program and its running environment. For example, stdout (a writable stream) can be used to display information, and stdin (a readable stream) can capture user input.
2. **File Operations**: Streams are beneficial when working with large files. This is because they streamline the process by breaking it down into smaller, manageable chunks, thereby conserving memory.
3. **Server Operations**: Streams facilitate data transfer for operations such as network requests, database communications, and more.

4. **Pipelines**: Streams can be easily combined using `pipe()` to create powerful, efficient operations called pipelines. For instance, to compress a file and then write it to disk, you can pipe a readable stream to a transform stream and then to a writable stream. This arrangement neatly dictates the flow of data.