Node.js

TABLE OF CONTENTS

Preface 1
Introduction 1
Advantages and Disadvantages of Node.js 1
Installing Node.js 1
Node.js Architecture 2
Node.js Stack 3
The Event Loop 3
Common APIs 4
The Node Package Manager (NPM) 6
Node.js "Hello Word" Example 7
Pyramid Of Doom 8
Popular Modules And Frameworks 10
Resources 11

PREFACE

This Node.js Cheatsheet has been meticulously crafted to serve as a quick reference guide for Node.js developers of all levels. It provides a concise and organized collection of essential information, tips, and best practices to help you navigate the Node.js ecosystem with confidence and efficiency.

INTRODUCTION

Node.js is an open-source, server-side runtime environment that allows developers to build scalable, high-performance web applications using JavaScript. It is designed to be lightweight, efficient, and ideal for building real-time, data-intensive applications. Node.js is based on the V8 JavaScript engine and is known for its event-driven, non-blocking I/O model, making it a popular choice for building network applications and APIs.

Node.js has gained widespread popularity in web development due to its versatility and the ability to use a single language, JavaScript, on both the client and server sides. It has a vast ecosystem of modules and packages available through NPM (Node Package Manager), enabling developers to streamline their development process.

ADVANTAGES AND DISADVANTAGES OF NODE.JS

These advantages and disadvantages can help you understand when Node.js is an appropriate choice for your project and when it might have limitations. Remember that the suitability of Node.js depends on the specific requirements of your application and your team's familiarity with asynchronous JavaScript programming.

Advantages	Disadvantages
High Performance: Node.js is built on the V8 JavaScript engine, which compiles JavaScript into native machine code, resulting in exceptional performance.	**Single-Threaded**: While Node.js's event loop is efficient, it's still single-threaded, which can lead to performance bottlenecks for CPU-bound tasks.
Non-Blocking I/O: Node.js uses an event-driven, non-blocking I/O model, allowing for efficient handling of a large number of concurrent connections.	**Limited Multithreading**: Although Node.js can use worker threads, it lacks native support for true multithreading, which is a limitation for certain applications.
Vast Ecosystem: Node.js has a rich ecosystem of modules and packages available through NPM, making it easy to find and reuse code.	**Learning Curve**: Developers new to asynchronous programming may face a learning curve when working with Node.js.
Cross-Platform: Node.js runs on various platforms, including Windows, macOS, and Linux, providing a consistent development environment.	**Debugging Challenges**: Debugging asynchronous code can be more challenging compared to synchronous code.
Real-Time Capabilities: Node.js is excellent for building real-time applications like chat apps and online gaming, thanks to its low-latency and event-driven architecture.	**Callback Hell**: Managing asynchronous code using callbacks can lead to complex and hard-to-maintain code structures, commonly referred to as "callback hell."
Community and Support: Node.js has a large and active community, offering extensive documentation, tutorials, and support.	
Scalability: Node.js applications can be easily scaled horizontally by adding more server instances due to their lightweight nature.	

INSTALLING NODE.JS

To download and install Node.js on your computer, follow these steps:

1. Visit the Node.js website

Go to the official Node.js website at https://nodejs.org.

2. Choose the Node.js version

You'll find two versions of Node.js on the website: LTS (Long-Term Support) and Current. For most production and development purposes, it's recommended to choose the LTS version as it's more stable and receives long-term support. Click on the "LTS" tab to see the LTS version.

3. Download the installer

Click on the appropriate installer for your operating system. Node.js supports various operating systems, including Windows, macOS, and Linux. Choose the installer that matches your system. For example, on Windows, you can choose the ".msi" installer, while on macOS, you'll want the ".pkg" installer.

4. Run the installer

After downloading the installer, run it by double-clicking on the downloaded file. This will launch the Node.js installer.

5. Install Node.js

Follow the on-screen instructions in the installer to install Node.js on your system. You can typically accept the default settings, but you may customize the installation directory if needed.

6. Verify the installation

After the installation is complete, open a terminal or command prompt, and type the following commands to verify that Node.js and NPM (Node Package Manager) are correctly installed:

```
node -v
npm -v
```

These commands should display the versions of Node.js and NPM, confirming that the installation was successful.

7. Optional: Install a code editor (IDE)

To start developing with Node.js, you may want to install a code editor or integrated development environment (IDE). Popular choices include Visual Studio Code, Sublime Text, and WebStorm.

NODE.JS ARCHITECTURE

Node.js has a unique architecture that sets it apart from traditional server-side technologies. Understanding its event-driven, non-blocking nature empowers developers to create responsive, real-time applications and manage I/O-bound operations effectively.

The main difference between synchronous and asynchronous programming lies in how they handle tasks and operations that may take time to complete. Synchronous programming executes tasks one after another, in a blocking manner, while asynchronous programming enables concurrent execution and continues working on other tasks without waiting for operations to complete. Asynchronous programming is especially beneficial for I/O-bound operations and situations where responsiveness and resource utilization are essential, but it can be more challenging to write and reason about due to its event-driven nature.

The key components of Node.js architecture include:

- **V8 Engine**: At the core of Node.js is the V8 JavaScript engine, developed by Google. V8 is known for its high-performance JavaScript execution and just-in-time (JIT) compilation of JavaScript code into native machine code. This enables Node.js to execute JavaScript with remarkable speed.
- **Libuv**: Libuv is a C library that Node.js uses to abstract and handle I/O operations and provide an event loop. It enables Node.js to work across various operating systems, managing tasks like file system operations and network communication in an efficient, non-blocking manner.
- **Event Loop**: The event loop is the central part of Node.js, responsible for managing asynchronous operations. It allows Node.js to handle I/O operations, such as file reading, network requests, and database queries, without blocking the entire process. Node.js is single-threaded, but the event loop's non-blocking I/O operations make it highly efficient

for handling a large number of concurrent connections.

- **Callback Queue**: Callbacks are functions that are executed once an asynchronous operation is completed. Node.js relies on a callback mechanism to handle asynchronous code. When an asynchronous task finishes, its callback is placed in the callback queue, and the event loop processes these callbacks in a first-in, first-out (FIFO) order.

NODE.JS STACK

The Node.js stack consists of several layers, starting with the V8 engine, linuv and other native support libraries at the core.

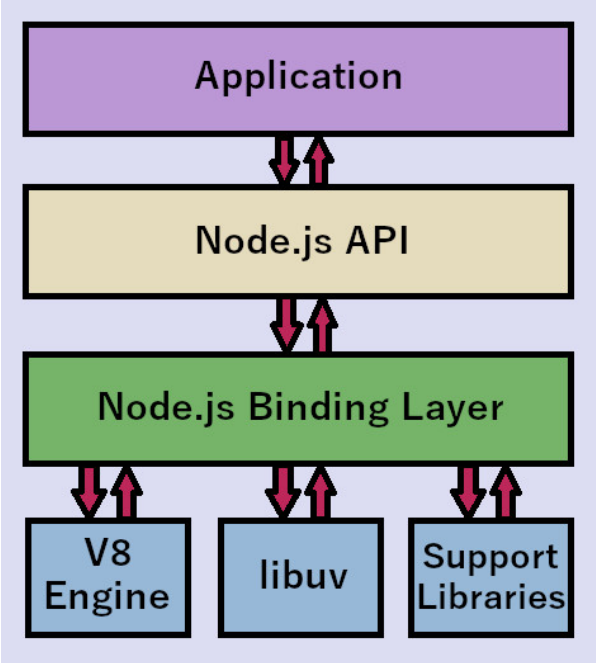

Figure 1. Node.js Stack

The Node.js binding layer acts as a bridge between JavaScript and C/C++ code. It allows Node.js to incorporate native code modules, such as those used to interact with hardware or low-level system functions. This layer is responsible for managing and exposing native functionality to JavaScript, making it accessible in Node.js applications.

On top of the binding layer resides the Node.js API layer which provides a wide range of JavaScript modules and functions that allow developers to work with the file system, network, cryptography, streams, and more. These modules simplify common tasks and make them accessible through JavaScript code. The Node.js API layer essentially serves as a set of built-in libraries that extend the capabilities of JavaScript beyond what's available in a browser environment.

Finally, the application layer is where Node.js developers write their JavaScript code to create server-side applications and services. This layer includes the user's custom JavaScript code, which utilizes the Node.js API and any additional third-party modules installed via NPM. Developers build web servers, APIs, real-time applications, and various other server-side applications in this layer. They also handle business logic and application-specific functionality.

THE EVENT LOOP

Node.js is single-threaded, which means it runs on a single main thread of execution. This single thread is responsible for executing JavaScript code and handling I/O operations. Nevertheless, it is also non-blocking, meaning it doesn't wait for I/O operations (like file reading, network requests) to complete. Instead, it initiates the operation and continues executing other code while it waits for the I/O to complete. In other words, Node.js is event-driven, meaning it relies on events and callbacks to handle asynchronous operations. It uses event emitters and listeners to manage events and responses.

The event loop is a loop that continually checks if there are any pending events, such as I/O operations or timer expirations. When an event is detected, the associated callback function is added to a callback queue for execution. The callback queue holds all the callback functions that are waiting to be executed. Callbacks are processed in the order they were added to the callback queue (FIFO).

The event loop follows a repeating cycle. Each phase in the cycle has its own callback queue and only when the callback queue has been exhausted (or the callback limit is reached), the event loop moves to the next phase:

- **Timers**: Check the event queue for timer expiration events and executes any related callbacks (e.g. registered using the `setTimeout()` or `setInterval()` functions).
- **Pending Callbacks**: Executes I/O-related callbacks e.g., regarding file read or network request events, deferred to the next loop

iteration.

- **Idle, Prepare**: Internal mechanisms for managing the event loop.
- **Poll**: Check the event queue for new I/O events (blocks if necessary); execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and `setImmediate()`).
- **Check**: Execute any callbacks registered via `setImmediate()`.
- **Close Callbacks**: Execute any 'close' event callbacks, such as `on('close')` for server connections.

Prior moving to a next phase, all callback functions registered using `process.nextTick()` are executed. This is the case for moving to the first phase also. The queue that maintains this type of callbacks is also known as the microtasks queue. Since they are the first to get executed, microtasks have a higher priority than other asynchronous operations, such as timer and I/O related callbacks. `process.nextTick()` is used when you want to ensure that a callback is executed at the completion of the current callback, before other asynchronous tasks, making it useful for scenarios like setting up event listeners before emitting events.

Between each run of the event loop, Node.js checks if it is waiting for any asynchronous I/O or timers and shuts down cleanly if there are not any.

Node.js's event-driven, non-blocking architecture makes it suitable for handling many concurrent connections efficiently. Multiple requests can be processed without the need for creating a new thread for each request, reducing the overhead associated with thread management.

COMMON APIS

Node.js provides a diverse set of built-in APIs that empower developers to perform a wide range of tasks, from working with the file system to creating web servers and handling cryptographic operations. This table provides a quick reference for some of the most commonly used Node.js APIs, along with concise descriptions of their purposes and functionalities. These APIs form the foundation for building feature-rich, high-performance server-side applications in Node.js. A complete list of APIs included with Node.js can be found in the Node.js API documentation.

Node.js Api	Description
`fs` (File System)	Provides functions for working with the file system, allowing you to read, write, delete, and manipulate files and directories.
`http`, `https`, `http2`	Enables you to create HTTP servers and clients, making it possible to build web servers and interact with HTTP resources. Similar to the `http` module `https` is used for HTTPS (secure) connections. `http2` extends the capabilities of the `http` and `https` modules to support HTTP/2, a modern version of the HTTP protocol that enhances performance.
`os` (Operating System)	Provides information about the operating system on which the Node.js application is running, such as CPU architecture and system memory.
`path`	Offers utilities for working with file and directory paths in a cross-platform manner, helping to resolve and manipulate file paths.
`events`	Allows you to work with event emitters and listeners, facilitating event-driven programming and event handling within your application.

Node.js Api	Description
`util`	Provides various utility functions for debugging, formatting, and inspecting objects, as well as other miscellaneous utilities.
`child_process`	Enables the creation of child processes to execute external commands and scripts, making it possible to run additional programs alongside your Node.js application.
`crypto`	Offers cryptographic functionality for secure data hashing, encryption, and decryption, as well as support for SSL and TLS protocols.
`net`	Allows you to create network servers and clients for building custom network protocols and handling TCP/UDP communication.
`dns` (Domain Name System)	Provides DNS resolution and DNS server functionality, enabling the translation of domain names to IP addresses and vice versa.
`url`	Offers utilities for URL parsing and formatting, making it easier to work with URLs, both for HTTP requests and other purposes.
`querystring`	Facilitates the parsing and formatting of URL query strings, allowing for the manipulation of query parameters in URLs.

Node.js Api	Description
`zlib`	Provides compression and decompression functionality using the zlib library, allowing you to work with compressed data streams.
`readline`	Offers a simple way to create interactive command-line interfaces (CLIs) by providing methods for reading input and writing output to the console.
`cluster`	Allows you to create child processes and manage them efficiently to utilize multiple CPU cores for improved performance in a Node.js application.
`stream`	Provides a framework for working with streams of data, enabling efficient data processing and transmission, such as reading/writing large files or network communication.
`buffer`	Represents a fixed-size memory allocation used for working with binary data directly, which is especially important for file I/O and network communication.
`process`	Offers information and control over the Node.js process, including environment variables, command-line arguments, standard I/O, and process events.

Node.js Api	Description
`debugger`	Provides a debugging API and tools for debugging Node.js applications. It allows you to inspect and debug your code during development.
`modules`	Node.js uses the CommonJS module system, allowing you to create, load, and manage modules for better code organization and reusability.

THE NODE PACKAGE MANAGER (NPM)

The Node Package Manager (NPM) is a package manager for JavaScript and Node.js. It is the default package manager that comes bundled with Node.js. NPM plays a central role in managing and distributing open-source JavaScript packages and libraries. Here's a brief description of its main components:

- **Packages:** Packages are reusable JavaScript modules or libraries. A package typically includes JavaScript files, documentation, and metadata. These packages can be installed in Node.js applications, allowing developers to leverage existing code to build applications more efficiently.
- **Registry:** The NPM registry is a massive online collection of public JavaScript packages. It is the central repository where package authors publish their code for others to use. The registry hosts a wide variety of items, ranging from small utility libraries to entire web frameworks.
- **CLI (Command-Line Interface):** The NPM CLI is a command-line tool that you can use to interact with the registry and manage packages. It provides commands for installing, updating, publishing, and managing dependencies.
- `package.json` **File:** Every Node.js project typically includes a `package.json` file, which serves as a manifest for the project. This file contains metadata about the project, including its name, version, dependencies, and scripts. The `package.json` file helps NPM understand the project's requirements and facilitates version management.
- **Dependencies:** Dependencies are packages that a Node.js project relies on to function correctly. These dependencies are listed in the project's `package.json` file. When you run `npm install`, NPM reads the `package.json` and downloads the specified dependencies from the registry. Managing dependencies ensures that your application can be easily replicated on different systems.

To create a new NPM package based on existing files in your project directory, just navigate inside that directory and type the following command: `npm init`

Running this command will prompt you to answer a series of questions to create a `package.json` file for your project. You can press Enter to accept the default values or provide your own answers. After completing the initialization, NPM will generate a `package.json` file in your project directory. Here's an example of what the `package.json` file might look like using default values (<package-name> is replaced with the project's directory name):

```
{
  "name": "<package-name>",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test
specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

To add a new dependency to your Node.js project, you can use the `npm install` command followed by the name of the package you want to install. For this example, let's say you want to add the `lodash` package as a dependency. You can install it using the following command while in your project's top

folder: `npm install lodash`

Under the hood, the `npm install` command downloads the designated package from the registry and installs it, along with all of its dependencies, to the `node_modules/` directory inside your project folder; it then updates your project's `package.json` file to include the newly added dependency. Open the `package.json` file, and you should see an updated `"dependencies"` section that lists the `lodash` package and its version. The latest version available at the time of installation is selected.

```
...
  "dependencies": {
    "lodash": "^4.17.21"
  }
...
```

NPM also allows you to define and execute scripts in your Node.js project. These scripts are defined as key-value pairs in the `scripts` section of your `package.json` file. The key is the script name, and the value is the command to execute when the script is run. For example, let's create a new custom script called `"start"` that will run our Node.js application by executing the command `node app.js` (assuming our application is contained in `app.js`):

```
...
  "scripts": {
    "test": "echo \"Error: no test
specified\" && exit 1",
    "start": "node app.js"
  }
...
```

To run the script, use the `npm run` command followed by the script name like this: `npm run start`

NPM also provides a few built-in scripts that you can use without defining them in the `"scripts"` section. Some common built-in scripts include:

- `npm start`: Runs the `"start"` script if defined.
- `npm test`: Runs the `"test"` script if defined.
- `npm install`: Installs the project's dependencies as listed in `package.json`.

By defining and running scripts in your `package.json` file, you can easily automate common tasks, making it convenient to start, test, and manage your Node.js applications.

NODE.JS "HELLO WORD" EXAMPLE

To run JavaScript files in Node.js, you can use the `node` command followed by the name of the file containing the code to be executed.

For a "hello world" application, that simply outputs a "hello world" message to the console, let's create a JavaScript file, name it `myapp.js` (you are free to choose whatever name you like of course) and add the following code in it:

```
console.log("Hello, Node.js!");
```

Execute the file by using the command `node myapp.js`. Node.js will interpret the JavaScript code and execute it in the Node.js runtime environment. You should see the Hello, Node.js! output in your terminal.

You can run any JavaScript file this way, whether it's a simple script like the one in this example or a more complex Node.js application. For example, a simple HTTP server can be created using the following code:

```
// Import the 'http' module, which
is a built-in module for creating
HTTP servers.
const http = require('http');

// Define the hostname and port for
the server to listen on.
const hostname = '127.0.0.1'; //
Loopback address for the local
machine
const port = 3000; // Port number

// Create an HTTP server using the
'createServer' method of the 'http'
module.
const server = http.createServer
```

```
((req, res) => {
  // Set the response's HTTP status
code and content type.
  res.statusCode = 200;
  res.setHeader('Content-Type',
'text/plain');

  // Write a response message.
  res.end('Hello, Node.js HTTP
Server!\n');
});

// Start the server and make it
listen for incoming HTTP requests.
server.listen(port, hostname);

// Output message that the server
has started
console.log(`Server running at
https://${hostname}:${port}/`);
```

Although Node.js is single-threaded, listening on the specified port will not block execution. Instead, a callback is registered that executes the handler logic defined in the `createServer` function, when an HTTP request is received. Once the callback is registered, the phrase Server running at https://127.0.0.1:3000/ is outputted.

Note that this application requires the `http` package which allows us to create an HTTP server and handle incoming HTTP requests. You can install the `http` package by running `npm install http` while in the top folder of your project.

PYRAMID OF DOOM

The "pyramid of doom" is a term used to describe a situation where multiple levels of nested callbacks result in code that has a deeply nested and indented structure. This nested structure can make your program hard to read, understand, and maintain, resembling a pyramid-like shape.

The "pyramid of doom" typically occurs in scenarios like handling I/O operations, making network requests, or dealing with event-driven programming in general. Here's an example of the "pyramid of doom" scenario using three nested callbacks:

```
fs.readFile('file1.txt', 'utf8',
(err, data1) => {
  if (err) {
    console.error(err);
  } else {
    fs.readFile('file2.txt', 'utf8',
(err, data2) => {
      if (err) {
        console.error(err);
      } else {
        fs.readFile('file3.txt',
'utf8', (err, data3) => {
          if (err) {
            console.error(err);
          } else {
            // Do something with
data1, data2, and data3
          }
        });
      }
    });
  }
});
```

To mitigate the pyramid of doom and improve code readability, several approaches have been adopted in JavaScript and Node.js:

Promises: `Promise` objects provide a way to handle asynchronous operations in a more linear and readable fashion. They allow you to chain `.then()` and `.catch()` calls, avoiding deeply nested callbacks.

```
const fs = require('fs').promises;
// Using Promises for file
operations

fs.readFile('file1.txt', 'utf8')
  .then(data1 => fs.readFile(
'file2.txt', 'utf8'))
  .then(data2 => fs.readFile(
'file3.txt', 'utf8'))
  .then(data3 => {
    // Do something with data1,
data2, and data3
  })
  .catch(err => {
```

```
    console.error(err);
  });
```

Here the `fs.readFile()` function returns a `Promise` object. `Promise` objects support two properties, the state and result of the asynchronous execution of the function that spawned them. A `Promise` can either be in the pending, fulfilled or rejected state. In the fulfilled state the corresponding function has executed successfully and the `Promise` result has the returned value, whereas in the rejected state `Promise` result is the error raised after a failed function execution. To handle the result (either value or error) both `Promise.then()` and/or `Promise.catch()` can be used. `Promise.then()` takes two arguments, a callback for success and an optional callback for failure. `Promise.then()` also returns a `Promise` object regarding the asynchronous execution of either its success or failure callback, to enable chaining.

Async/Await: `async/await` is a modern JavaScript feature that simplifies asynchronous code by allowing you to write it in a more synchronous, structured way. It's built on top of `Promises`.

```
const fs = require('fs').promises;
// Using Promises for file
operations

async function readFiles() {
  try {
    const data1 = await fs.readFile
('file1.txt', 'utf8');
    const data2 = await fs.readFile
('file2.txt', 'utf8');
    const data3 = await fs.readFile
('file3.txt', 'utf8');

    // Do something with data1,
data2, and data3
  } catch (err) {
    console.error(err);
  }
}

readFiles();
```

Here the `async` keyword is used to create an asynchronous function. Asynchronous functions remove much of the boilerplate code required for the resolution of `Promises` and `Promise` chains within the function body. Also the `await` keyword is permitted, which can be used to "wait" for a `Promise` to be resolved and return its resolved value. So, in our example, the three assignments of `data1`, `data2` and `data3`, happen sequentially and only after each respective `Promise` has been resolved.

It is important to note that using `await` does not actually block the executing thread. Using `async /await` just abstracts the underlaying use of `Promises` to allow us to write asynchronous code in a more synchronous manner.

Control Flow Libraries: Libraries like `async.js` offer utilities for controlling the flow of asynchronous operations using functions like `async.waterfall` and `async.series`.

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

async.waterfall([
  function(callback) {
    fs.readFile('file1.txt', 'utf8',
callback);
  },
  function(data1, callback) {
    fs.readFile('file2.txt', 'utf8',
(err, data2) => callback(err, data1,
data2));
  },
  function(data1, data2, callback) {
    fs.readFile('file3.txt', 'utf8',
(err, data3) => callback(err, data1,
data2, data3));
  }
], (err, data1, data2, data3) => {
  if (err) {
    console.error(err);
  } else {
    // Do something with data1,
data2, and data3
  }
});
```

Event Emitters: In event-driven scenarios, you can

use Node.js's event emitter pattern to handle asynchronous events more elegantly.

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

const emitter = new EventEmitter();

emitter.on('data1', () => {
  fs.readFile('file1.txt', 'utf8',
(err, data1) => {
    if (err) {
      console.error(err);
    } else {
      emitter.emit('data2', data1);
    }
  });
});

emitter.on('data2', (data1) => {
  fs.readFile('file2.txt', 'utf8',
(err, data2) => {
    if (err) {
      console.error(err);
    } else {
      emitter.emit('data3', data1,
data2);
    }
  });
});

emitter.on('data3', (data1, data2)
=> {
  fs.readFile('file3.txt', 'utf8',
(err, data3) => {
    if (err) {
      console.error(err);
    } else {
      // Do something with data1,
data2, and data3
    }
  });
});

emitter.emit('data1');
```

These approaches help to organize and simplify asynchronous code, making it more readable and maintainable, while avoiding the pyramid of doom. The choice of which approach to use depends on the specific requirements of your project and your personal coding style.

POPULAR MODULES AND FRAMEWORKS

The Node.js ecosystem thrives on the contributions of third-party developers, resulting in a vast collection of modules and frameworks that extend the capabilities of Node.js for various purposes. In this table, we provide a quick overview of some of the most popular third-party modules and frameworks, along with brief descriptions of their key features and use cases. These tools empower Node.js developers to streamline development, implement real-time features, simplify authentication, and much more. Explore this table to discover the wealth of resources available to enhance your Node.js projects.

Module/framework	Description
Express	Express.js is a widely used web application framework for Node.js. It simplifies the process of building robust and scalable web applications with a rich set of features and middleware. Express is known for its minimalistic and flexible design.
Passport	Passport is an authentication middleware for Node.js applications. It supports various authentication strategies, such as username and password, OAuth, and OpenID, making it easier to implement user authentication and authorization in your web applications.

Module/framework	Description
Sequelize	Sequelize is an Object-Relational Mapping (ORM) library for Node.js that simplifies database interactions with relational databases like MySQL, PostgreSQL, and SQLite. It provides a model-based approach to database operations and supports migrations, validations, and associations.
NestJS	NestJS is a progressive and modular framework for building efficient and scalable server-side applications. It is built with TypeScript and embraces the use of decorators and dependency injection to create structured and maintainable code. NestJS is often used for building APIs and microservices.
Jest	Jest is a popular JavaScript testing framework for Node.js and the browser. It offers a comprehensive and developer-friendly testing experience, with features like test runners, assertion libraries, mocking, and code coverage reporting. Jest is often used for unit testing, integration testing, and end-to-end testing of Node.js applications.
PM2 (Process Manager)	PM2 is a production process manager for Node.js applications. It simplifies process management, load balancing, and monitoring for Node.js apps. It can be used to keep applications running continuously, manage clustering, and provide production-ready environment management for Node.js apps.
Axios	Axios is a popular HTTP client library for making HTTP requests from Node.js applications. It supports promises and async/await, provides an easy-to-use API for sending HTTP requests, and includes features like request/response interception and error handling. Axios is often used for interfacing with RESTful APIs and other HTTP services.

RESOURCES

These resources cover a wide range of learning and development needs for Node.js, whether you're a beginner or an experienced developer looking to stay current with the latest trends and best practices.

- Node.js Official Website: The official website provides downloads, documentation, and updates about Node.js.
- Node.js Documentation: The official documentation is a comprehensive resource for learning about Node.js, its core modules, and best practices.
- npm Website: npm is the package manager for Node.js, offering a vast repository of packages and modules that you can use in your projects. You can search for packages and manage

dependencies using the `npm` command-line tool.

- GitHub Node.js Topic: GitHub hosts many open-source Node.js projects. You can explore, contribute to, and learn from Node.js repositories on GitHub.
- nodeschool.io: NodeSchool offers interactive workshops for learning Node.js and related technologies.
- learnyounode: Learn You The Node.js For Much Win! is a NodeSchool workshop for getting hands-on experience with Node.js.
- Node.js Official Blog: The official Node.js blog provides updates, announcements, and insights into the Node.js ecosystem.