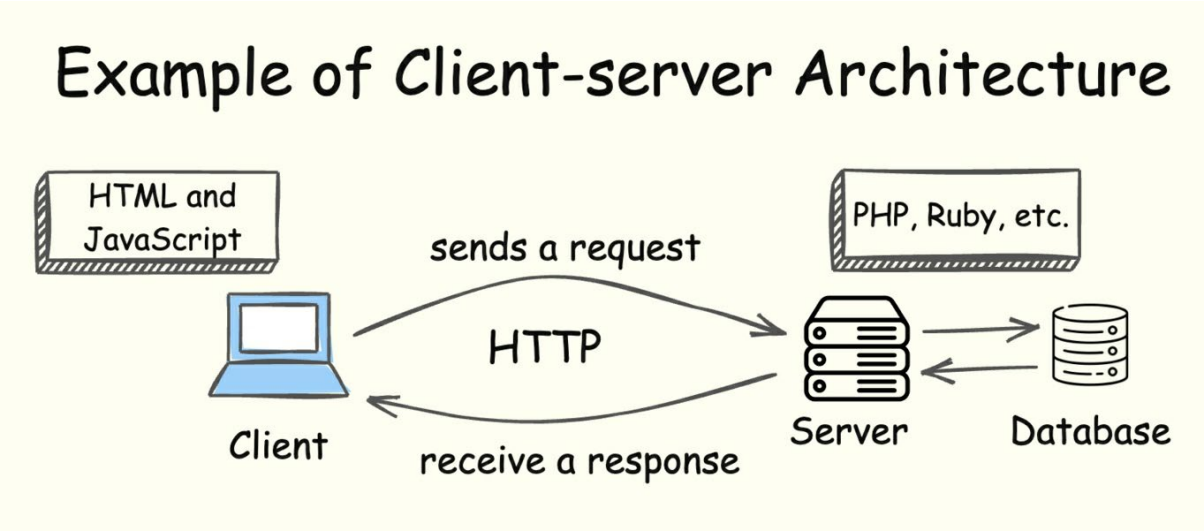
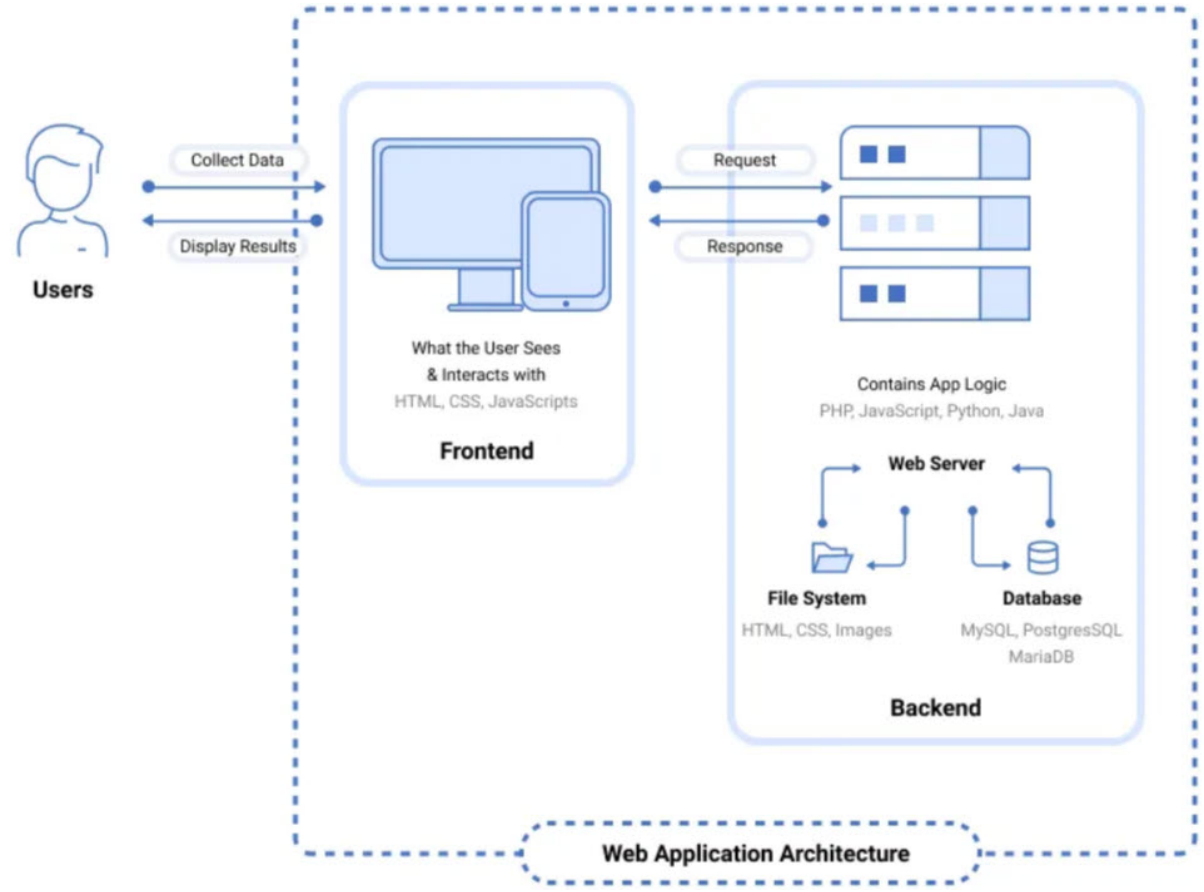
1. Node.js - Introduction

A complete web application consists of two main components: the frontend, also known as the client side, and the backend, or server side. In the Client-Side Web Technologies module, you learned that JavaScript operates exclusively within web browsers. The client-server architecture below shows how a web application is developed:





Client:

The user interacts with the front-end part of a web application. The front-end is usually developed using languages like HTML and CSS styles, along with extensive usage of JavaScript-based frameworks like ReactJS and Angular, which help with application design.

Server:

The server is responsible for taking the client requests, performing the required tasks, and sending responses back to the clients. It acts as a middleware between the front-end and stored data to enable operations on the data by a client. Node.js, PHP, and Java are the most popular technologies in use to develop and maintain a web server.

Database:

The database stores the data for a web application. The data can be created, updated, and deleted whenever the client requests. MySQL and MongoDB are among the most popular databases used to store data for web applications.

The communication between the client and server happens through HTTP requests, enabling a dynamic and interactive web experience.

Since JavaScript is limited to running within the browser, a different programming language is typically needed for building the server side of a web application. Popular server-side languages include PHP, Python, and Ruby.

However, this changed with the development of Node.js, a platform that allows JavaScript to run outside of the browser. With Node.js, JavaScript can now be used to develop both the client and server sides of a web application, as illustrated below:

1.1Node.JS

Node.js is an **open-source**, **cross-platform JavaScript runtime environment** built on Google Chrome's **V8 engine**. It enables the execution of JavaScript code outside of a browser, specifically on the server-side.

Node.js brings JavaScript, a language **traditionally used for client-side scripting, into the backend,** allowing developers to use a single programming language across the entire stack.

Node.js is also highly extensible, with developers contributing a vast range of open-source libraries and tools that streamline the web development process. Additionally, its flexible architecture allows for rapid prototyping of applications.

In summary, Node.js brings JavaScript into the backend, allowing developers to use a single programming language to develop an entire web application.

1.2Comparison with Traditional Server-Side Technologies

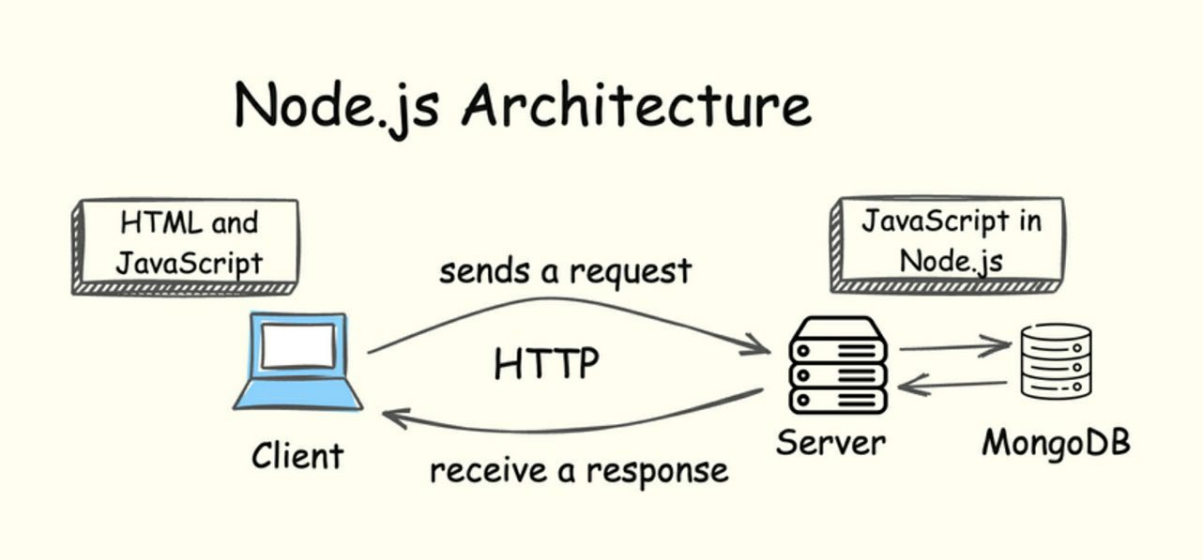
1. **Node.js vs. PHP:**

PHP uses a blocking I/O model, where each request is handled in sequence. Node.js, on the other hand, processes requests asynchronously, resulting in better performance for applications that need to handle a large number of simultaneous connections.

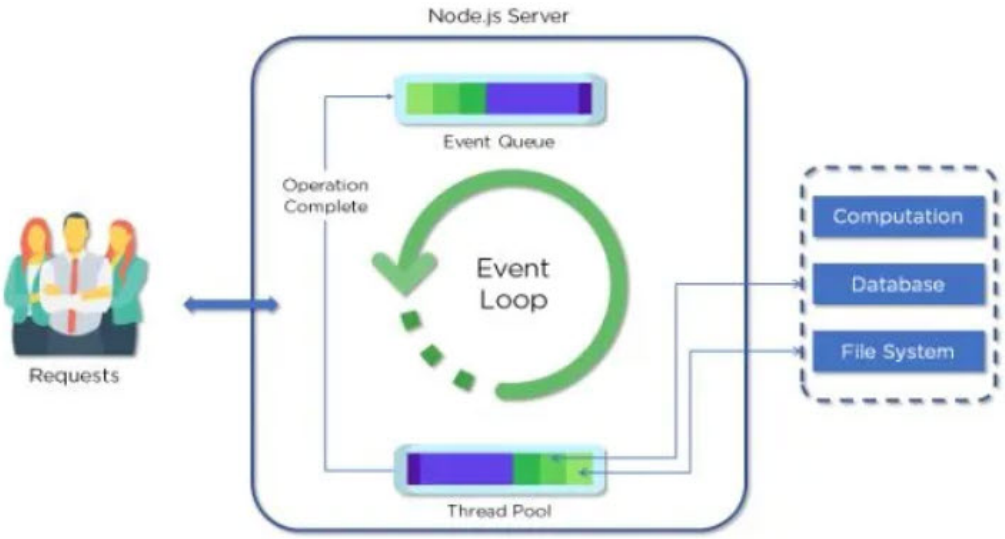
2. **Single Language Advantage**:

In traditional development, you often use one language for the client (JavaScript) and another for the server (PHP, Ruby, etc.). With Node.js, you can use JavaScript for both, leading to more **consistent code** and easier code sharing between client and server.

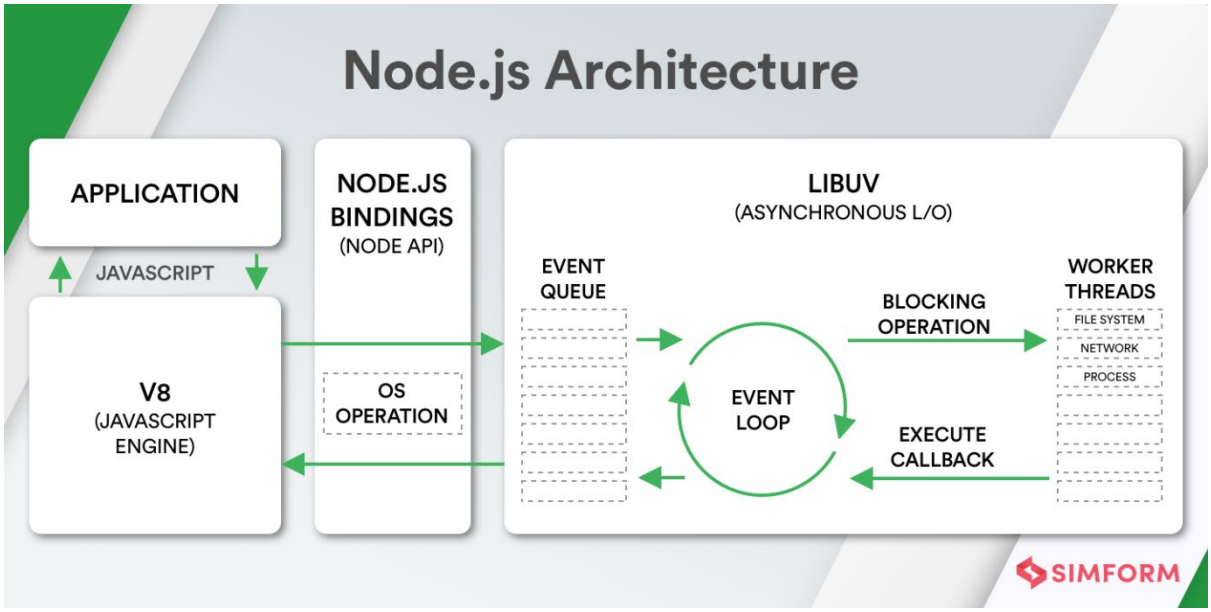
1.3Node.js Architecture



Node.js Architecture:



Node.js operates on a single thread, allowing it to handle thousands of simultaneous event loops. Here's a diagram, provided by Sinform.com, that best illustrates Node.js architecture.

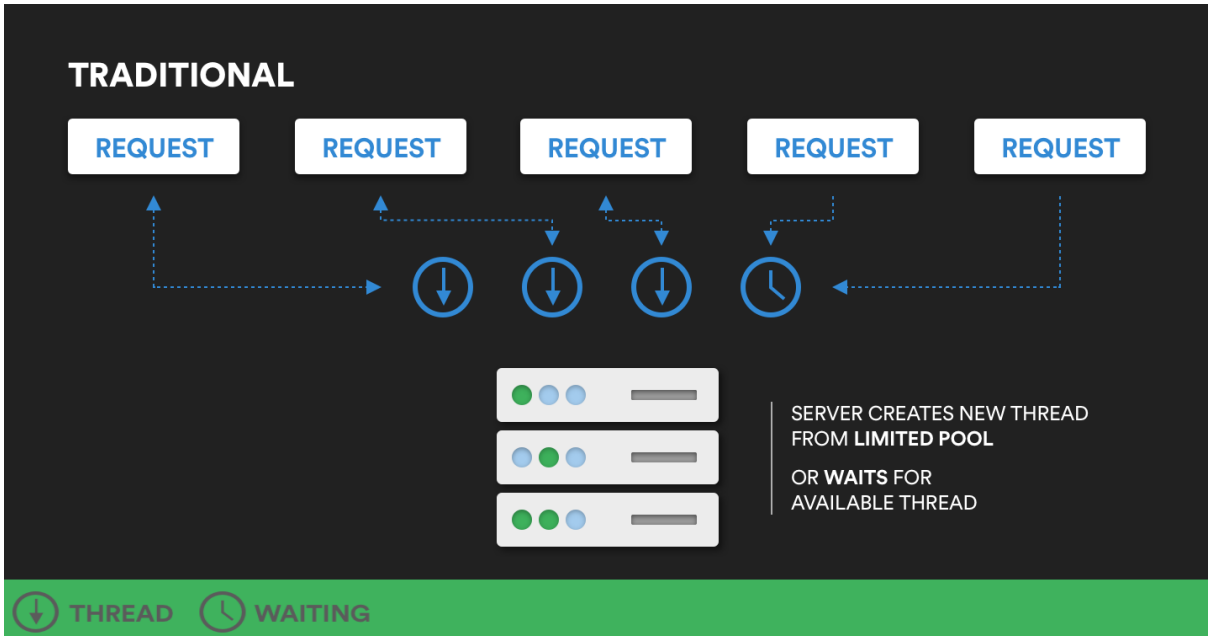


Node.js runs on Google's V8 Javascript engine, where web applications are event-based in an asynchronous manner. Node.js platform uses a "***single-threaded event loop***."

1.4 Difference between multi-threaded request-response and single-threaded event loop

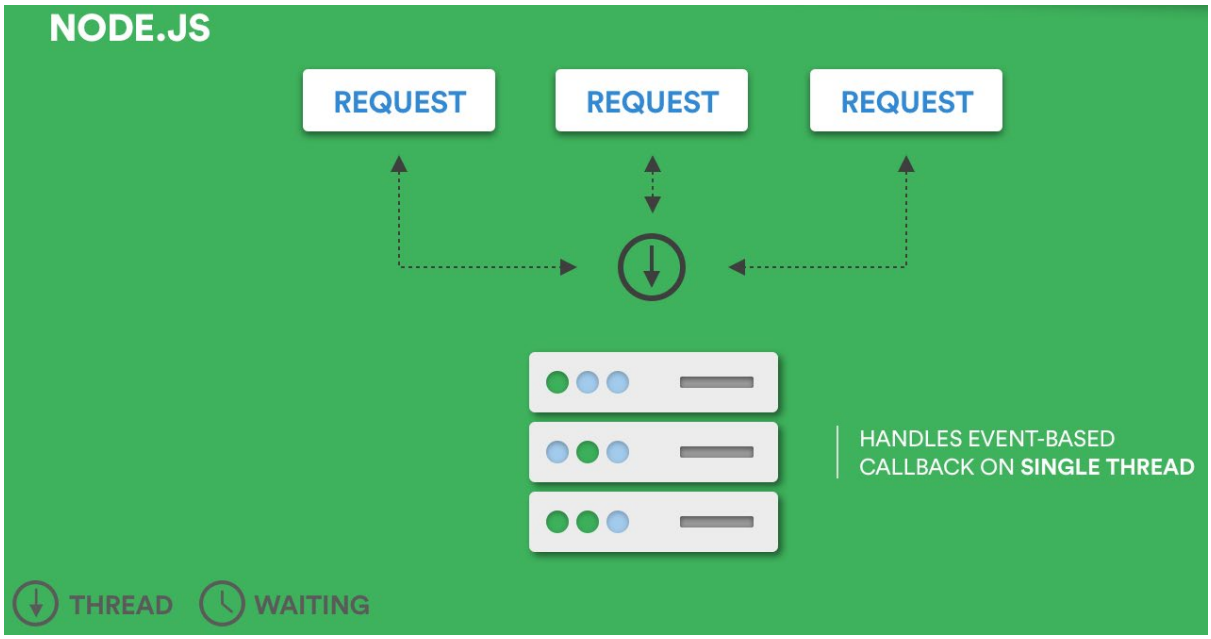
1. Multi-threaded request-response

A *multi-threaded request-response* architecture tends to be slower and cannot handle multiple concurrent threads effectively. Each request generates a new thread, which can quickly fill up system memory (RAM) and lead to performance issues."



2. Single-threaded event loop

Unlike the traditional web-serving technique, where each request creates a new thread, Node.js operates on a single thread. This enables it to support thousands of concurrent connection handling event loops.



The platform does not use a traditional request-response, multi-threaded, stateless model. Instead, it operates on a simplified single-threaded event loop model. Node.js utilizes a library called 'Libuv' to implement this event loop mechanism, which is primarily based on JavaScript's event-driven model and relies heavily on callbacks for asynchronous processing

When a client sends a request to a Node.js server, the request is added to an event queue. The event loop continuously checks this queue and processes each request. If a request involves an I/O operation, Node.js offloads it to the system kernel, which handles it asynchronously. Once the I/O operation is complete, the kernel notifies Node.js, executing the corresponding callback function.

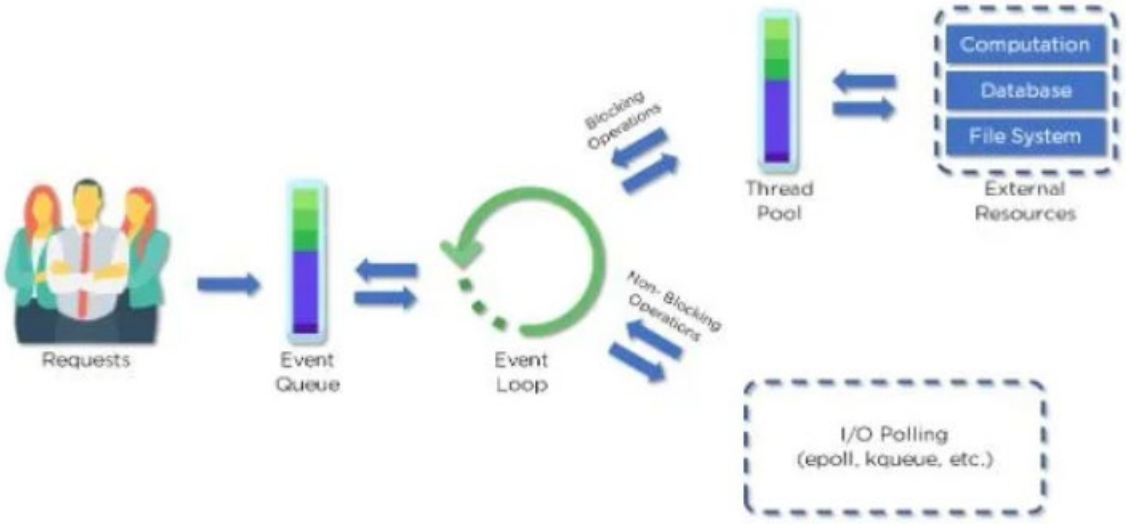
This non-blocking I/O and event-driven model allows Node.js to handle many simultaneous connections efficiently, making it ideal for building scalable, high-performance network applications.

Parts of Node.js Architecture:

- **Requests:** Incoming requests can be blocking (complex) or non-blocking (simple), depending upon the tasks that a user wants to perform in a web application
- **Node.js server:** Node.js server is a server-side platform that takes requests from users, processes those requests, and returns responses to the corresponding users
- **Event Queue:** Event Queue in a Node.js server stores incoming client requests and passes those requests one-by-one into the Event Loop
- **Thread pool:** Thread pool consists of all the threads available for carrying out some tasks that might be required to fulfil client requests
- **Event loop:** Event Loop indefinitely receives requests and processes them, and then returns the responses to corresponding clients.
- **External Resources:** External resources are required to deal with blocking client requests. These resources can be for computation, data storage, etc.

1.5The Workflow of Node.js Architecture:

A web server built with Node.js generally follows a workflow similar to the diagram shown below. Let's examine this sequence of operations in detail



1. Clients send requests to the webserver to interact with the web application. Requests can be non-blocking or blocking (for example, Querying the data, updating the data or deleting the data).
2. Node.js retrieves the incoming requests and adds those requests to the Event Queue.
3. The requests are then passed one-by-one through the Event Loop. It checks if the requests are simple enough to not require any external resources.
4. Event Loop processes simple requests (non-blocking operations), such as I/O Polling, and returns the responses to the corresponding clients.

A single thread from the Thread Pool is assigned to a single complex request. This thread is responsible for completing a particular blocking request by accessing the external resources, such as compute, database, file system, etc.

Once, the task is carried out completely, the response is sent to the Event Loop that in turn sends that response back to the Client.

1.6Features of Node.js

Event-Driven

Node.js employs a non-blocking, event-driven architecture that centers around events, allowing actions to be triggered and handled asynchronously. This design makes it lightweight and efficient for real-time applications. Operations like file handling, network requests, and database interactions generate events, which execute corresponding **callback functions** when triggered. This non-blocking I/O model enables Node.js to manage multiple requests concurrently. For instance, when a user sends an HTTP request to a Node.js server, the receipt of that request activates the appropriate handling function.

Asynchronous and non-blocking:

Compared to traditional server-side technologies (e.g., PHP or Ruby), Node.js is non-blocking and asynchronous. While PHP is blocking and can handle one request at a time, Node.js can handle multiple requests simultaneously, resulting in better scalability.

Node.js conserves resources by enabling other processes to run during input/output operations. This is achieved through its **non-blocking I/O operations**. For example, Node.js doesn't wait for operations like database queries or file reads to complete before handling other requests. Another example is when reading a file where Node.js starts the operation, moves to the next task, and then comes back to handle the result once the file is read.

The **event loop** is the core mechanism in Node.js that allows it to handle concurrent asynchronous operations. The event loop works with a thread pool to manage non-blocking I/O operations, improving performance without interrupting other processes. Instead of creating multiple threads, Node.js processes tasks in a loop, delegating I/O tasks to the system and running callbacks when the I/O operation completes.

How It Works:

- Node.js starts an I/O operation (e.g., reading a file or querying a database).
- While waiting for the result, Node.js processes other requests or events.
- When the I/O operation is complete, the callback function associated with it is executed. This prevents blocking of the server and increases scalability.
- **Event Emitter**: Node.js implements events using the EventEmitter class. This allows applications to emit and listen for events to perform asynchronous actions.

Single-Threaded but Scalable:

Node.js uses a single-threaded, event-driven model with non-blocking I/O, making it highly scalable and efficient for handling multiple concurrent requests. Unlike traditional multi-threaded servers, Node.js responds asynchronously, enabling it to manage high volumes of requests and making it ideal for I/O-intensive applications like API servers, chat apps, real-time data processing, and online gaming. Node.js also leverages its cluster module for load balancing across CPU cores, further enhancing scalability—an essential quality for modern software applications demanded by companies today.

V8 JavaScript Engine:

Node.js is built on Google's V8 JavaScript engine, providing high performance and speed for executing code. The Node.js runtime environment plays a critical role in how applications execute JavaScript, with the V8 engine at its core. This engine efficiently converts JavaScript into machine code, benefiting from performance improvements driven by Google's Chrome browser.

No Buffering

Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.

1.7 Advantages of Node.js

Cross-Platform:

Node.js is available on **Windows**, **macOS**, and various **Linux** distributions. This makes it highly flexible for deployment across different environments.

Rich Ecosystem of Packages - NPM (Node Package Manager):

Node Package Manager (NPM) is the largest software repository in the world. It provides access to a vast repository of libraries, open-source packages and modules that can be easily integrated into Node.js projects.

Developers can install third-party packages or create their own reusable packages, which can then be shared with the community. This extensive ecosystem allows developers to easily integrate functionalities into their applications, saving time and effort.

Node.js, along with npm libraries, also helps to reduce bugs and decrease the size of the web application with its reusable templates.

Single Programming Language

Another advantage of Node.js is that it allows developers to use JavaScript on both the client and server sides, creating a unified development environment that simplifies the coding process.

Developers familiar with JavaScript from front-end work can use their knowledge to build back-end services, avoiding the need to learn a new language for server-side development. This streamlines the development process, reduces context switching, and promotes code reusability.

High Performance:

Node.js uses the V8 engine, which compiles JavaScript to native machine code, leading to

faster execution compared to interpreted languages. In addition, Google itself invests heavily in its search engine to consistently improve its performance.

Handling I/O Operations Efficiently:

Node.js excels in applications that require frequent I/O operations such as reading/writing to databases, reading files, or interacting with external services (e.g., APIs). Node.js uses asynchronous, non-blocking I/O operations, allowing it to handle multiple requests simultaneously without getting blocked. This results in improved performance and scalability.

Event-Driven and Real-Time Capabilities:

Node.js is especially good for building **real-time applications**, such as chat systems, online gaming platforms or live collaboration tools. Events are processed asynchronously, allowing high concurrency. For example, a chat app where multiple users can send messages without waiting for others' requests to complete.

Its event-driven architecture and WebSocket support enable seamless bidirectional communication between clients and servers.

Community Support

Node.js has a large and active community of developers who contribute to its growth and provide support through forums, documentation, and tutorials.

This community-driven approach ensures continuous improvement and innovation in the Node.js ecosystem.

No need for creating multiple threads

Event Loop handles all requests one-by-one, so there is no need to create multiple threads. Instead, a single thread is sufficient to handle a blocking incoming request.

Requires fewer resources and memory

Node.js server, most of the time, requires fewer resources and memory due to the way it handles the incoming requests. Since the requests are processed one at a time, the overall process becomes less taxing on the memory.

1.8Use Cases of Node.js

Real-Time Applications:

Node.js is perfect for applications needing real-time communication, such as chat apps, online gaming, and collaborative tools. It offers advanced features for multi-user chatbots and applications that handle intensive data and heavy traffic across devices. Node.js stands out for its cross-device compatibility, efficient push notifications, and server-side event loops, making it ideal for instant messaging and real-time applications.

Single-Page Applications (SPAs) development:

Node.js, especially with frameworks like Express.js, is ideal for creating single-page applications (SPAs) that require minimal reloading and communicate efficiently with back-end services through APIs. With its ability to deliver a desktop-like experience in the browser, Node.js is well-suited for building social networking platforms, dynamic websites, and mailing solutions. Its asynchronous data flow capabilities on the backend make Node.js an excellent choice for SPA development.

Streaming Applications:

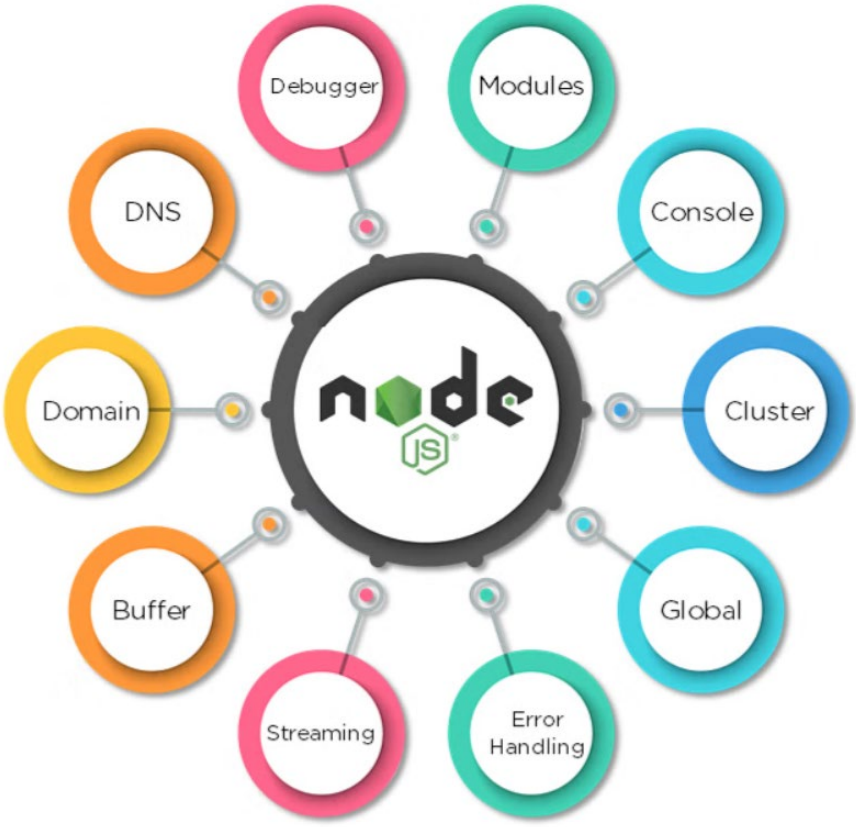
Node.js is well-suited for streaming applications like video and audio platforms due to its event-driven architecture, enabling high-speed file processing and efficient, lightweight encoding. This makes it an ideal choice for data-intensive platforms, including video streaming services like Netflix and various e-commerce and fashion apps.

IoT application development:

Node.js has become a preferred choice for IoT solutions due to its ability to handle high volumes of concurrent requests and events efficiently. Its event-driven, asynchronous architecture is ideal for managing intensive I/O operations and supports readable and writable channels, making it well-suited for IoT app development

1.9Parts of Node.js

The following diagram specifies some important parts of Node.js:



1.10 Examples of some node.js modules

HTTP Module:

The http module allows you to create an HTTP server. This server can handle requests and send responses, making it possible to serve web pages, APIs, and more.

Example:

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

http.createServer((req, res) => {
   res.writeHead(200, { 'Content-Type': 'text/plain' });
   res.end('Hello World\n');
}).listen(3000, () => {
   console.log('Server running at http://localhost:3000/');
});
```

File System (fs) Module:

This module allows interaction with the file system to read, write, or delete files.

Example, Reading a file asynchronously:

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

fs.readFile('example.txt', 'utf8', (err, data) => {
   if (err) throw err;
   console.log(data);
});
```

Path Module:

The path module helps with handling and transforming file paths. It provides utilities to work with file and directory paths in a cross-platform way.

Example:

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

const filePath = path.join(__dirname, 'example.txt');

console.log(filePath);
```

1.11 Creating a Simple Node.js Application

Let’s put together a simple example where Node.js serves an HTML page and reads data from a file:

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

const fs = require('fs');


http.createServer((req, res) => {
   if (req.url === '/') {
      fs.readFile('index.html', (err, data) => {
         if (err) {
            res.writeHead(404, { 'Content-Type': 'text/html' });
            res.write('Page not found');
            res.end();
         } else {
            res.writeHead(200, { 'Content-Type': 'text/html' });
            res.write(data);
            res.end();
         }
      });
   }
}).listen(3000, () => {
   console.log('Server running at http://localhost:3000/');
});
```

In this example:

- The server listens for requests on port 3000.
- When the homepage (/) is requested, it reads an index.html file from the server and sends its contents back as a response.