Getting Started With Node.js

Introduction to Node.js

What is Node.js?

- Node.js is an open-source, cross-platform JavaScript runtime.
- It runs on Chrome's V8 JavaScript engine.
- Node.js enables server-side JavaScript programming.
- Node.js helps create scalable and high-performance web applications.

Note: Node.js is neither a library nor a framework.

Why was Node.js created?

- Node.js was created to overcome the limitations of existing server-side technologies, such as Apache and PHP, which had issues handling multiple simultaneous connections.
- Node.js was a better solution, especially for JavaScript developers.

How was Node.js created?

Node.js is a server-side runtime for JavaScript created by Ryan Dahl in 2009. His goal was to develop a fast and efficient platform that could handle the demands of modern web applications. He used Google's V8 JavaScript engine for its high performance and paired it with an event-driven, non-blocking I/O model to create a potent and effective server-side runtime for JavaScript.

Advantages of using an event-driven, non-blocking I/O model

The main advantage of using an event-driven, non-blocking I/O model is that it allows for better performance and resource utilization, especially when handling multiple connections simultaneously.

Why Is Node.js Popular?

- High-Performance
- Role of javascript
- Lightweight
- Works well with data-intensive applications

Runtimes

A runtime is an environment that allows a programming language to execute code. It provides the necessary resources and tools for a language to interact with the operating system and hardware. Regarding JavaScript, the browser serves as the runtime environment, and NodeJS works as runtime on the server.

What do Runtimes do?

- Complies or Interprets
- Memory Management
- Handles Input/Output Operations
- Garbage Collection

Different Browser Runtimes for JavaScript

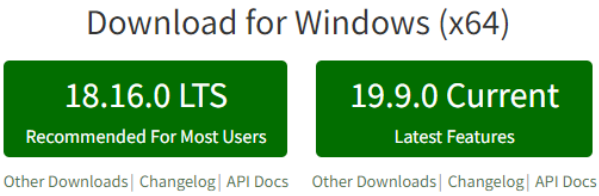
There are several browser runtimes (Javascript engine) for JavaScript, such as V8 for Google Chrome, SpiderMonkey for Mozilla Firefox, Chakra for Microsoft Edge, and JavaScriptCore for Apple Safari. Each of these engines has its way of implementing and optimizing JavaScript.

Setting up Node

Steps to Install Node.js

1. Go to nodejs.org, the official Node.js website. Link

2. Choose between the LTS (Long Term Support) or the Current version of Node.js.



3. It is recommended to select the LTS version for better stability, especially for beginners.
4. Download the LTS installer that matches your operating system.
5. Follow the installation steps.
6. After the installation process, Node.js will be installed and ready to use.
7. To check the Node.js installation, open the terminal/command prompt.
8. Type `node -v` and press enter.
9. The installed version will be displayed.

Creating and Running the "Hello World" Node.js Program

1. Open a text editor and create a new file named "hello-world.js"
2. Type the following code into the file:

```
console.log("Hello, World!");
```

3. Save the file with the .js extension.
4. Open the terminal/command prompt and navigate to the directory where the file is saved.
5. Type `node hello-world.js` and press enter.
6. The program will execute, and the output "Hello, World!" will be displayed in the terminal/command prompt.

Blocking and Non-Blocking Code

Blocking code, or synchronous code, is code that stops the execution of your program until a task is completed. This can cause your application to become

unresponsive, especially when dealing with tasks that take significant time, like calculations or loops.

Here's an example of blocking code:

```
console.log('Starting loop...')

for(let i = 0; i < 1000000000; i++){
 //This code illustrates a time-consuming task and does not perform any //
practical action.
}
console.log('Finished loop.')
```

The code is blocking because it contains a long-running synchronous task.

The program cannot perform other tasks during the execution of the loop, which may cause it to appear frozen or unresponsive to the user.

Non-Blocking or asynchronous code enables a program to continue execution while waiting for a task to complete. This is done using callbacks, promises, or async/await syntax in JavaScript.

Here's an example of non-blocking code:

```
console.log('Starting timer')

setTimeout(() => {
    console.log('Timer finished.')
}, 5000)
console.log('Finished timer.')
```

In this example, the setTimeout function doesn't block the execution. Instead, it takes a callback function that gets executed after a specified delay.

How does NodeJS work?

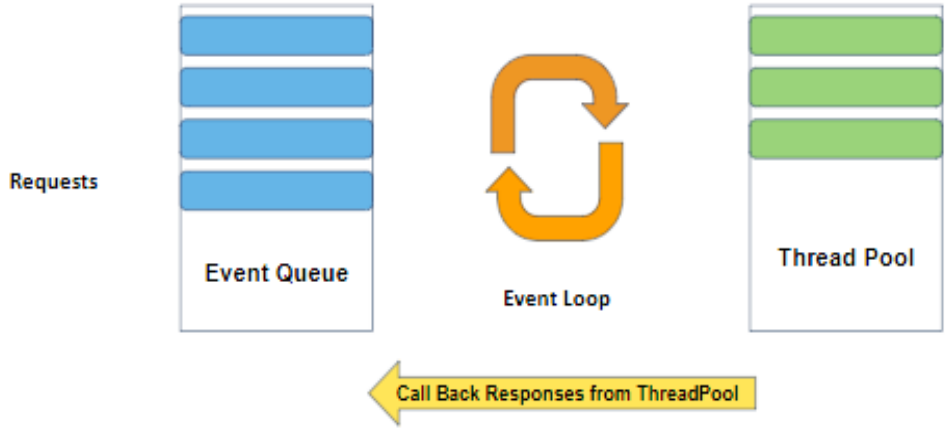
Event Loop

The **Event Loop** is a crucial component of Node.js, responsible for enabling non-blocking I/O operations in a single-threaded architecture. Essentially, it is a continuous loop that checks for pending tasks and executes them one by one in the order they were added to the queue. This process continues until there are no more tasks left in the queue. By allowing Node.js to perform I/O operations asynchronously, the Event Loop helps to ensure that the application remains responsive even when handling a large number of requests.

I/O Operations

Refer to the tasks that involve reading or writing data to external resources like files, databases, or network connections. As these operations are generally time-consuming, Node.js manages them asynchronously to prevent the main thread from being blocked.

Handling I/O Operations



- Node.js uses a combination of the Event Loop, **worker threads**, and callback functions to handle I/O operations.
- When an I/O operation is initiated, Node.js sends the task to a worker thread, which is separate from the main thread.

- The worker thread handles the I/O operation in the background, allowing the main thread to continue executing other tasks.
- Once the I/O operation is complete, the worker thread adds a callback function associated with the operation to the Event queue.
- The Event Loop executes the callback function when it becomes available, allowing Node.js to handle the result of the I/O operation asynchronously.

Performance:

Pros

- Node.js is excellent for I/O-bound operations.
- Non-blocking, the event-driven architecture enables it to handle many simultaneous connections.

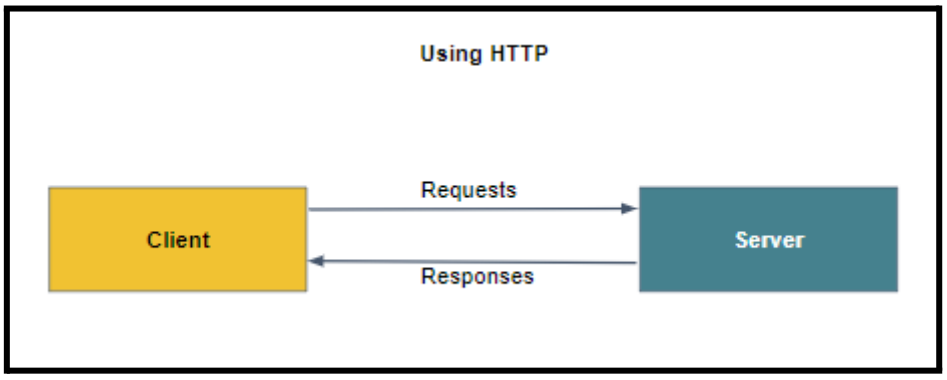
Cons

- It may struggle with CPU-bound tasks.
- Single-threaded execution of JavaScript can cause decreased performance for complex calculations and data processing.

What is a Server?

How Web Applications Work?

Web applications follow a client-server architecture, where the client sends requests to the server. The server processes the request and sends a response back to the client. This communication between the client and server happens via HTTP requests and responses.



Server:

A server is a computer or software that provides resources or services to other computers over a network. In web applications, servers store and process data, handle user authentication, and execute server-side code. Servers are responsible for serving static files to the browser for rendering web pages, as well as receiving and processing user input to enable dynamic updates on the page.

Creating an HTTP Server

What is HTTP?

HTTP stands for Hypertext Transfer Protocol. It is a communication protocol used to transmit and receive data over the Internet. HTTP allows clients (such as web browsers) to send requests to servers, and servers respond with the requested data.

Creating a Server

To create a server in Node.js, use the built-in **'http' module**.

1. Start by importing it:

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

2. Now, create a simple server using the **'http.createServer()'** method:

The createServer method creates an HTTP server. It takes a callback function as an argument that will be called every time a request is made to the server. The callback function has two arguments: req, which represents the incoming HTTP request, and res, the HTTP response object we can use to send data back to the client.

```
const server = http.createServer((req, res){
    res.end('Hello World!')
});
```

3. Make the server listen on a specific port:

```
const PORT = 3000;
server.listen(PORT)
console.log(`Server is listening at http://localhost:${PORT}`)
```

With this code, we've created a basic server that listens on port 3000 and sends back a "Hello World!" message for every request.

Understanding Ports

A port is a unique address that identifies a process or service. Each application has a unique port number assigned to it. When running multiple servers on a single computer, using a different port number for each server allows the client to know which server to communicate with.

Returning HTML as Response

To return an HTML file, we need to read it using the **fs** module and then send its content as a response. Let's create a simple example to demonstrate this.

1. First, create an HTML file named index.html with some primary content:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>My Portfoli</title>
</head>
<body>
    <h1>Welcome to My Portfolio!</h1>
    <p>This is a simple example of serving an HTML file using Node.js</p>
</body>
</html>
```

2. Now, modify the server that serves this file:

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

const server = http.createServer((req, res){
    const data = fs.readFileSync('index.html')
    res.end(data)
})

server.listen(3100)

console.log('Server is listening on 3100')
```

In this example, we are using the **fs.readFileSync** method to read the index.html file, and it can read files and provide content to you.

Summarising it

Let's summarise what we have learned in this module:

- Introduction to Node.js.
- Understanding the runtime and how it works
- Advantages of using event-driven non-blocking I/O model
- Installation of Node.js
- Differences between blocking and non-blocking code
- Role of the Event Loop, worker threads, and callback functions in handling I/O operations
- Communication between web applications and servers
- Creating a server using the http module
- Sending text and HTML as a response to an HTTP request

Some Additional Resources:

- A Complete Visual Guide to Understanding the Node.js Event Loop
- How To Create a Web Server in Node.js with the HTTP Module