

Lecture 4.

Node.js.

JSON/XML

Express Framework

Differences Node.js /
JavaScript

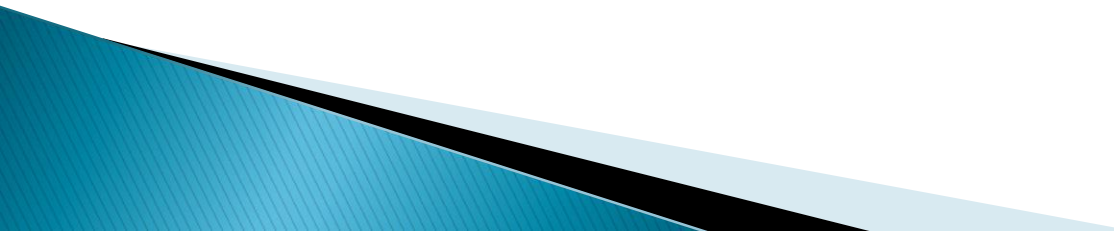
Introducing Node

NODE (or more formally *Node.js*) is an open-source, cross-platform runtime environment that allows developers to create all kinds of **server-side tools and applications in JavaScript**.


The runtime is intended for use outside of a browser context (i.e. running directly on a computer or server OS). As such, the environment omits browser-specific JavaScript APIs and adds support for more traditional OS APIs including HTTP and file system libraries.

Learning Node.js will allow you to write your front-end code and your back-end code in the same language. **Using JavaScript throughout your entire stack can help reduce time for context switching, and libraries are more easily shared between your back-end server and front-end projects.**

Real-time applications, like video streaming, or applications that continuously send and receive data, can run more efficiently when written in Node.js.



From a web server development perspective Node has a number of benefits:

- ▶ **Great performance.** Node was designed to optimize throughput and scalability in web applications and is a good solution for many common web-development problems (e.g. real-time web applications).
 - ▶ **Code is written in "plain old JavaScript",** which means that less time is spent dealing with "context shift" between languages when you're writing both client-side and server-side code.
 - ▶ JavaScript is a relatively new programming language and benefits from improvements in language design when compared to other traditional web-server languages (e.g. Python, PHP, etc.)
 - ▶ **The node package manager (NPM) provides access to hundreds of thousands of reusable packages.** It also has best-in-class dependency resolution and can also be used to automate most of the build toolchain.
 - ▶ **Node.js is portable.** It is available on Microsoft Windows, macOS, Linux, Solaris, FreeBSD, OpenBSD, WebOS, and NonStop OS. Furthermore, it is well-supported by many web hosting providers, that often provide specific infrastructure and documentation for hosting Node sites.
 - ▶ Using Node.js make it possible to create a simple web server using the Node HTTP package.
- 

More about Node.js

Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser. (It's the thing that takes our JavaScript and executes it while browsing with Chrome).

V8 provides the runtime environment in which JavaScript executes. The DOM, and the other Web Platform APIs are provided by the browser.

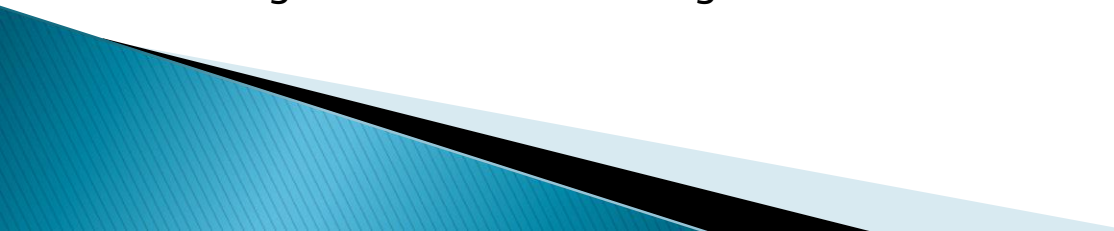
JavaScript engine is independent of the browser in which it's hosted.

A Node.js app runs in a single process, without creating a new thread for every request.

Node.js provides a set of asynchronous I/O primitives in its standard library that prevent JavaScript code from blocking and generally, libraries in Node.js are written using non-blocking paradigms, making blocking behavior the exception rather than the norm.

When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.

This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

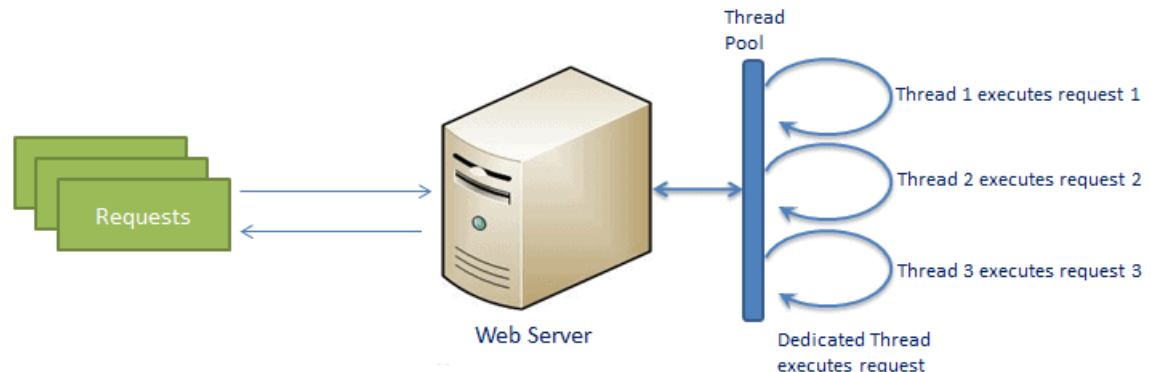


Traditional Web Server Model

In the traditional web server model, each request is handled by a dedicated thread from the thread pool.

If no thread is available in the thread pool at any point of time then the request waits till the next available thread.

Dedicated thread executes a particular request and does not return to thread pool until it completes the execution and returns a response.



Node.js Process Model

Node.js processes user requests differently when compared to a traditional web server model.

Node.js runs in a single process and the application code runs in a single thread and thereby needs less resources than other platforms.

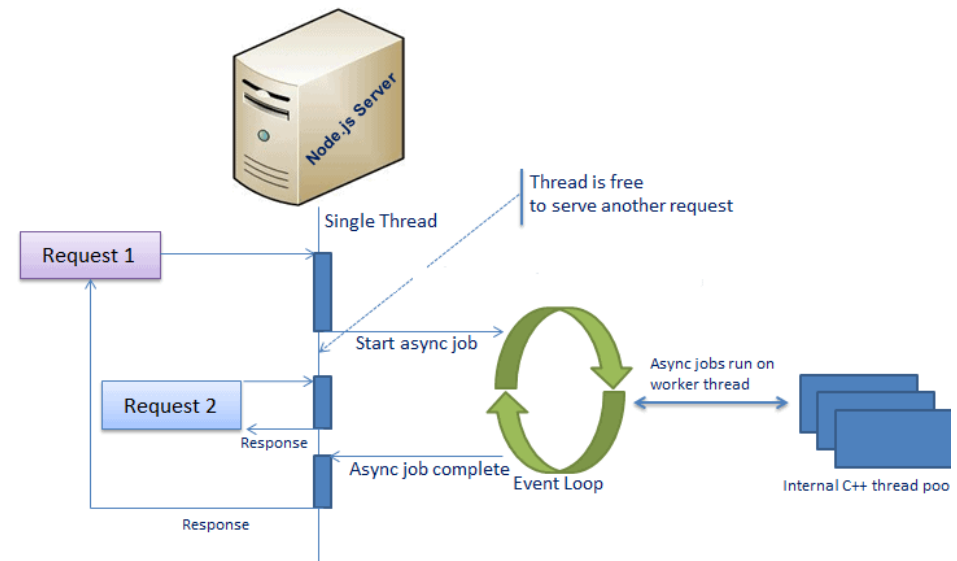
All the user requests to your web application will be handled by a single thread and all the I/O work or long running job is performed asynchronously for a particular request.

This single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response.

An event loop is constantly watching for the events to be raised for an asynchronous job and executing callback function when the job completes. Internally, Node.js uses `libev` for the event loop which in turn uses internal C++ thread pool to provide asynchronous I/O.

Node.js process model increases the performance and scalability with a few caveats.

Node.js is not fit for an application which performs CPU-intensive operations like image processing or other heavy computation work because it takes time to process a request and thereby blocks the single thread.



Install Node.js

Node.js development environment can be setup in Windows, Mac, Linux and Solaris.

The following tools/SDK are required for developing a Node.js application on any platform.

- ▶ Node.js
- ▶ Node Package Manager (NPM)
- ▶ IDE (Integrated Development Environment) or TextEditor

NPM (Node Package Manager) is included in Node.js installation since Node version 0.6.0., so there is no need to install it separately.



Install Node.js on Windows


- ▶ Visit Node.js official web site <https://nodejs.org>.
 - ▶ It will automatically detect OS and display download link as per your Operating System.
 - ▶ Download the installer for windows by clicking on LTS or Current version button. Here, install the latest version LTS for windows that has long time support.
 - ▶ Download the MSI, double-click on it to start the installation as shown below.
 - ▶ Click Next to read and accept the License Agreement and then click Install. It will install Node.js quickly on your computer.
 - ▶ Click finish to complete the installation.
- 

Install Node.js on Mac/Linux

- ▶ Visit Node.js official web site
- ▶ Click on the appropriate installer for Mac (.pkg or .tar.gz) or Linux to download the Node.js installer.
- ▶ Once downloaded, click on the installer to start the Node.js installation wizard.
- ▶ Click on **Continue** and follow the steps. After successful installation, it will display summary of installation about the location where it installed Node.js and NPM.
- ▶ After installation, verify the Node.js installation using terminal window and enter the following command. It will display the version number of Node.js installed on your Mac.

The components of a Node.js application

A Node.js application consists of the following three important components :

- ▶ **Import required modules** – the `require` directive to load Node.js modules.
 - ▶ **Create server** – A server which will listen to client's requests similar to Apache HTTP Server.
 - ▶ **Read request and return response** – The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response.
- 

Creating Node.js Application

Step 1 – Import Required Module

Use the **require** directive to load the http module and store the returned HTTP instance into an http variable as follows –

```
var http = require("http");
```

Step 2 – Create Server

Use the created http instance and call **http.createServer()** method to create a server instance and then we bind it at port 8081 using the **listen** method associated with the server instance.

Pass it a function with parameters request and response. Write the sample implementation to always return "Hello World".

Step 3 – Testing Request & Response

Now execute the main.js to start the server as follows –

```
$ node main.js
```

Verify the Output. Server has started.

Server running at http://127.0.0.1:8081/

```
http.createServer(function (request, response) {  
  // Send the HTTP header  
  // HTTP Status: 200 : OK  
  // Content Type: text/plain  
  response.writeHead(200, {'Content-Type': 'text/plain'});  
  
  // Send the response body as "Hello World"  
  response.end('Hello World\n');  
}).listen(8081);  
  
// Console will print the message  
console.log('Server running at http://127.0.0.1:8081/');
```



Outputting to the Console

To write a “Hello, World!” program, open up a command line text editor such as `nano` and create a new file:

```
nano hello.js
```

With the text editor opened, enter the following code:

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

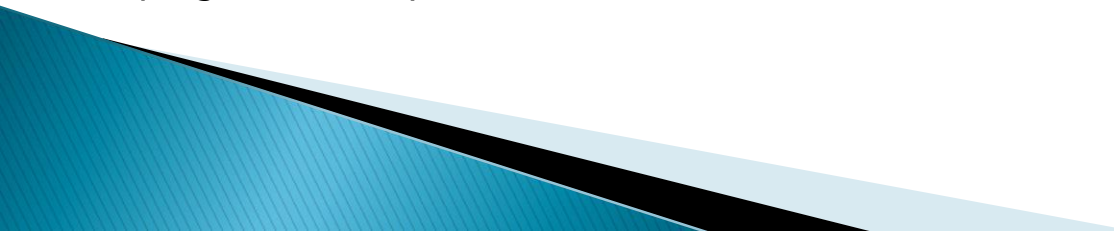
The console object in Node.js provides simple methods to write to `stdout`, `stderr`, or to any other `Node.js` stream, which in most cases is the command line.

The log method prints to the `stdout` stream, so you can see it in your console.

In the context of Node.js, *streams* are objects that can either receive data, like the `stdout` stream, or objects that can output data, like a network socket or a file.

In the case of the `stdout` and `stderr` streams, any data sent to them will then be shown in the console. One of the great things about streams is that they’re easily redirected, in which case you can redirect the output of your program to a file, for example.

Save and exit `nano` by pressing `CTRL+X`, when prompted to save the file, press `Y`. Now your program is ready to run



Running the Program

To run this program, use the node command as follows:

```
node hello.js
```

The hello.js program will execute and display the following output:

```
Hello World
```

The `Node.js` interpreter read the file and executed `console.log("Hello World");` by calling the log method of the global console object.

The string "Hello World" was passed as an argument to the log function.

Although quotation marks are necessary in the code to indicate that the text is a string, they are not printed to the screen.

Having confirmed that the program works.

https://www.digitalocean.com/community/tutorial_series/how-to-code-in-node-js



Node.js Console/REPL

The Node.js Read-Eval-Print-Loop (REPL) is an interactive shell that processes Node.js expressions. The shell reads JavaScript code the user enters, evaluates the result of interpreting the line of code, prints the result to the user, and loops until the user signals to quit. The REPL is bundled with every Node.js installation and allows you to quickly test and explore JavaScript code within the Node environment without having to store it in a file.

It performs the following tasks –

- ▶ **Read** – Reads user's input, parses the input into JavaScript data-structure, and stores in memory.
- ▶ **Eval** – Takes and evaluates the data structure.
- ▶ **Print** – Prints the result.
- ▶ **Loop** – Loops the above command until the user presses **ctrl-c** twice.

To launch the REPL (Node shell), open command prompt (in Windows) or terminal (in Mac or UNIX/Linux) and type *node* as shown below. It will change the prompt to **>** in Windows and MAC.

- ▶ You can test any Node.js/JavaScript expression in REPL. It will display it immediately in new line.
- ▶ The **+** operator also concatenates strings as in browser's JavaScript
- ▶ If you need to write multi line JavaScript expression or function then just press **Enter** whenever you want to write something in the next line as a continuation of your code. The REPL terminal will display three dots (...), it means you can continue on next line. Write **.break** to get out of continuity mode.
- ▶ You can execute an external JavaScript file by executing the **node fileName** command. For example, the following runs **mynodejs-app.js** on the command prompt/terminal and displays the result.
- ▶ To exit from the REPL terminal, press **Ctrl + C** twice or write **.exit** and press Enter.

You can execute any Node.js/JavaScript code in the node shell (REPL). This will give you a result which is similar to the one you will get in the console of Google Chrome browser.

Starting and Stopping the REPL

If you have node installed, then you also have the Node.js REPL. To start it, simply enter node in your command line shell:

```
$ node
```

This results in the REPL prompt: 1

The > symbol lets you know that you can enter JavaScript code to be immediately evaluated.

For an example, try adding two numbers in the REPL by typing this:

```
> 2 + 2
```

When you press ENTER, the REPL will evaluate the expression and return:4

To exit the REPL, you can type .exit, or press CTRL+D once, or press CTRL+C twice, which will return you to the shell prompt.

Using REPL Commands

The REPL has specific keywords to help control its behavior. Each command begins with a dot .

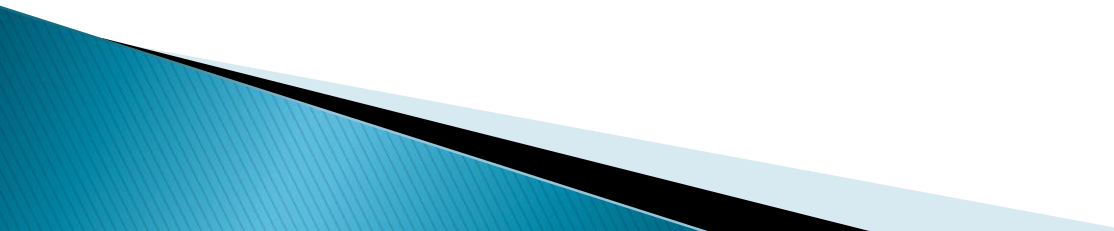
Example: .help

More:

- .break** – Sometimes you get stuck, this gets you out
- .clear** – Alias for **.break**
- .editor** – Enter editor mode
- .exit** – Exit the repl
- .help** – Print this help message
- .load** – Load JS from a file into the REPL session
- .save** – Save all evaluated commands in this REPL session to a file

Press **^C** to abort current expression, **^D** to exit the repl

The REPL is an interactive environment that allows you to execute JavaScript code without first having to write it to a file.



REPL Command	Description
.help	Display help on all the commands
tab Keys	Display the list of all commands.
Up/Down Keys	See previous commands applied in REPL.
.save filename	Save current Node REPL session to a file.
.load filename	Load the specified file in the current Node REPL session.
ctrl + c	Terminate the current command.
ctrl + c (twice)	Exit from the REPL.
ctrl + d	Exit from the REPL.
.break	Exit from multiline expression.
.clear	Exit from multiline expression.

Node.js Module

Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.

Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope. Also, each module can be placed in a separate .js file under a separate folder.

Node.js implements CommonJS modules standard.

CommonJS is a group of volunteers who define JavaScript standards for web server, desktop, and console application.

Node.js Module Types

Node.js includes three types of modules:

- ▶ Core Modules
- ▶ Local Modules
- ▶ Third Party Modules

Node.js Core Modules

Node.js is a light weight framework. The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts.

However, you need to import the core module first in order to use it in your application.

The following table lists some of the important core modules in Node.js

Core Module	Description
http	http module includes classes, methods and events to create Node.js http server.
url	url module includes methods for URL resolution and parsing.
querystring	querystring module includes methods to deal with query string.
path	path module includes methods to deal with file paths.
fs	fs module includes classes, methods, and events to work with file I/O.
util	util module includes utility functions useful for programmers.

Loading Core Modules

In order to use Node.js core or NPM modules, you first need to import it using `require()` function :

```
var module = require('module_name');
```

As per above syntax, specify the module name in the `require()` function.

The `require()` function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.

Using Node.js `http` module to create a web server:

Example: Load and Use Core `http` Module

```
var http = require('http');  
var server = http.createServer(function(req, res){  
  //write code here  
});  
server.listen(5000);
```

In the above example, `require()` function returns an object because `http` module returns its functionality as an object, you can then use its properties and methods using dot notation e.g.

```
http.createServer().
```

In this way, you can load and use Node.js core modules in your application.



Node.js Local Module

Local modules are modules created locally in your Node.js application. These modules include different functionalities of your application in separate files and folders. You can also package it and distribute it via NPM, so that Node.js community can use it.

In Node.js, module should be placed in a separate JavaScript file. Create a Log.js file and write the following code in it.

```
var log = {  
  info: function (info) {  
    console.log('Info: ' + info);  
  },  
  warning: function (warning) {  
    console.log('Warning: ' + warning);  
  },  
  error: function (error) {  
    console.log('Error: ' + error); }  
};  
module.exports = log
```

There are three functions – info(), warning() and error().

At the end, was assigned this object to **module.exports**. The module.exports exposes a log object as a module.

The *module.exports* is a special object which is included in every JS file in the Node.js application by default. Use **module.exports** or **exports** to expose a function, object or variable as a module in Node.js.

Loading Local Module

To use local modules in the application, you need to load it using `require()` function in the same way as core module.

The following example demonstrates how to use the above logging module.

`app.js`

```
var myLogModule = require('./Log.js');  
myLogModule.info('Node.js started');
```

`app.js` is using `log` module. First, it loads the logging module using `require()` function and specified path where logging module is stored.

Logging module is contained in `Log.js` file in the root folder. So, we have specified the path `./Log.js` in the `require()` function. The `.'` denotes a root folder.

The `require()` function returns a `log` object because logging module exposes an object in `Log.js` using `module.exports`.

Now you can use logging module as an object and call any of its function using dot notation e.g `myLogModule.info()` or `myLogModule.warning()` or `myLogModule.error()`

Node.js Web Server

To access web pages of any web application, you need a **web server**. The web server will handle all the http requests for the web application e.g IIS is a web server for ASP.NET web applications and Apache is a web server for PHP or Java web applications.

Node.js provides capabilities to create your own web server which will handle HTTP requests asynchronously. You can use IIS or Apache to run Node.js web application but it is recommended to use Node.js web server.

Create Node.js Web Server

Node.js makes it easy to create a simple web server that processes incoming requests asynchronously.

```
server.js
var http = require('http'); // 1 - Import Node.js core module
var server = http.createServer(function (req, res) { // 2 - creating server

//handle incoming requests here..

});
server.listen(5000); //3 - listen for any incoming requests
console.log('Node.js web server at port 5000 is running..')
```

Import the http module using `require()` function. The http module is a core module of Node.js, so no need to install it using NPM. The next step is to call `createServer()` method of http and specify callback function with request and response parameter. Finally, call `listen()` method of server object which was returned from `createServer()` method with port number.

Handle HTTP Request

The `http.createServer()` method includes request and response parameters which is supplied by Node.js. The request object can be used to get information about the current HTTP request e.g., url, request header, and data. The response object can be used to send a response for a current HTTP request.

The following example demonstrates handling HTTP request and response in Node.js.

```
server.js
var http = require('http'); // Import Node.js core module
var server = http.createServer(function (req, res) { //create web server
    if (req.url == '/') { //check the URL of the current request

        // set response header
        res.writeHead(200, { 'Content-Type': 'text/html' });

        // set response content
        res.write('<html><body><p>This is home Page.</p></body></html>');
        res.end(); }

    else if (req.url == "/student") {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.write('<html><body><p>This is student Page.</p></body></html>');
        res.end(); }

    else if (req.url == "/admin") {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.write('<html><body><p>This is admin Page.</p></body></html>');
        res.end(); }

    else
        res.end('Invalid Request!');

});

server.listen(5000); //6 - listen for any incoming requests
console.log('Node.js web server at port 5000 is running..')
```


Sending JSON Response

```
server.js

var http = require('http');

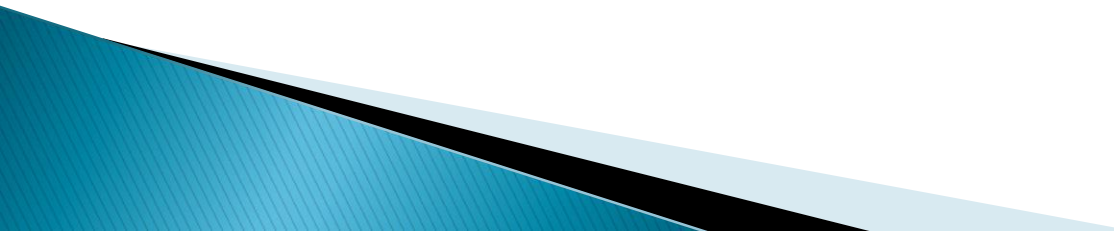
var server = http.createServer(function (req, res) {

    if (req.url == '/data') { //check the URL of the current request
        res.writeHead(200, { 'Content-Type': 'application/json' });
        res.write(JSON.stringify({ message: "Hello World" }));
        res.end();
    }

});

server.listen(5000);

console.log('Node.js web server at port 5000 is running..')
```



JSON

- ▶ JSON stands for **JavaScript Object Notation**
- ▶ JSON is a **text format** for storing and transporting data
- ▶ JSON is "self-describing" and easy to understand

This example is a JSON string:

```
'{"name":"John", "age":30, "car":null}'
```

It defines an object with 3 properties:

- ▶ name
- ▶ age
- ▶ car

Each property has a value.

If you parse the JSON string with a JavaScript program, you can access the data as an object:

```
let personName = obj.name;  
let personAge = obj.age;
```

JSON vs XML

JSON (JavaScript Object Notation) is a lightweight data-interchange format and it completely language independent. It is based on the JavaScript programming language and easy to understand and generate. It is a **data interchange format** and only provides a data encoding specification.

XML (Extensible markup language) was designed to carry data, not to display data. Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. The design goals of XML focus on simplicity, generality, and usability across the Internet. It is a textual data format with strong support via Unicode for different human languages. Although the design of XML focuses on documents, the language is widely used for the representation of arbitrary data structures such as those used in web services.

Both JSON and XML can be used to receive data from a web server.

The following JSON and XML examples both define an employees object, with an array of 3 employ:

JSON Example

```
{ "employees": [
  { "firstName": "John", "lastName": "Doe" },
  { "firstName": "Anna", "lastName": "Smith" },
  { "firstName": "Peter", "lastName": "Jones" }
]}
```

XML Example

```
<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>
```

Similarity between JSON and XML :

- ▶ Both JSON and XML are "self describing" (human readable)
- ▶ Both JSON and XML are hierarchical (values within values)
- ▶ Both JSON and XML can be parsed and used by lots of programming languages
- ▶ Both JSON and XML can be fetched with an XMLHttpRequest

Differences between JSON and XML :

- ▶ JSON doesn't use end tag
- ▶ JSON is shorter
- ▶ JSON is quicker to read and write
- ▶ JSON can use arrays

The biggest difference is: XML has to be parsed with an XML parser. JSON can be parsed by a standard JavaScript function.

JSON's benefits comparing to XML :

- ▶ XML is much more difficult to parse than JSON.
- ▶ JSON is parsed into a ready-to-use JavaScript object.

For AJAX applications, JSON is faster and easier than XML:

Using XML

- ▶ Fetch an XML document
- ▶ Use the XML DOM to loop through the document
- ▶ Extract values and store in variables

Using JSON

- ▶ Fetch a JSON string
- ▶ JSON.Parse the JSON string

JSON	XML
It is JavaScript Object Notation	It is Extensible markup language
It is based on JavaScript language.	It is derived from SGML.
It is a way of representing objects.	It is a markup language and uses tag structure to represent data items.
It does not provides any support for namespaces.	It supports namespaces.
It supports array.	It doesn't supports array.
Its files are very easy to read as compared to XML.	Its documents are comparatively difficult to read and interpret.
It doesn't use end tag.	It has start and end tags.
It is less secured.	It is more secured than JSON.
It doesn't supports comments.	It supports comments.
It supports only UTF-8 encoding.	It supports various encoding.

Creating a Web Server using Node. Step 1.

Node.js provides an **http** module which can be used to create an HTTP client of a server.

Following is the bare minimum structure of the HTTP server which listens at 8081 port.

File: server.js

```
var http = require('http');
var fs = require('fs');
var url = require('url');

// Create a server
http.createServer( function (request, response) {
  // Parse the request containing file name
  var pathname = url.parse(request.url).pathname;

  // Print the name of the file for which request is made.
  console.log("Request for " + pathname + " received.");

  // Read the requested file content from file system
  fs.readFile(pathname.substr(1), function (err, data) {
    if (err) {
      console.log(err);

      // HTTP Status: 404 : NOT FOUND
      // Content Type: text/plain
      response.writeHead(404, {'Content-Type': 'text/html'});
    } else {
      //Page found
      // HTTP Status: 200 : OK
      // Content Type: text/plain
      response.writeHead(200, {'Content-Type': 'text/html'});

      // Write the content of the file to response body
      response.write(data.toString());
    }

    // Send the response body
    response.end();
  });
}).listen(8081);

// Console will print the message
console.log('Server running at http://127.0.0.1:8081/');
```

Step 2

Create html file named **index.html** in the same directory where you created **server.js**.

File: index.htm

```
<html>
  <head>
    <title>Sample Page</title>
  </head>

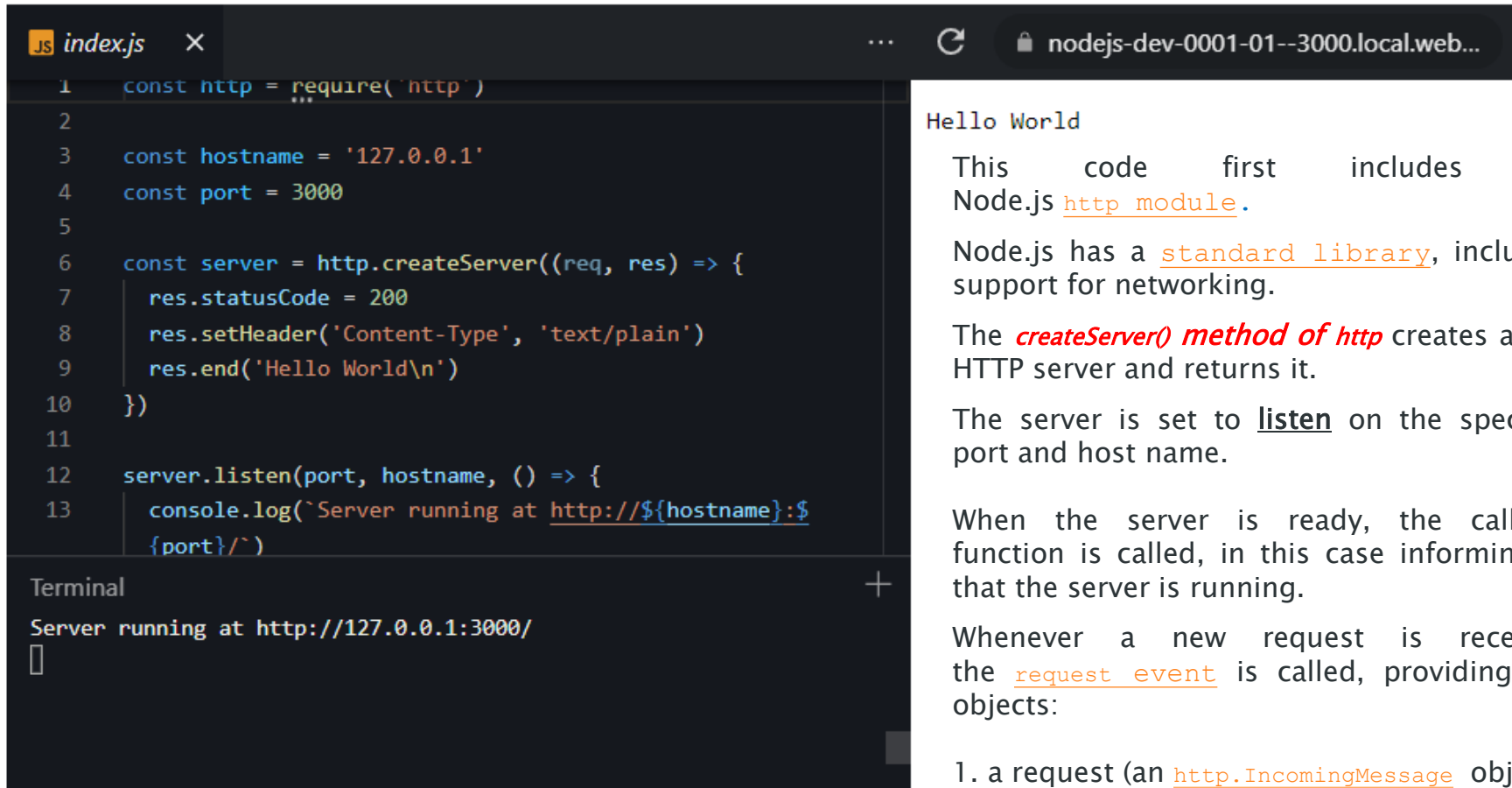
  <body>
    Hello World!
  </body>
</html>
```

- ▶ run the server.js to see the result –
\$ node server.js

Verify the Output.

Server running at <http://127.0.0.1:8081/>

An Example Node.js Application



```
JS index.js x ... nodejs-dev-0001-01--3000.local.web...  
1 const http = require('http')  
2  
3 const hostname = '127.0.0.1'  
4 const port = 3000  
5  
6 const server = http.createServer((req, res) => {  
7   res.statusCode = 200  
8   res.setHeader('Content-Type', 'text/plain')  
9   res.end('Hello World\n')  
10 })  
11  
12 server.listen(port, hostname, () => {  
13   console.log(`Server running at http://${hostname}:${port}/`)  
14 })
```

Hello World

Server running at http://127.0.0.1:3000/

Hello World

This code first includes the Node.js [http module](#).

Node.js has a [standard library](#), including support for networking.

The *[createServer\(\) method of http](#)* creates a new HTTP server and returns it.

The server is set to [listen](#) on the specified port and host name.

When the server is ready, the callback function is called, in this case informing us that the server is running.

Whenever a new request is received, the [request event](#) is called, providing two objects:

1. a request (an [http.IncomingMessage](#) object)
- and
2. a response (an [http.ServerResponse](#) object).

Those 2 objects are essential to handle the HTTP call.

Run Node.js scripts from the command line

The usual way to run a Node.js program is to run the node globally available command (once you install Node.js) and pass the name of the file you want to execute.

If your main Node.js application file is app.js, you can call it by typing:

```
node app.js
```

Above, you are explicitly telling the shell to run your script with node. You can also embed this information into your JavaScript file with a "shebang" line. The "shebang" is the first line in the file, and tells the OS which interpreter to use for running the script. Below is the first line of JavaScript:

```
#!/usr/bin/node
```

Above, we are explicitly giving the absolute path of interpreter. Not all operating systems have node in the bin folder, but all should have env. You can tell the OS to run env with node as parameter:

```
#!/usr/bin/env node  
// your code
```

To use a shebang, your file should have executable permission. You can give app.js the executable permission by running:

```
chmod u+x app.js
```

While running the command, make sure you are in the same directory which contains the app.js file.

Restart the application automatically

The node command has to be re-executed in bash whenever there is a change in the application, to restart the application automatically, nodemon module is used.

Install the `nodemon` module globally to system path

```
npm i -g nodemon
```

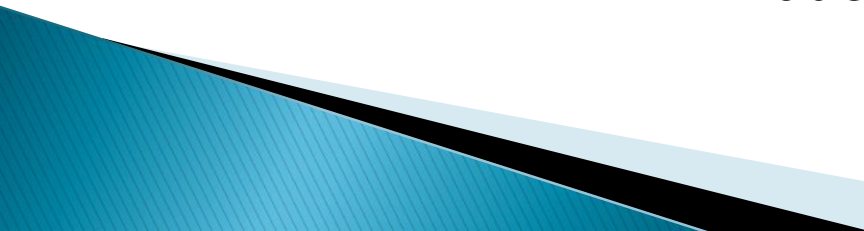
You can also install nodemon as a development-dependency

```
npm i --save-dev nodemon
```

This local installation of nodemon can be run by calling it from within npm script such as `npm start` or using `npx nodemon`.

Run the application using nodemon followed by application file name.

```
nodemon app.js
```



Node.js Frameworks and Tools

The following table lists frameworks for Node.js.

AdonisJS: A TypeScript-based fully featured framework highly focused on developer ergonomics, stability, and confidence. Adonis is one of the fastest Node.js web frameworks.

Egg.js: A framework to build better enterprise frameworks and apps with Node.js & Koa.

Express: It provides one of the most simple yet powerful ways to create a web server. Its minimalist approach, focused on the core features of a server, is key to its success.

Fastify: A web framework highly focused on providing the best developer experience with the least overhead and a powerful plugin architecture. Fastify is one of the fastest Node.js web frameworks.

FeatherJS: Feathers is a lightweight web-framework for creating real-time applications and REST APIs using JavaScript or TypeScript. Build prototypes in minutes and production-ready apps in days.

Gatsby: A React-based, GraphQL powered, static site generator with a very rich ecosystem of plugins and starters.

Hapi: A rich framework for building applications and services that enables developers to focus on writing reusable application logic instead of spending time building infrastructure.

Koa: It is built by the same team behind Express, aims to be even simpler and smaller, building on top of years of knowledge. The new project born out of the need to create incompatible changes without disrupting the existing community.

Loopback.io: Makes it easy to build modern applications that require complex integrations.

Meteor: An incredibly powerful full-stack framework, powering you with an isomorphic approach to building apps with JavaScript, sharing code on the client and the server. Once an off-the-shelf tool that provided everything, now integrates with frontend libs React, Vue, and Angular. Can be used to create mobile apps as well.

Micro: It provides a very lightweight server to create asynchronous HTTP microservices.

NestJS: A TypeScript based progressive Node.js framework for building enterprise-grade efficient, reliable and scalable server-side applications.

Next.js: React framework that gives you the best developer experience with all the features you need for production: hybrid static & server rendering, TypeScript support, smart bundling, route pre-fetching, and more.

Nx: A toolkit for full-stack monorepo development using NestJS, Express, React, Angular, and more. Nx helps scale your development from one team building one application to many teams collaborating on multiple applications!

Remix: Remix is a fullstack web framework for building excellent user experiences for the web. It comes out of the box with everything you need to build modern web applications (both frontend and backend) and deploy them to any JavaScript-based runtime environment (including Node.js).

Sapper: Sapper is a framework for building web applications of all sizes, with a beautiful development experience and flexible filesystem-based routing. Offers SSR and more!

Socket.io: A real-time communication engine to build network applications.

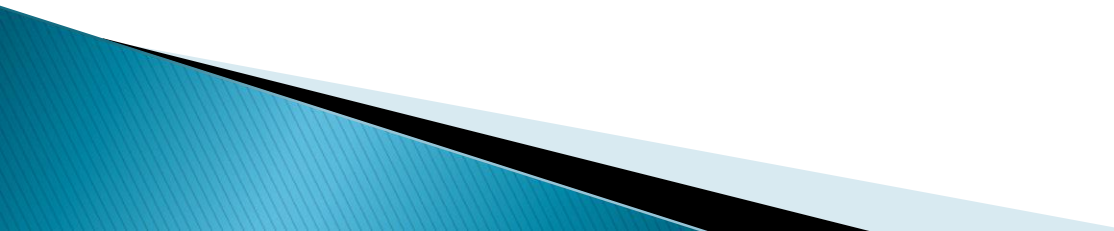
Strapi: Strapi is a flexible, open-source Headless CMS that gives developers the freedom to choose their favorite tools and frameworks while also allowing editors to easily manage and distribute their content. By making the admin panel and API extensible through a plugin system, Strapi enables the world's largest companies to accelerate content delivery while building beautiful digital experiences.

Express Overview

Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications.

It facilitates the rapid development of Node based Web applications.

Following are some of the core features of Express framework –

- ▶ Allows to set up middlewares (**A request handler with access to the application's request-response cycle**) to respond to HTTP Requests.
 - ▶ Defines a routing table which is used to perform different actions based on HTTP Method and URL.
 - ▶ Allows to dynamically render HTML Pages based on passing arguments to templates.
- 

Installing Express

Firstly, install the Express framework globally using NPM so that it can be used to create a web application using node terminal.

```
$ npm install express --save
```

The above command saves the installation locally in the **node_modules** directory and creates a directory **express** inside **node_modules**. You should install the following important modules along with **express** –

- ▶ **body-parser** – This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.
- ▶ **cookie-parser** – Parse Cookie header and populate **req.cookies** with an object keyed by the cookie names.
- ▶ **multer** – This is a node.js middleware for handling multipart/form-data.

```
$ npm install body-parser --save
```

```
$ npm install cookie-parser --save
```

```
$ npm install multer --save
```



Hello world Example

Following is a very basic Express app which starts a server and listens on port 8081 for connection.

This app responds with **Hello World!** for requests to the homepage.

For every other path, it will respond with a **404 Not Found**.

Save the above code in a file named server.js and run it with the following command.

\$ node server.js

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World');
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

Request & Response

Express application uses a callback function whose parameters are **request** and **response** objects.

- ▶ **Request Object** – The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
- ▶ **Response Object** – The response object represents the HTTP response that an Express app sends when it gets an HTTP request.

You can print **req** and **res** objects which provide a lot of information related to HTTP request and response including cookies, sessions, URL, etc.

```
app.get('/', function (req, res) {  
    // --  
})
```


GET Method

Here is a simple example which passes two values using HTML FORM GET method.

We are going to use **process_get** router inside **server.js** to handle this input.

Let's save above code in **index.html** and modify **server.js** to handle home page requests as well as the input sent by the HTML form.

```
<html>
  <body>

    <form action = "http://127.0.0.1:8081/process_get" method = "GET">
      First Name: <input type = "text" name = "first_name"> <br>
      Last Name: <input type = "text" name = "last_name">
      <input type = "submit" value = "Submit">
    </form>

  </body>
</html>
```

```
var express = require('express');
var app = express();

app.use(express.static('public'));
app.get('/index.htm', function (req, res) {
  res.sendFile( __dirname + "/" + "index.htm" );
})

app.get('/process_get', function (req, res) {
  // Prepare output in JSON format
  response = {
    first_name:req.query.first_name,
    last_name:req.query.last_name
  };
  console.log(response);
  res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

Result

Accessing the HTML document using *`http://127.0.0.1:8081/index.htm`* will generate the following form



The image shows a screenshot of a web form. It has a light gray background. On the left side, there are two text labels: "First Name:" and "Last Name:". To the right of "First Name:" is a white rectangular input field. To the right of "Last Name:" is another white rectangular input field. Below these two input fields is a small rectangular button with the word "Submit" written on it.

enter the First and Last Name and then click submit button to see the result and it should return the following result –

```
{"first_name": "Alla", "last_name": "Jammine"}
```

POST Method

Here is a simple example which passes two values using HTML FORM POST method. We are going to use `process_get` router inside `server.js` to handle this input.

Save the above code in `index.html` and modify `server.js` to handle home page requests as well as the input sent by the HTML form.

```
<html>
  <body>

    <form action = "http://127.0.0.1:8081/process_post" method = "POST">
      First Name: <input type = "text" name = "first_name"> <br>
      Last Name: <input type = "text" name = "last_name">
      <input type = "submit" value = "Submit">
    </form>

  </body>
</html>

var express = require('express');
var app = express();
var bodyParser = require('body-parser');

// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })

app.use(express.static('public'));
app.get('/index.htm', function (req, res) {
  res.sendFile( __dirname + "/" + "index.htm" );
})

app.post('/process_post', urlencodedParser, function (req, res) {
  // Prepare output in JSON format
  response = {
    first_name:req.body.first_name,
    last_name:req.body.last_name
  };
  console.log(response);
  res.end(JSON.stringify(response));
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})
```

Result

Accessing the HTML document using *`http://127.0.0.1:8081/index.htm`* will generate the following form



The image shows a screenshot of a web form. It has a light gray background. On the left side, there are two text labels: "First Name:" and "Last Name:". To the right of "First Name:" is a white rectangular input field. To the right of "Last Name:" is another white rectangular input field. Below these two input fields is a small rectangular button with the word "Submit" written on it.

enter the First and Last Name and then click submit button to see the result and it should return the following result –

```
{"first_name":"Alla","last_name":"Jammine"}
```

File Upload

The following HTML code creates a file uploader form.

This form has method attribute set to **POST** and *enctype* attribute is set to **multipart/form-data**

```
<html>
  <head>
    <title>File Uploading Form</title>
  </head>

  <body>
    <h3>File Upload:</h3>
    Select a file to upload: <br />

    <form action = "http://127.0.0.1:8081/file_upload" method = "POST"
      enctype = "multipart/form-data">
      <input type="file" name="file" size="50" />
      <br />
      <input type = "submit" value = "Upload File" />
    </form>

  </body>
</html>
```

```

var express = require('express');
var app = express();
var fs = require("fs");

var bodyParser = require('body-parser');
var multer = require('multer');

app.use(express.static('public'));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(multer({ dest: '/tmp/'}));

app.get('/index.htm', function (req, res) {
  res.sendFile( __dirname + "/" + "index.htm" );
})

app.post('/file_upload', function (req, res) {
  console.log(req.files.file.name);
  console.log(req.files.file.path);
  console.log(req.files.file.type);
  var file = __dirname + "/" + req.files.file.name;

  fs.readFile( req.files.file.path, function (err, data)
    fs.writeFile(file, data, function (err) {
      if( err ) {
        console.log( err );
      } else {
        response = {
          message:'File uploaded successfully',
          filename:req.files.file.name
        };
      }
    });
  });
}

```



Create
Index.html and
modify server.js
to handle home
page requests as
well as file
upload.

```

}

console.log( response );
res.end( JSON.stringify( response ) );
});
});
});

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

  console.log("Example app listening at http://%s:%s", host, port)
})

```

Result

Accessing the HTML document using *<http://127.0.0.1:8081/index.htm>* will generate the following form

File Upload:

Select a file to upload:

Выберите файл Файл не выбран

Upload File

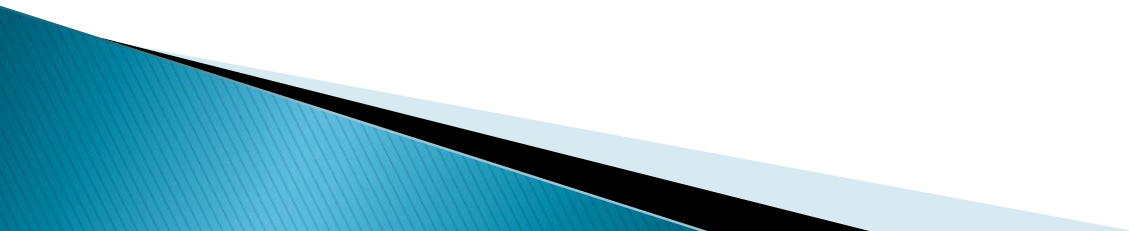
Differences Between Node.js and JavaScript?

- ▶ 1. **JavaScript is a client-side scripting language** that is lightweight, cross-platform, and interpreted. Both Java and HTML include it. **Node.js**, on the other hand, **is a V8-based server-side programming language**. As a result, it is used to create network-centric applications. It's a networked system made for data-intensive real-time applications. If we compare node js vs. python, it is clear that node js will always be the preferred option.⁴
- ▶ 2. **JavaScript is a simple programming language that can be used with any browser that has the JavaScript Engine installed**. **Node.js**, on the other hand, **is an interpreter or execution environment for the JavaScript programming language**. It requires libraries that can be conveniently accessed from JavaScript programming to be more helpful.
- ▶ 3. **Any engine may run JavaScript**. As a result, writing **JavaScript is incredibly easy**, and any working environment is similar to a complete browser. **Node.js**, on the other hand, **only enables the V8 engine**. Written JavaScript code, on the other hand, can run in any context, regardless of whether the V8 engine is supported.
- ▶ 4. A specific non-blocking operation is required to access any operating system. There are a **few essential objects in JavaScript, but they are entirely OS-specific**. **Node.js**, on the other hand, **can now operate non-blocking software tasks out of any JavaScript programming**. **It contains no OS-specific constants**. Node.js establishes a strong relationship with the system files, allowing companies to read and write to the hard drive.
- ▶ 5. The critical benefits of JavaScript include a wide choice of interfaces and interactions and just the proper amount of server contact and direct visitor input. Node.js, on the other hand, offers node package management with over 500 modules and the capacity to handle many requests at the same time. It also offers the unique ability to enable microservice architecture and the Internet of Things. Even when comparing node js vs. react js, node js always wins.

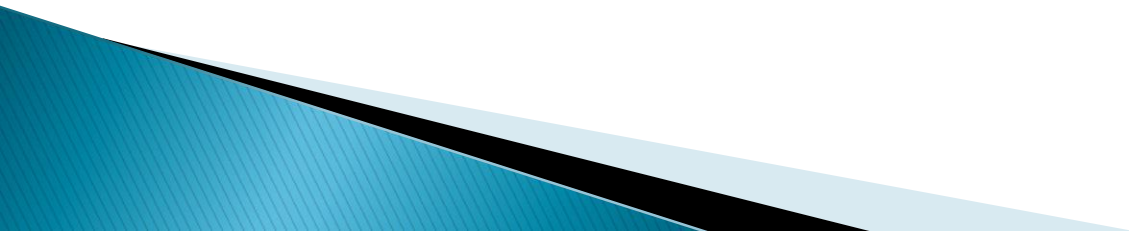
Major Comparison Between Node.js and JavaScript

JavaScript	Node.js
It is an accessible, bridge, parsed, lightweight, reactive, and web apps programming language.	It's a bridge, open-source Js runtime environment for executing Js on the server.
It's a programming language, after all. Any browser with a competent browser engine will operate.	It's a JavaScript translator and environment that includes some valuable libraries for JavaScript programming.
It's most commonly used on client-side servers.	It's mainly popular on the server-side.
The node community does not care about JavaScript.	All node projects represent the JavaScript community.
It's made for creating network-centric apps.	It's made for real-time data-intensive apps that run on multiple platforms.
It's a new release of the ECMA script that works on the C++-based V8 engine.	C++, C, and JavaScript are used.
TypedJS, RamdaJS, and other JavaScript frameworks are examples.	Nodejs modules include Lodash and Express. All of these modules must be imported from npm.

Conclusions / Questions

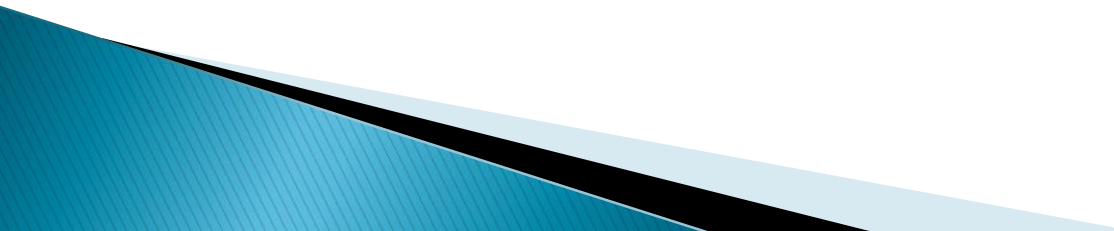


What is NodeJS and why?

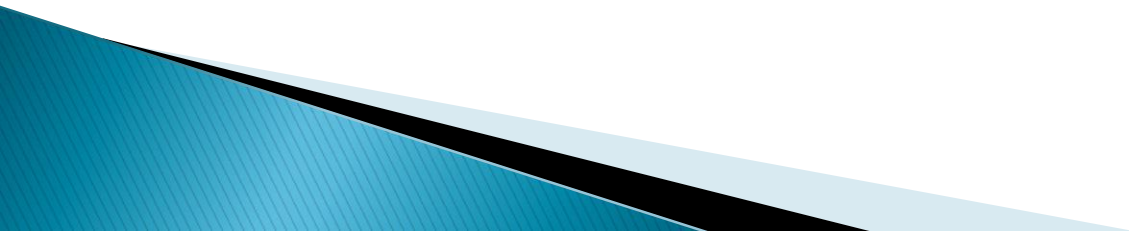


Answer

Node.js is a single-threaded, open-source, cross-platform runtime environment for building fast and scalable server-side and networking applications. It runs on the V8 JavaScript runtime engine, and it uses event-driven, non-blocking I/O architecture, which makes it efficient and suitable for real-time applications.

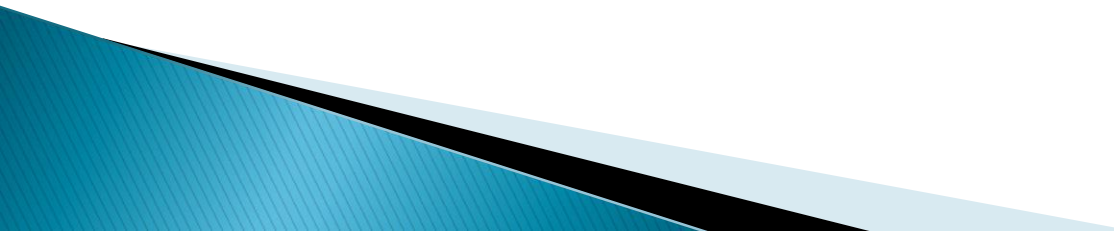


What is NodeJS used for?

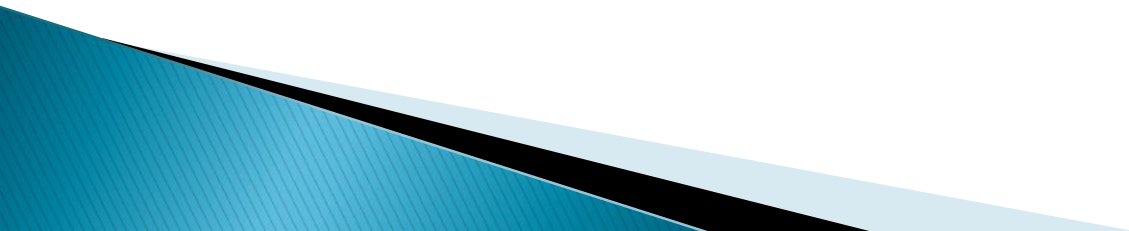


Answer

Node.js is an open-source, cross-platform JavaScript runtime environment and library for **running web applications outside the client's browser**. Ryan Dahl developed it in 2009, and its latest iteration, version 15.14, was released in April 2021. Developers use Node.

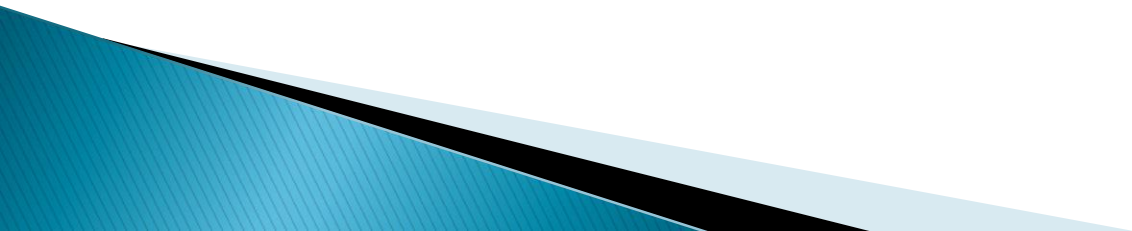


Is node JS frontend or backend?

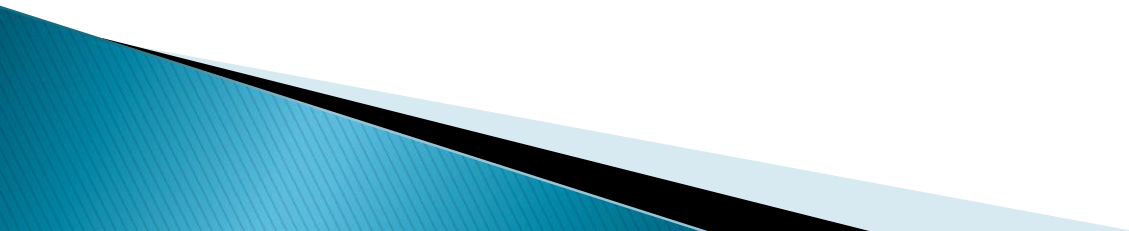


Answer

Node.js is sometimes misunderstood by developers as a backend framework that is exclusively used to construct servers. This is not the case; **Node.js can be used on the frontend as well as the backend.**



Is node js basically JavaScript?



Answer

Node.js is a JavaScript runtime environment for server-side development. Node.js allows you to write JavaScript code that runs on the server instead of in the browser.