Lesson 1: Node JS Introduction and Features

Node.js Introduction:

Node.js is an open-source server side runtime environment built on Chrome's V8 JavaScript engine. It provides an event driven, non-blocking (asynchronous) I/O and cross-platform runtime environment for building highly scalable server-side application using JavaScript.

Node.js can be used to build different types of applications such as command line application, web application, real-time chat application, REST API server etc. However, it is mainly used to build network programs like web servers, similar to PHP, Java, or ASP.NET.

Node.js was written and introduced by Ryan Dahl in 2009.

Node.js applications are written in JavaScript, and can be run within the Node.js runtime on mac OS X, Microsoft Windows, and Linux.

You need to know that NodeJS isn't a framework, and it's not a programing language.

Node.js lets developers use JavaScript to write command line tools and **for server-side scripting**—running scripts server-side to produce **dynamic web page content before the page is sent to the user's web browser.**

Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

Node.js = Runtime Environment + JavaScript Library

Node.js official web site: https://nodejs.org

Node.js on github: https://github.com/nodejs/node

Node.js community conference http://nodeconf.com

Why NODE?

- When you create websites with PHP for example, you associate the language with an HTTP web server such as Apache or Nginx. Each of them has its own role in the process:
- Apache manages HTTP requests to connect to the server. Its role is more or less to manage the in/out traffic.
- PHP runs the .php file code and sends the result to Apache, which then sends it to the visitor.
- As several visitors can request a page from the server at the same time, Apache is responsible for spreading them out and running different threads at the same time.
- Each thread uses a different processor on the server (or a processor core)

- **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.**

A common task for a web server can be to open a file on the server and return the content to the client.

Here is how PHP or ASP handles a file request:

- **Sends the task to the computer's file system.**
- **Waits while the file system opens and reads the file.**
- **Returns the content to the client.**
- **Ready to handle the next request.**

Here is how Node.js handles a file request:

- **Sends the task to the computer's file system.**

- ➢ **Ready to handle the next request.**
- ➢ **When the file system has opened and read the file, the server returns the content to the client.**
- ➢ **Node.js eliminates the waiting, and simply continues with the next request.**

Features of Node.js

Following are some of the important features that make Node.js the first choice of software architects.

- **Asynchronous and Event Driven** − All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call.
- **Very Fast** − Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
- **Single Threaded but Highly Scalable** − Node.js uses a single threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server.
- **No Buffering** − Node.js applications never buffer any data. These applications simply output the data in chunks.
- **License** − Node.js is released under the MIT license

Where to Use Node.js?

Following are the areas where Node.js is proving itself as a perfect technology partner.

- I/O bound Applications
- Data Streaming Applications
- Data Intensive Real-time Applications (DIRT)
- JSON APIs based Applications
- Single Page Applications

Advantages of Node.js

1. Node.js is an open-source framework under MIT license. (MIT license is a free software license originating at the Massachusetts Institute of Technology (MIT).)
2. Uses JavaScript to build entire server side application.
3. Lightweight framework that includes bare minimum modules. Other modules can be included as per the need of an application.
4. Asynchronous by default. So it performs faster than other frameworks.
5. Cross-platform framework that runs on Windows, MAC or Linux

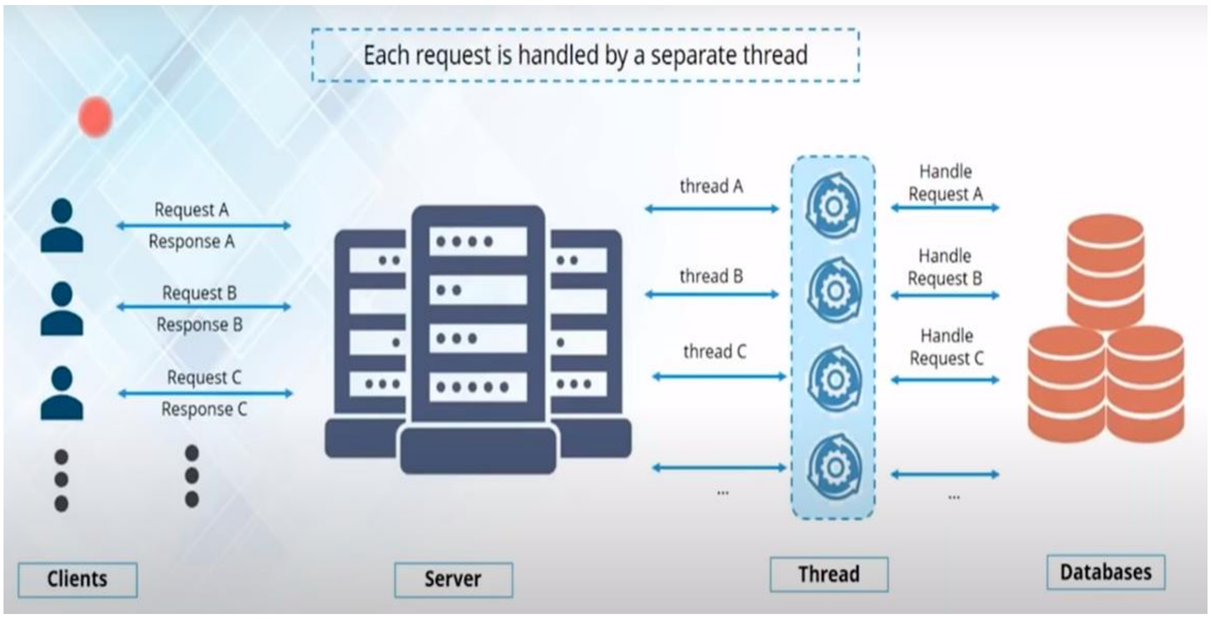
What Can Node.js Do?

- Node.js can generate dynamic page content
- Node.js can create, open, read, write, delete, and close files on the server
- Node.js can collect form data
- Node.js can add, delete, modify data in your database

Node JS Process Model:

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.

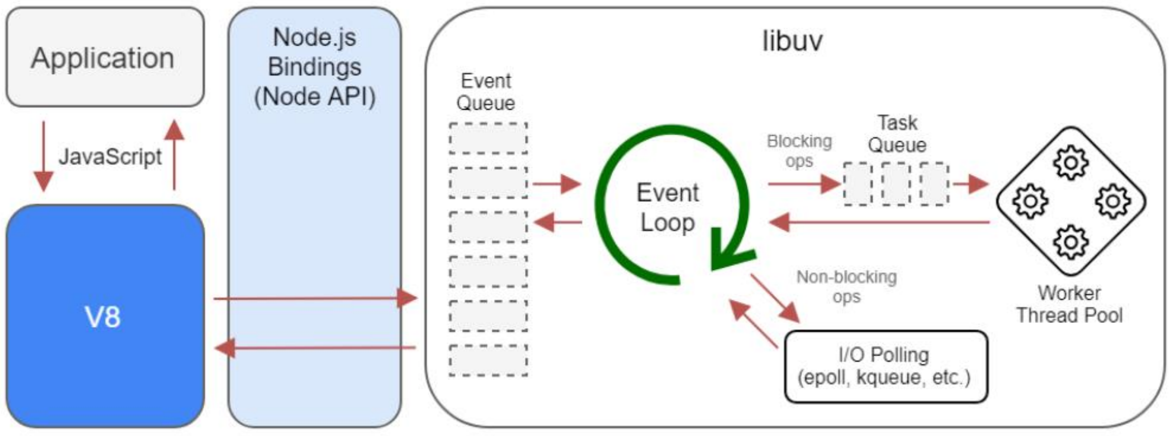


Node.js Process Model

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. So, 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.

The following figure illustrates asynchronous web server model using Node.js.



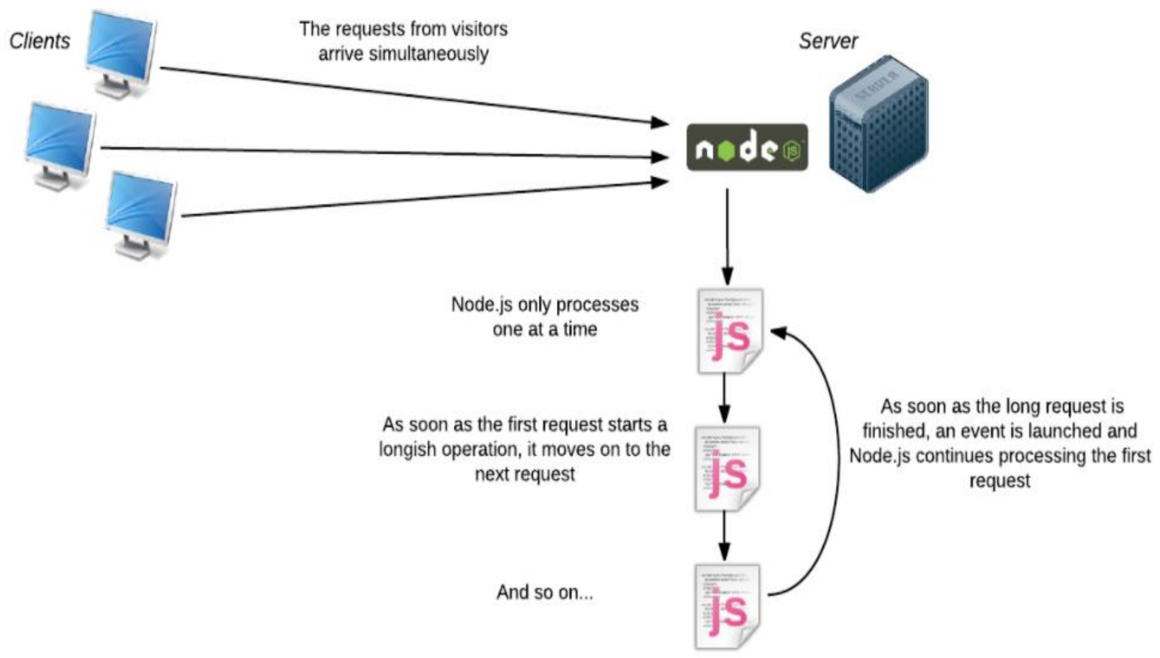
Node.js Process Model

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.

Mono-Thread-Handling multiple Requests?

Node.js doesn't use an HTTP server like Apache. In fact, it's up to us to create the server! Isn't that great?

Unlike Apache, Node.js is monothread. This means that there is only one process and one version of the program that can be used at any one time in its memory.


Clients
The requests from visitors arrive simultaneously
Server
node
Node.js only processes one at a time
As soon as the first request starts a longish operation, it moves on to the next request
As soon as the long request is finished, an event is launched and Node.js continues processing the first request
And so on...

<u>V8 JavaScript Engine</u>

Chrome needs to run some Javascript for a particular Web page, it doesn't run the JavaScript itself. It uses the V8 engine to get that done so it passes the code to V8 and it gets the results back. Same case with Node also to run a Javascript code.

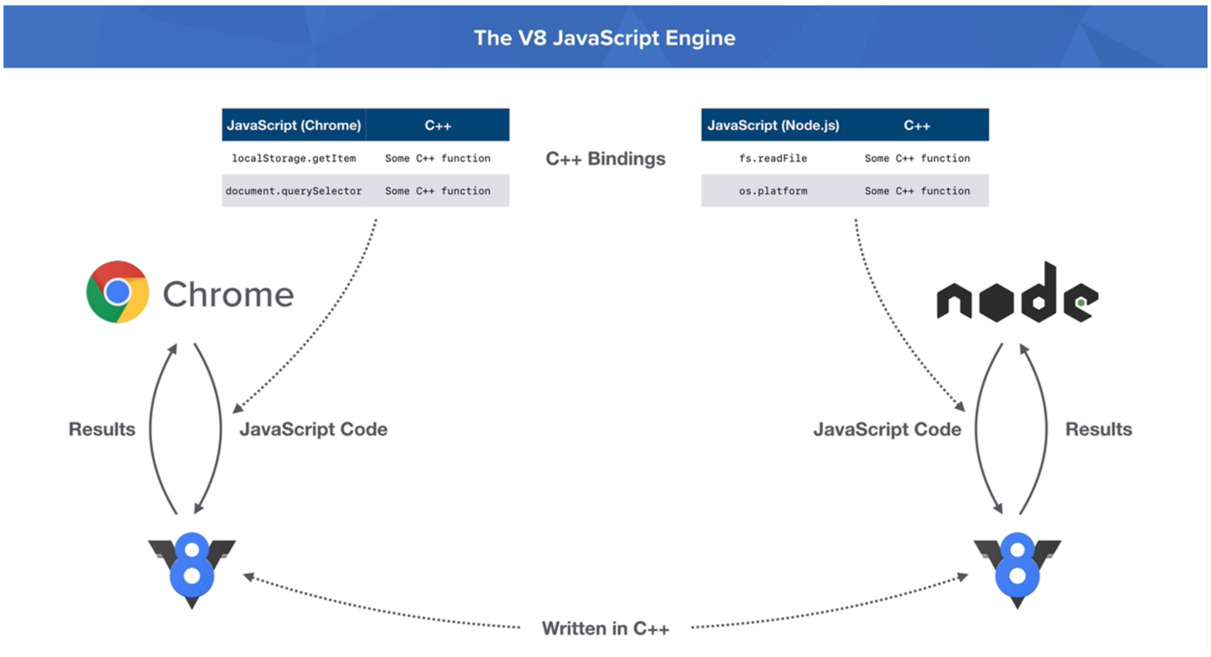
V8 is written in C++ . Chrome and Node are largely written in C++ because they both provide bindings when they're instantiating the V8 engine.

This facilitates to create their own JavaScript runtime with interesting and novel features.

Example Instance:

Chrome to interact with the DOM when the DOM isn't part of JavaScript.

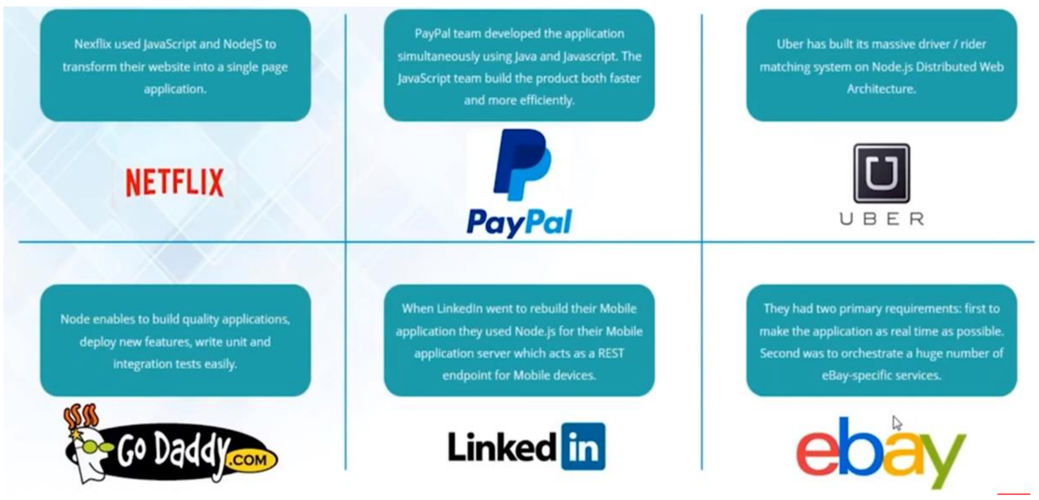
Node to interact with file system when the file system isn't part of JavaScript



JavaScript (Chrome)	C++
localStorage.getItem	Some C++ function
document.querySelector	Some C++ function

JavaScript (Node.js)	C++
fs.readFile	Some C++ function
os.platform	Some C++ function

NodeJS Success Stories:


Nexflix used JavaScript and NodeJS to transform their website into a single page application.

PayPal team developed the application simultaneously using Java and Javascript. The JavaScript team build the product both faster and more efficiently.

Uber has built its massive driver / rider matching system on Node.js Distributed Web Architecture.

Node enables to build quality applications, deploy new features, write unit and integration tests easily.

When LinkedIn went to rebuild their Mobile application they used Node.js for their Mobile application server which acts as a REST endpoint for Mobile devices.

They had two primary requirements: first to make the application as real time as possible. Second was to orchestrate a huge number of eBay-specific services.

To know more success stories, look into the below link

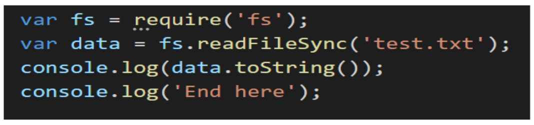
https://thinkmobiles.com/blog/node-js-app-examples/

Callbacks:

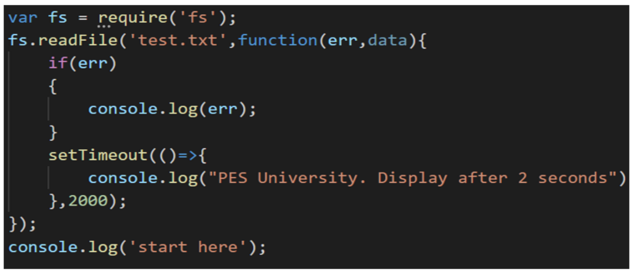
Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All the APIs of Node are written in such a way that they support callbacks.

For example, a function to read a file may start reading file and return the control to the execution environment immediately so that the next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as a parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results.

BLOCKING I/O



```
var fs = require('fs');
var data = fs.readFileSync('test.txt');
console.log(data.toString());
console.log('End here');
```



```
var fs = require('fs');
fs.readFile('test.txt',function(err,data){
    if(err)
    {
        console.log(err);
    }
    setTimeout(()=>{
        console.log("PES University. Display after 2 seconds")
    },2000);
});
console.log('start here');
```

NON-BLOCKING I/O

These two examples explain the concept of blocking and non-blocking calls.

- The first example shows that the program blocks until it reads the file and then only it proceeds to end the program.
- The second example shows that the program does not wait for file reading and proceeds to print "Start here" and at the same time, the program without blocking continues reading the file.

Thus, a blocking program executes very much in sequence. From the programming point of view, it is easier to implement the logic but non-blocking programs do not execute in sequence. In case a program needs to use any data to be processed, it should be kept within the same block to make it sequential execution.

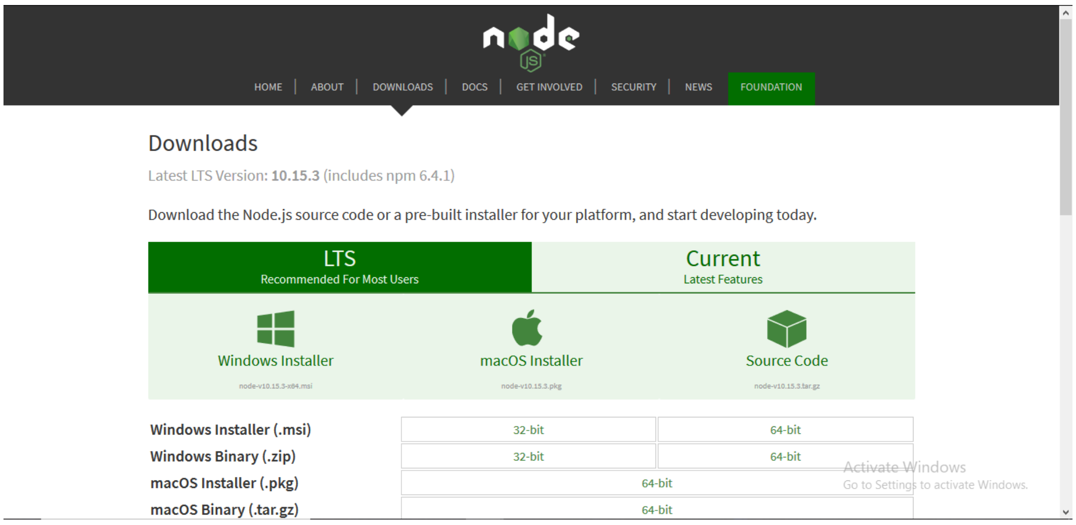
Installation of Node.js on Windows

Installing Node on Windows (WINDOWS 10):

You have to follow the following steps to install the Node.js on your Windows:

Step-1: Downloading the Node.js '.msi' installer.

The first step to install Node.js on windows is to download the installer. Visit the official Node.js website i.e) https://nodejs.org/en/download/ and download the .msi file according to your system environment (32-bit & 64-bit). An MSI installer will be downloaded on your system.



Downloads

Latest LTS Version: **10.15.3** (includes npm 6.4.1)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.zip)	32-bit	64-bit
macOS Installer (.pkg)	64-bit	
macOS Binary (.tar.gz)	64-bit	

NOTE :

A prompt saying – "This step requires administrative privileges" will appear.

Authenticate the prompt as an "Administrator"

Step-2: Installing Node.js.

Do not close or cancel the installer until the install is complete

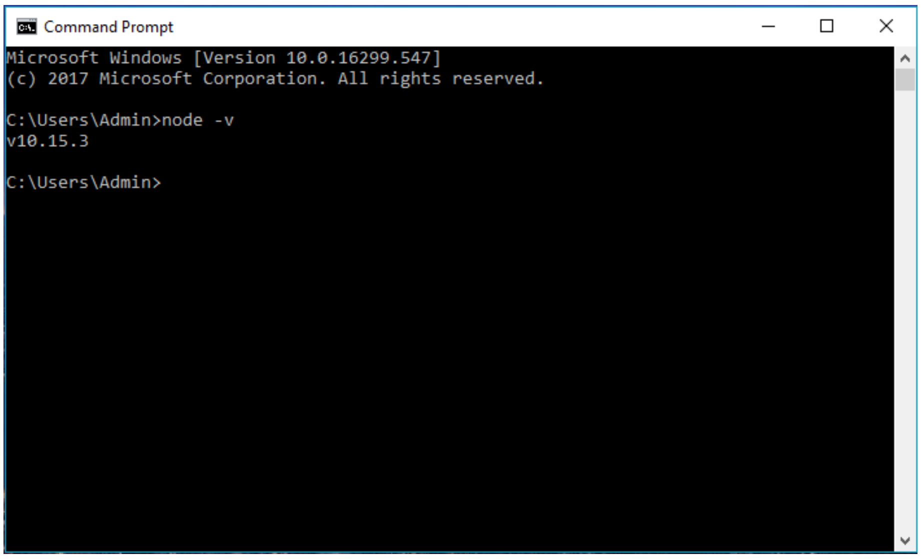
Complete the Node.js Setup Wizard.

Click “Finish”

Step 3: Verify that Node.js was properly installed or not.

To check that node.js was completely installed on your system or not, you can run the following command in your command prompt or Windows Powershell and test it:-

C:\Users\Admin> node -v



```
Microsoft Windows [Version 10.0.16299.547]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\Admin>node -v
v10.15.3

C:\Users\Admin>
```

If node.js was completely installed on your system, the command prompt will print the version of the node.js installed.

Step 4: Updating the Local npm version.

The final step in node.js installed is the updation of your local npm version(if required) – the package manager that comes bundled with Node.js.

You can run the following command, to quickly update the npm

npm install npm –global // Updates the ‘CLI’ client

Node.js Basics

Node.js is a cross-platform JavaScript runtime environment.

It allows the creation of scalable Web servers without threading and networking tools using JavaScript and a collection of "modules" that handle various core functionalities.

It can make console-based and web-based node.js applications.

Datatypes: Node.js contains various types of data types similar to JavaScript.

- Boolean
- Undefined
- Null
- String
- Number

Loose Typing: Node.js supports loose typing, it means you don't need to specify what type of information will be stored in a variable in advance. We use var keyword in Node.js to declare any type of variable. Examples are given below:

Example:

```
// Variable store number data type

var a = 35;

console.log(typeof a);


// Variable store string data type

a = "PES";

console.log(typeof a);


// Variable store Boolean data type

a = true;

console.log(typeof a);
```

```
// Variable store undefined (no value) data type
a = undefined;
console.log(typeof a);
```

Objects & Functions

Node.js objects are same as JavaScript objects i.e. the objects are similar to variable and it contains many values which are written as name: value pairs. Name and value are separated by colon and every pair is separated by comma.

Example:

```
var company = {
   Name: "PES University",
   Address: "India",
   Contact: "+919876543210",
   Email: "www.pes.edu"
};

// Display the object information
console.log("Information of variable company:", company);

// Display the type of variable
console.log("Type of variable company:", typeof company);
```

Functions:

Node.js functions are defined using function keyword then the name of the function and parameters which are passed in the function.

In Node.js, we don’t have to specify datatypes for the parameters and check the number of arguments received.

Node.js functions follow every rule which is there while writing JavaScript functions.

```
function multiply(num1, num2) {


    // It returns the multiplication
    // of num1 and num2
    return num1 * num2;
}


// Declare variable
var x = 2;
var y = 3;


// Display the answer returned by
// multiply function
console.log("Multiplication of", x, "and", y, "is", multiply(x, y));
```

If you observe in the above example, we have created a function called “multiply” with parameters same like JavaScript.

String and String Functions: In Node.js we can make a variable as string by assigning a value either by using single (”) or double (“”) quotes and it contains many functions to manipulate to strings.

Following is the example of defining string variables and functions in node.js.

```
var x = "Welcome ";
var y = 'Node.js Tutorials';
var z = ['Node', 'server', 'side'];
console.log(x);
console.log(y);
```

console.log("Concat Using (+) :", (x + y));

console.log("Concat Using Function :", (x.concat(y)));

console.log("Split string: ", x.split(' '));

console.log("Join string: ", z.join(', '));

console.log("Char At Index 5: ", x.charAt(5));

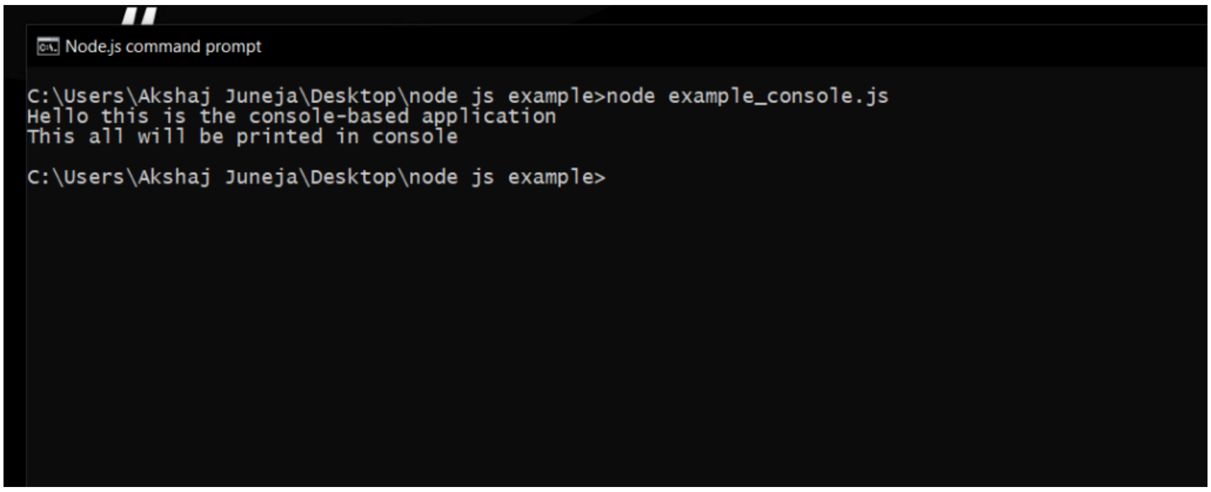
Node.js console-based application: Make a file called console.js with the following code.

console.log('Hello this is the console-based application');

console.log('This all will be printed in console');

// The above two lines will be printed in the console.

To run this file, **open node.js command prompt** and go to the folder where console.js file exist and write the following command. It will display content on console.



```
C:\Users\Akshaj Juneja\Desktop\node js example>node example_console.js
Hello this is the console-based application
This all will be printed in console

C:\Users\Akshaj Juneja\Desktop\node js example>
```

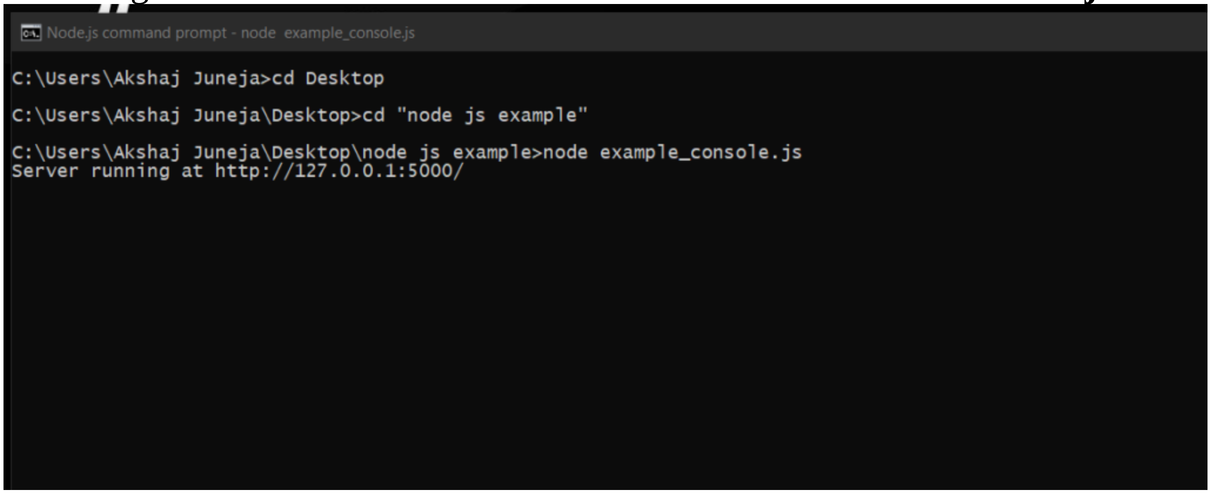
Node.js web-based application: Node.js web application contains different types of modules which is imported using **require()** directive and we have to create a server and write code for the read request and return response. Make a file web.js with the following code.

```
// Require http module
var http = require("http");
// Create server
http.createServer(function (req, res) {
    // Send the HTTP header
    // HTTP Status: 200 : OK
    // Content Type: text/plain
    res.writeHead(200, {'Content-Type': 'text/plain'});

    // Send the response body as "This is the example
    // of node.js web based application"
   res.end('This is the example of node.js web-based application \n');

// Console will display the message
}).listen(5000,
    ()=>console.log('Server running at http://127.0.0.1:5000/'));
```

To run this file follow the steps as given below:

- Search the node.js command prompt in the search bar and open the node.js command prompt.

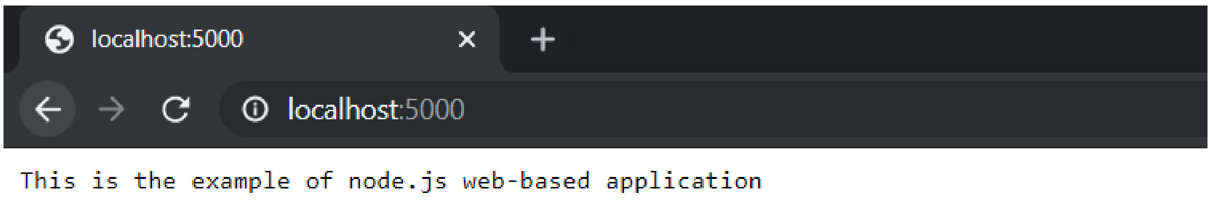
- Go to the folder using **cd** command in command prompt and write the following command **node web.js**



Node.js command prompt - node example_console.js

```
C:\Users\Akshaj Juneja>cd Desktop

C:\Users\Akshaj Juneja\Desktop>cd "node js example"

C:\Users\Akshaj Juneja\Desktop\node js example>node example_console.js
Server running at http://127.0.0.1:5000/
```

Now the server has started and go to the browser and open this url localhost:5000



```
This is the example of node.js web-based application
```

<u>NODE Modules</u>

NPM Package

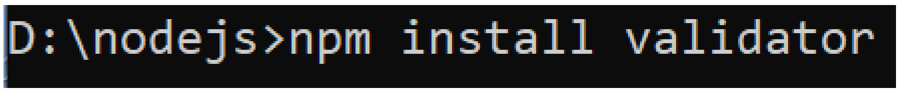
NPM is a package manager for Node.js packages, or modules if you like.

www.npmjs.com hosts thousands of free packages to download and use.

The NPM program is installed on your computer when you install Node.js

A package in Node.js contains all the files you need for a module.

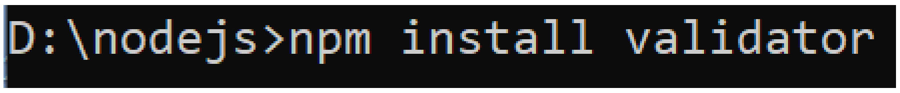
Modules are JavaScript libraries you can include in your project.

Example: 

```
D:\nodejs>npm install validator
```

validator package is downloaded and installed. NPM creates a folder named "node_modules", where the package will be placed.

To include a module, use the require() function with the name of the module



```
D:\nodejs>npm install validator
```

<u>What is a Module in Node.js?</u>

- Modules are the blocks of encapsulated code that communicates with an external application on the basis of their related functionality.
- Modules can be a single file or a collection of multiples files/folders.
- The reason programmers are heavily reliant on modules is because of their re-usability as well as the ability to break down a complex piece of code into manageable chunks

There are 2 types of Modules:

- **Built in Modules**
- **Local Modules**

To see all the available modules

- https://nodejs.org/dist/latest-v12.x/docs/api/

Modules can be imported to code using

require () function

- Few modules are inbuilt globally available **no need of require function.**

Ex: Console module, Timer Module

- Many modules need to be **explicitly included** in our application

Ex: File System module

Such modules need to be required at first in the application

Node.js has many built-in modules that are part of the platform and comes with Node.js installation. **These modules can be loaded into the program by using the require function.**

Syntax:

var module = require('module_name');

The require() function will return a JavaScript type depending on what the particular module returns.

Importing your own modules

- The module.exports is a special object which is included in every JavaScript file in the Node.js application by default.
- The module is a variable that represents the current module, and exports is an object that will be exposed as a module.
- So, whatever you assign to module. exports will be exposed as a module. It can be
    - Export Literals
    - Export Objects
    - Export Functions
    - Export Function as a class

Node JS Local-Modules

local modules are created locally in your Node.js application. Let’s create a simple calculating module that calculates various operations. Create a calc.js file that has the following code:

Filename: calc.js

```
exports.add = function (x, y) {
   return x + y;
};


exports.sub = function (x, y) {
   return x - y;
};


exports.mult = function (x, y) {
   return x * y;
};
```

Since this file provides attributes to the outer world via exports, another file can use its exported functionality using the require() function.

Filename: index.js

```
var calculator = require('./calc');


var x = 50, y = 20;


console.log("Addition of 50 and 10 is "
                + calculator.add(x, y));


console.log("Subtraction of 50 and 10 is "
                + calculator.sub(x, y));
```

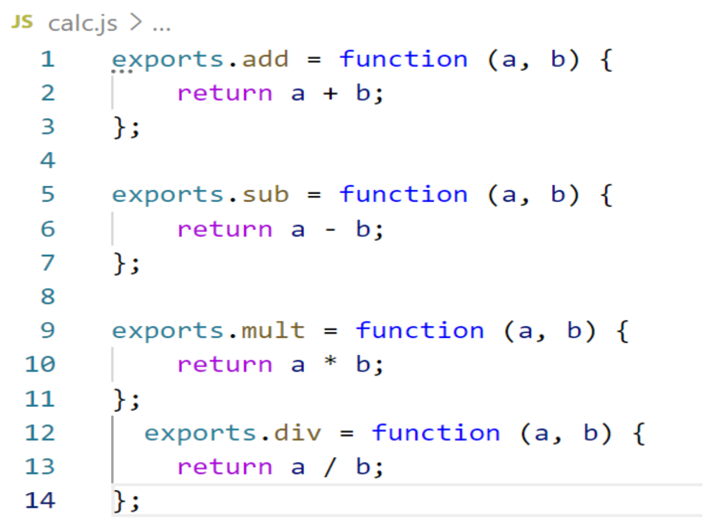
console.log("Multiplication of 50 and 10 is "

+ calculator.mult(x, y));

Step to run this program: Run index.js file using the following command:

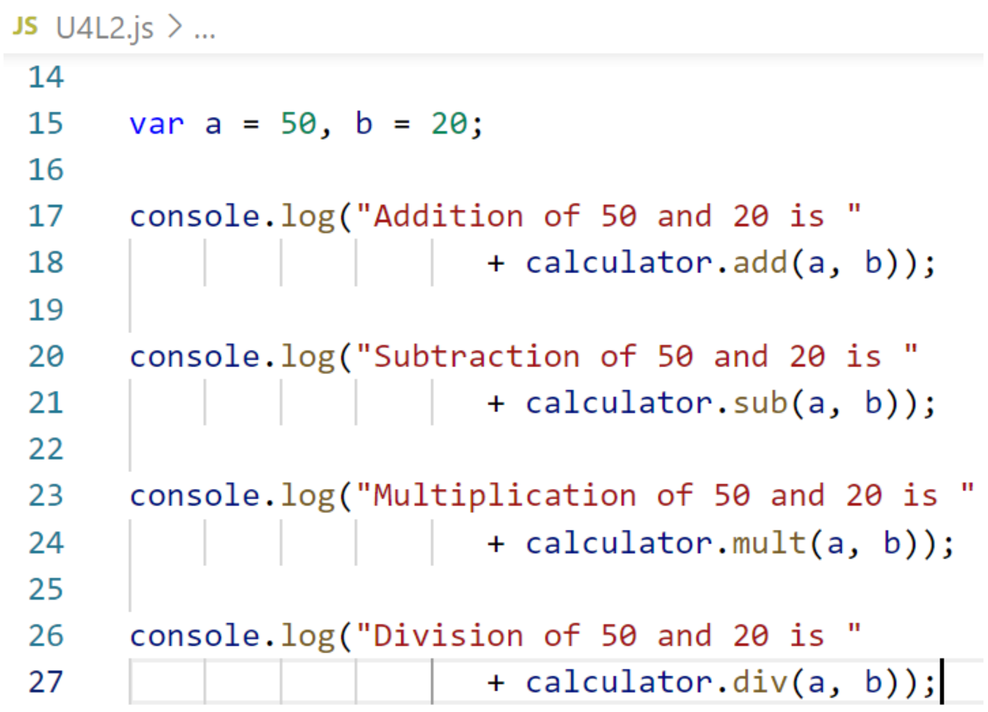
node index.js

Create Modules in Node.js

- To create a module in Node.js, use **exports** keyword tells Node.js that the function can be used outside the module.
- **Create a file that you want to export**
- 

calc.js > ...

```
exports.add = function (a, b) {
    return a + b;
};

exports.sub = function (a, b) {
    return a - b;
};

exports.mult = function (a, b) {
    return a * b;
};
  exports.div = function (a, b) {
    return a / b;
};
```


JS U4L2.js > ...

```
var a = 50, b = 20;

console.log("Addition of 50 and 20 is "
                    + calculator.add(a, b));

console.log("Subtraction of 50 and 20 is "
                    + calculator.sub(a, b));

console.log("Multiplication of 50 and 20 is "
                    + calculator.mult(a, b));

console.log("Division of 50 and 20 is "
                    + calculator.div(a, b));
```

Node JS Built-in Modules

- Node.js has many built-in modules that are part of the platform and comes with Node.js installation.
- These modules can be loaded into the program by using the require function.
- Syntax:
- **var module = require('module_name');**
- The require() function will return a JavaScript type depending on what the particular module returns.

Core Modules	**Description**
http	**creates an HTTP server in Node.js.**
assert	**set of assertion functions useful for testing.**
fs	**used to handle file system.**
path	**includes methods to deal with file paths.**
process	**provides information and control about the current Node.js process.**
os	**provides information about the operating system.**
querystring	**utility used for parsing and formatting URL query strings.**
url	**module provides utilities for URL resolution and parsing.**

Some Modules which are Globally Available where there is no need of Require () function

Ex: Console

Timer Module

Node JS Timer Module

- This module provides a way for functions to be called later at a given time.
- The Timer object is a global object in Node.js, and it is not necessary to import it

Method	**Description**
clearImmediate()	Cancels an Immediate object
clearInterval()	Cancels an Interval object
clearTimeout()	Cancels a Timeout object
ref()	Makes the Timeout object active. Will only have an effect if the Timeout.unref() method has been called to make the Timeout object inactive.
setImmediate()	Executes a given function immediately.
setInterval()	Executes a given function at every given milliseconds
setTimeout()	Executes a given function after a given time (in milliseconds)
unref()	Stops the Timeout object from remaining active

Ex:

```
function printHello() {
  console.log( "Hello, World!");
}
// Now call above function after 2 seconds
var timeoutObj = setTimeout(printHello, 2000);
```

Some More examples of Local Modules:

1.Exporting Date Module

Date.js

```
exports.myDateTime=function(){
return Date()
};
```

Main Program where we are importing Date

```
var D=require('./Date.js')
console.log(D.myDateTime());
```

2.

LOG.JS

```
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
```

In the above example of logging module, we have created an object with three functions - info(), warning() and error(). At the end, we have assigned this object to module.exports. The module.exports in the above example 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 your application, you need to load it using require() function in the same way as core module. However, you need to specify the path of JavaScript file of the module.

The following example demonstrates how to use the above logging module contained in Log.js.

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

myLogModule.info('Node.js started');
```

In the above example, 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**. So 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()

3.

Local.js

```
const welcome = {
   sayHello : function() {
       console.log("Hello  user");
  },
  currTime : new Date().toLocaleDateString(),
  companyName : "PESU"
}
```

Main.js

```
var local = require("./Welcome.js");
local.sayHello();
```

```
console.log(local.currTime);

console.log(local.companyName);

module.exports = welcome
```

Export Module in Node.js

The module.exports is a special object which is included in every JavaScript file in the Node.js application by default. The module is a variable that represents the current module, and exports is an object that will be exposed as a module. So, whatever you assign to module.exports will be exposed as a module.

FILE Module

- https://nodejs.org/api/fs.html, https://nodejs.org/api/querystring.html

The fs module provides a lot of very useful functionality to access and interact with the file system.

There is no need to install it. Being part of the Node.js core, it can be used by simply requiring it:

const fs = require('fs')

Common use for the File System module:

Read files

Create files

Update files

Delete files

Rename files

Synchronous vs Asynchronous

- Every method in the fs module has synchronous as well as asynchronous forms.
- Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as error.
- It is better to use an asynchronous method instead of a synchronous method, as the former never blocks a program during its execution, whereas the second one does.

What is Synchronous and Asynchronous approach?

Synchronous approach: They are called blocking functions as it waits for each operation to complete, only after that, it executes the next operation, hence blocking the next command from execution i.e. a command will not be executed until & unless the query has finished executing to get all the result from previous commands.

Asynchronous approach:

They are called non-blocking functions as it never waits for each operation to complete, rather it executes all operations in the first go itself.

The result of each operation will be handled once the result is available i.e. each command will be executed soon after the execution of the previous command.

While the previous command runs in the background and loads the result once it is finished processing the data.

Example:

Synchronous:

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

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

Asynchronous:

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

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

Once you do so, you have access to all its methods, which include:

- fs.access(): check if the file exists and Node.js can access it with its permissions
- fs.appendFile(): append data to a file. If the file does not exist, it's created
- fs.chmod(): change the permissions of a file specified by the filename passed. Related: fs.lchmod(), fs.fchmod()

- fs.chown(): change the owner and group of a file specified by the filename passed. Related: fs.fchown(), fs.lchown()
- fs.close(): close a file descriptor
- fs.copyFile(): copies a file
- fs.createReadStream(): create a readable file stream
- fs.createWriteStream(): create a writable file stream
- fs.link(): create a new hard link to a file
- fs.mkdir(): create a new folder
- fs.mkdtemp(): create a temporary directory
- fs.open(): set the file mode
- fs.readdir(): read the contents of a directory
- fs.readFile(): read the content of a file. Related: fs.read()
- fs.readlink(): read the value of a symbolic link
- fs.realpath(): resolve relative file path pointers (., ..) to the full path
- fs.rename(): rename a file or folder
- fs.rmdir(): remove a folder
- fs.stat(): returns the status of the file identified by the filename passed. Related: fs.fstat(), fs.lstat()
- fs.symlink(): create a new symbolic link to a file
- fs.truncate(): truncate to the specified length the file identified by the filename passed. Related: fs.ftruncate()
- fs.unlink(): remove a file or a symbolic link
- fs.unwatchFile(): stop watching for changes on a file
- fs.utimes(): change the timestamp of the file identified by the filename passed. Related: fs.futimes()
- fs.watchFile(): start watching for changes on a file. Related: fs.watch()
- fs.writeFile(): write data to a file. Related: fs.write()

One peculiar thing about the fs module is that all the methods are asynchronous by default, but they can also work synchronously by appending Sync.

For example:

- fs.rename()
- fs.renameSync()
- fs.write()
- fs.writeSync()

This makes a huge difference in your application flow.

Open a File

Syntax

Following is the syntax of the method to open a file in asynchronous mode −

```
fs.open(path, flags[, mode], callback)
```

Parameters

Here is the description of the parameters used −

- **path** − This is the string having file name including path.
- **flags** − Flags indicate the behavior of the file to be opened. All possible values have been mentioned below.
- **mode** − It sets the file mode (permission and sticky bits), but only if the file was created. It defaults to 0666, readable and writeable.
- **callback** − This is the callback function which gets two arguments (err, fd).

Flags

Flags for read/write operations are −

Sr.No.	Flag & Description
1	**r** Open file for reading. An exception occurs if the file does not exist.
2	**r+** Open file for reading and writing. An exception occurs if the file does not exist.
3	**rs** Open file for reading in synchronous mode.
4	**rs+** Open file for reading and writing, asking the OS to open it synchronously. See notes for 'rs' about using this with caution.
5	**w** Open file for writing. The file is created (if it does not exist) or truncated

	(if it exists).
6	**wx** Like 'w' but fails if the path exists.
7	**w+** Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
8	**wx+** Like 'w+' but fails if path exists.
9	**a** Open file for appending. The file is created if it does not exist.
10	**ax** Like 'a' but fails if the path exists.
11	**a+** Open file for reading and appending. The file is created if it does not exist.
12	**ax+** Like 'a+' but fails if the the path exists.

Writing a File

Syntax

Following is the syntax of one of the methods to write into a file −

fs.writeFile(filename, data[, options], callback)

This method will over-write the file if the file already exists. If you want to write into an existing file then you should use another method available.

Parameters

Here is the description of the parameters used −

- **path** − This is the string having the file name including path.

- **data** − This is the String or Buffer to be written into the file.
- **options** − The third parameter is an object which will hold {encoding, mode, flag}. By default. encoding is utf8, mode is octal value 0666. and flag is 'w'
- **callback** − This is the callback function which gets a single parameter err that returns an error in case of any writing error.

Reading a File

- Node implements File I/O using simple wrappers around standard POSIX functions.
- The Node File System (fs) module can be imported using the following syntax −

Synchronous vs Asynchronous

- Every method in the fs module has synchronous as well as asynchronous forms.
- Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as error.
- It is better to use an asynchronous method instead of a synchronous method, as the former never blocks a program during its execution, whereas the second one does.

Syntax

Following is the syntax of one of the methods to read from a file −

fs.read(fd, buffer, offset, length, position, callback)

This method will use file descriptor to read the file. If you want to read the file directly using the file name, then you should use another method available.

Parameters

Here is the description of the parameters used −

- **fd** − This is the file descriptor returned by fs.open().
- **buffer** − This is the buffer that the data will be written to.
- **offset** − This is the offset in the buffer to start writing at.
- **length** − This is an integer specifying the number of bytes to read.

- **position** − This is an integer specifying where to begin reading from in the file. If position is null, data will be read from the current file position.
- **callback** − This is the callback function which gets the three arguments, (err, bytesRead, buffer).

Closing a File

Syntax

Following is the syntax to close an opened file −

```
fs.close(fd, callback)
```

Parameters

Here is the description of the parameters used −

- **fd** − This is the file descriptor returned by file fs.open() method.
- **callback** − This is the callback function No arguments other than a possible exception are given to the completion callback.

Truncate a File

Syntax

Following is the syntax of the method to truncate an opened file −

```
fs.ftruncate(fd, len, callback)
```

Parameters

Here is the description of the parameters used −

- **fd** − This is the file descriptor returned by fs.open().
- **len** − This is the length of the file after which the file will be truncated.
- **callback** − This is the callback function No arguments other than a possible exception are given to the completion callback.

Delete a File

Syntax

Following is the syntax of the method to delete a file −

```
fs.unlink(path, callback)
```

Parameters

Here is the description of the parameters used −

- **path** − This is the file name including path.
- **callback** − This is the callback function No arguments other than a possible exception are given to the completion callback.

URL Module:

As nodejs.org suggests:

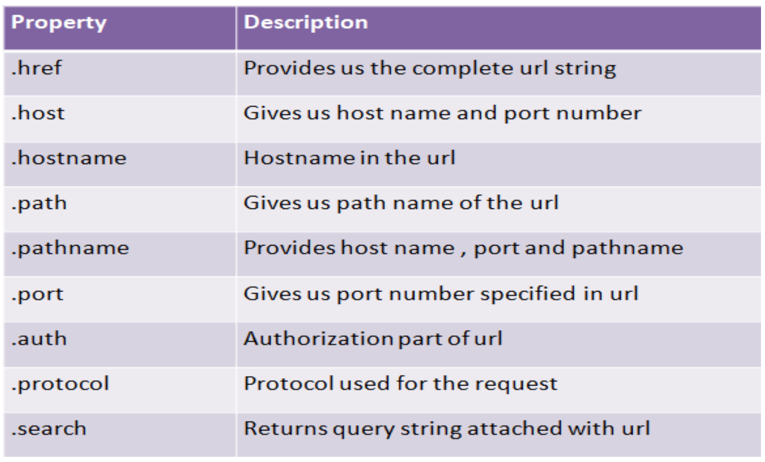
The URL module provides utilities for URL resolution and parsing. It can be accessed using:

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

Url module is one of the core modules that comes with node.js, which is used to parse the URL and its other properties.

By using URL module, it provides us with so many properties to work with.

These all are listed below:



Property	Description
.href	Provides us the complete url string
.host	Gives us host name and port number
.hostname	Hostname in the url
.path	Gives us path name of the url
.pathname	Provides host name , port and pathname
.port	Gives us port number specified in url
.auth	Authorization part of url
.protocol	Protocol used for the request
.search	Returns query string attached with url

As you can see in the above screen, there are various properties used for URL module.

Code Demonstrated in Class:

//
Opening a file:
//
var fs = require("fs");

```
// Asynchronous - Opening File
console.log("Going to open file!");
fs.open('input.txt', 'r+', function(err, data) {
  if (err) {
    return console.error(err);
  }
  console.log(data);
  console.log("File opened successfully!");
});


/////////////////////////////////////////////////////////////////////////
Reading from file:
/////////////////////////////////////////////////////////////////////////
var fs = require('fs');

fs.readFile('input.txt', function (err, data) {
                 if (err) throw err;

   console.log(data.toString());
});

var fs = require('fs');

var data = fs.readFileSync('data.txt', 'utf8');
console.log(data);

/////////////////////////////////////////////////////////////////////////////////////
Writing a file:
/////////////////////////////////////////////////////////////////////////
var fs = require("fs");

console.log("Going to write into existing file");
fs.writeFile('input.txt', 'PES University!', function(err) {
  if (err) {
    return console.error(err);
  }

  console.log("Data written successfully!");
  console.log("Let's read newly written data");

  fs.readFile('data.txt', function (err, data) {
    if (err) {
         return console.error(err);
      }
      console.log("Asynchronous read: " + data.toString());
   });
});

////////////////////////////////////////////////////////////////////
Reading from a file
////////////////////////////////////////////////////////////////////

var fs = require('fs');

fs.open('data.txt', 'r', function (err, fd) {

                        if (err) {
                        return console.error(err);
    }

                        var buffr = new Buffer(1024);
    console.log("File opened successfully!");
   console.log("Going to truncate the file after 3 bytes");

   // Truncate the opened file.
   fs.ftruncate(fd, 3, function(err) {
      if (err) {
         console.log(err);
      }
      console.log("File truncated successfully.");
      console.log("Going to read the same file");

    fs.read(fd, buffr, 0, buffr.length, 0, function (err, bytes) {

                        if (err) throw err;

                        // Print only read bytes to avoid junk.
                        if (bytes > 0) {
           console.log(buffr.slice(0, bytes).toString());
        }

                        // Close the opened file.
        fs.close(fd, function (err) {
                        if (err) throw err;
        });
    });
});

//////////////////////////////////////////////////////////////////////////////
Delete a file
//////////////////////////////////////////////////////////////////////////////
var fs = require('fs');

fs.unlink('data.txt', function () {

    console.log('write operation complete.');

});
```

HTTP Module

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

To include the HTTP module, use the require() method:

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

To process HTTP requests in JavaScript and Node.js, we can use the built-in http module. This core module is key in leveraging Node.js networking and is extremely useful in creating HTTP servers and processing HTTP requests.

The http module comes with various methods that are useful when engaging with HTTP network requests. One of the most commonly used methods within the http module is the .createServer() method. This method is responsible for doing exactly what its namesake implies; it creates an HTTP server. To implement this method to create a server, the following code can be used:

```
const server = http.createServer((req, res) => {
  res.end('Server is running!');
});

server.listen(8080, () => {
  const { address, port } = server.address();
  console.log(`Server is listening on: http://${address}:${port}`);
})
```

The .createServer() method takes a single argument in the form of a callback function. This callback function has two primary arguments; the request (commonly written as req) and the response (commonly written as res).

The req object contains all of the information about an HTTP request ingested by the server. It exposes information such as the HTTP method (GET,POST, etc.), the pathname, headers, body, and so on. The res object contains methods and properties pertaining to the generation of a response by the HTTP server. This object contains methods such as .setHeader (sets HTTP headers on the response), .statusCode (set the status code of the response), and .end() (dispatches the response to the client who made the request). In the example above, we use the .end() method to send the string 'Server is Running!' to the client, which will display on the web page.

Once the .createServer() method has instantiated the server, it must begin listening for connections. This final step is accomplished by the .listen() method on the server instance. This method takes a port number as the first argument, which tells the server to listen for connections at the given port number. In our example above, the server has been set to listen on port 8080. Additionally, the .listen() method takes an optional callback function as a second argument, allowing it to carry out a task after the server has successfully started.

HTTP Module: Some Important Methods

http.STATUS_CODES;	A collection of all the standard HTTP response status codes, and the short description of each.
http.request(options, [callback]);	This function allows one to transparently issue requests.
http.get(options, [callback]);	Set the method to GET and calls req.end() automatically.
server = http.createServer([requestListener]);	Returns a new web server object. The requestListener is a function which is automatically added to the 'request' event.
server.listen(port, [hostname], [backlog], [callback]);	Begin accepting connections on the specified port and hostname.
server.close([callback]);	Stops the server from accepting new connections.
request.write(chunk, [encoding]);	Sends a chunk of the body.
request.end([data], [encoding]);	Finishes sending the request. If any parts of the body are unsent, it will flush them to the stream.
request.abort();	Aborts a request.
response.write(chunk, [encoding]);	This sends a chunk of the response body. If this merthod is called and response.writeHead() has

	not been called, it will switch to implicit header mode and flush the implicit headers.
response.writeHead(statusCode, [reasonPhrase], [headers]);	Sends a response header to the request.
response.getHeader(name);	Reads out a header that's already been queued but not sent to the client. Note that the name is case insensitive.
response.removeHeader(name);	Removes a header that's queued for implicit sending.
response.end([data], [encoding]);	This method signals to the server that all of the response headers and body have been sent; that server should consider this message complete. The method, response.end(), MUST be called on each response.

Handling HTTP Requests

In the following section we will learn to handle HTTP request when we open http://localhost:9000 url.

The callback function that we saw above when we created the server has two parameters namely, request and response.

The request object has url property and it holds the requested url. So, we will use it.

The response object has writeHead method which helps in setting the header. Then we have the write method that helps to write the response body. And there is end method which helps in sending the response.

Lets say, we want to print out the message Hello World! when we visit the home page at http://localhost:9000. As, it is the home page so we have to check for /.

```
const http = require('http');
const PORT = 9000;
const server = http.createServer(function(request, response) {
   // we are setting the header
   const header = {
      'Content-Type': 'text/html'
   };
   // checking for home page url /
   if (request.url === '/') {
      response.writeHead(200, header);
      response.write('Hello World!');
   }
   // if requested url is not known then prompt error response
   else {
      response.writeHead(400, header);
      response.write('Bad request!');
   }
   response.end();
});
server.listen(PORT);
console.log(`Server started and listening at port ${PORT}...`);
```

Now, stop the running server in the terminal and re-run it. Now, visit the home page http://localhost:9000 url. This time we will see the message Hello World!

This happens because the if condition is satisfied and if-block is executed.

Now, if we visit something like http://localhost:9000/some-unknown-page then we will get Bad Request message.

This happens because we are not handling the /some-unknown-page and hence the else-block is executed which gives the Bad request! message as response.

Making HTTP Requests

The request() method supports multiple function signatures. You'll use this one for the subsequent example:

```
http.request(Options_Object, Callback_Function)
```

The first argument is a string with the API endpoint. The second argument is a JavaScript object containing all the options for the request. The last argument is a callback function to handle the response.

Making HTTP Requests: Options

host	A domain name or IP address of the server to issue the request to. Defaults to 'localhost'.
hostname	To support url.parse() hostname is preferred over host
port	Port of the remote server. Defaults to 80.
method	A string specifying the HTTP request method. Defaults to 'GET'.
path	Request path. Defaults to '/'. Should include query string if any. E.G. '/index.html?page=12'
headers	An object containing request headers.
auth	Basic authentication i.e. 'user:password' to compute an Authorization header.

```
const https = require('https')
const options = {
  hostname: 'example.com',
  port: 443,
  path: '/todos',
  method: 'GET'
}

const req = https.request(options, res => {
  console.log(`statusCode: ${res.statusCode}`)

 res.on('data', d => {
    process.stdout.write(d)
  })
})

req.on('error', error => {
  console.error(error)
})

req.end()
```

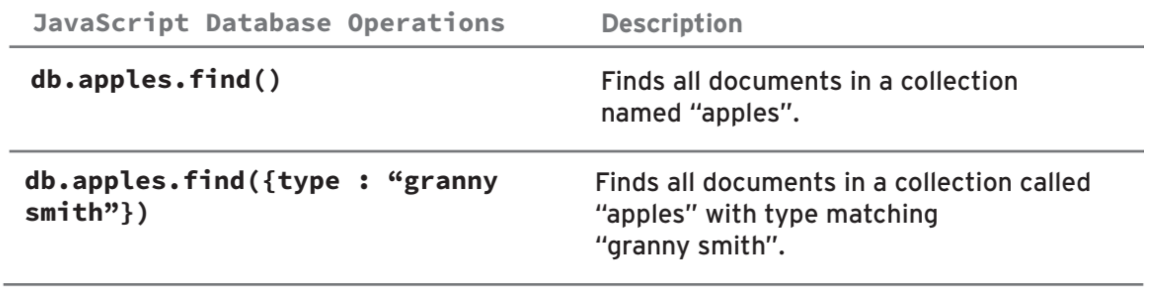
MongoDB and Its Features

MongoDB is a document-oriented NoSQL database used for high volume data storage. Instead of using tables and rows as in the traditional relational databases, MongoDB makes use of collections and documents. Documents consist of key-value pairs which are the basic unit of data in MongoDB. Collections contain sets of documents and function which is the equivalent of relational database tables. MongoDB is a database which came into light around the mid-2000s

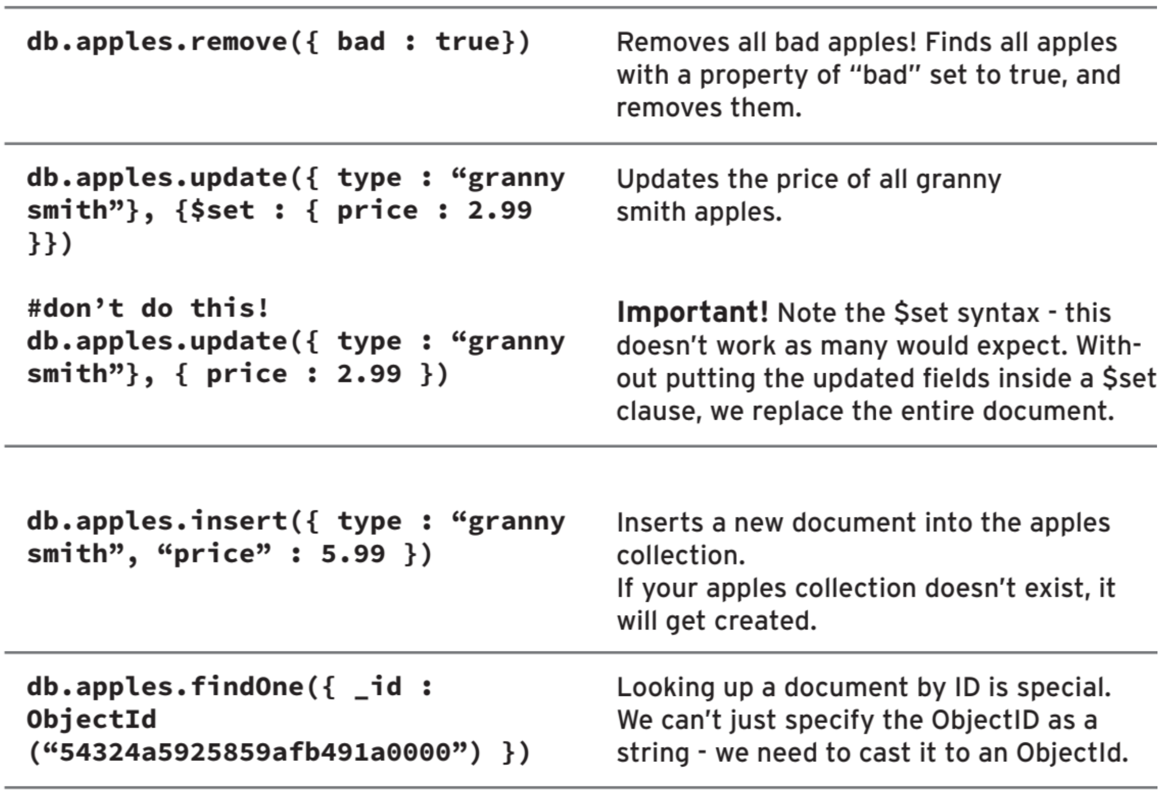
1. Each database contains collections which in turn contains documents. Each document can be different with a varying number of fields. The size and content of each document can be different from each other.
2. The document structure is more in line with how developers construct their classes and objects in their respective programming languages. Developers will often say that their classes are not rows and columns but have a clear structure with key-value pairs.
3. The rows (or documents as called in MongoDB) doesn't need to have a schema defined beforehand. Instead, the fields can be created on the fly.
4. The data model available within MongoDB allows you to represent hierarchical relationships, to store arrays, and other more complex structures more easily.
5. Scalability – The MongoDB environments are very scalable. Companies across the world have defined clusters with some of them running 100+ nodes with around millions of documents within the database

Working with a Collection

You can perform operations using the `db` variable. Every collection has a property on the `db` variable - e.g. the `apples` collection is `db.apples`.



JavaScript Database Operations	Description
`db.apples.find()`	Finds all documents in a collection named "apples".
`db.apples.find({type : "granny smith"})`	Finds all documents in a collection called "apples" with type matching "granny smith".



`db.apples.remove({ bad : true})`	Removes all bad apples! Finds all apples with a property of "bad" set to true, and removes them.
`db.apples.update({ type : "granny smith"}, {$set : { price : 2.99 }})`	Updates the price of all granny smith apples.
`#don't do this! db.apples.update({ type : "granny smith"}, { price : 2.99 })`	**Important!** Note the $set syntax - this doesn't work as many would expect. Without putting the updated fields inside a $set clause, we replace the entire document.
`db.apples.insert({ type : "granny smith", "price" : 5.99 })`	Inserts a new document into the apples collection. If your apples collection doesn't exist, it will get created.
`db.apples.findOne({ _id : ObjectId("54324a5925859afb491a0000") })`	Looking up a document by ID is special. We can't just specify the ObjectID as a string - we need to cast it to an ObjectId.

MongoDB Connectivity: Create a Database

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist, and make a connection to it.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/mydb";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  console.log("Database created!");
  db.close();
});
```

MongoDB Connectivity: Create a Collection

To create a collection in MongoDB, use the createCollection() method:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
```

```
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.createCollection("customers", function(err, res) {
    if (err) throw err;
    console.log("Collection created!");
    db.close();
  });
});
```

MongoDB Connectivity: Insert into Collection

To insert a record, or *document* as it is called in MongoDB, into a collection, we use the insertOne() method.

A **document** in MongoDB is the same as a **record** in MySQL

The first parameter of the insertOne() method is an object containing the name(s) and value(s) of each field in the document you want to insert.

It also takes a callback function where you can work with any errors, or the result of the insertion:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myobj = { name: "Company Inc", address: "Highway 37" };
  dbo.collection("customers").insertOne(myobj, function(err, res) {
    if (err) throw err;
    console.log("1 document inserted");
    db.close();
  });
});
```

MongoDB Connectivity: Read

To select data from a table in MongoDB, we can also use the find() method. The find() method returns all occurrences in the selection. The first parameter of the find() method is a query object. In this example we use an empty query object, which selects all documents in the collection.

No parameters in the find() method gives you the same result as **SELECT *** in MySQL.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection("customers").find({}).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

MongoDB Connectivity: Filter results

When finding documents in a collection, you can filter the result by using a query object. The first argument of the find() method is a query object, and is used to limit the search.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var query = { address: "Park Lane 38" };
  dbo.collection("customers").find(query).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

Lesson 5: HTTP web module

HTTP Module: - https://nodejs.org/api/http.html

A Web Server is a software application which handles HTTP requests sent by the HTTP client, like web browsers, and returns web pages in response to the clients.

Web servers usually deliver html documents along with images, style sheets, and scripts.

Most of the web servers support server-side scripts, using scripting languages or redirecting the task to an application server which retrieves data from a database and performs complex logic and then sends a result to the HTTP client through the Web server.

Apache web server is one of the most commonly used web servers. It is an open source project.

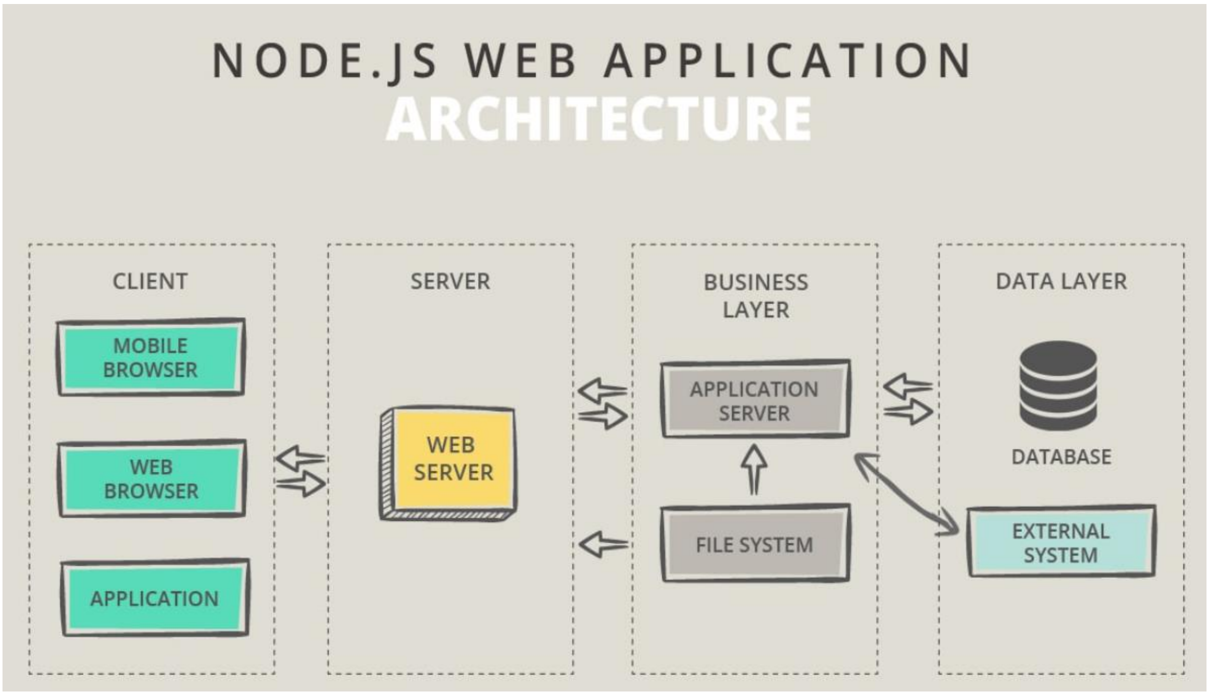
A Web application is usually divided into four layers −

Client − This layer consists of web browsers, mobile browsers or applications which can make HTTP requests to the web server.

Server − This layer has the Web server which can intercept the requests made by the clients and pass them the response.

Business − This layer contains the application server which is utilized by the web server to do the required processing. This layer interacts with the data layer via the database or some external programs.

Data − This layer contains the databases or any other source of data.



<u>Node.js Web Server</u>

- 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.

The components of a Node.js application.

Import required modules − We use the require directive to load Node.js modules.

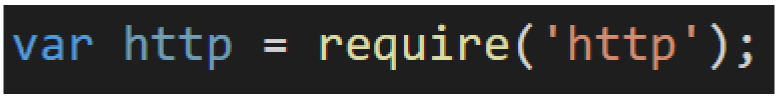
Create server − A server which will listen to client's requests similar to Apache HTTP Server. Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

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

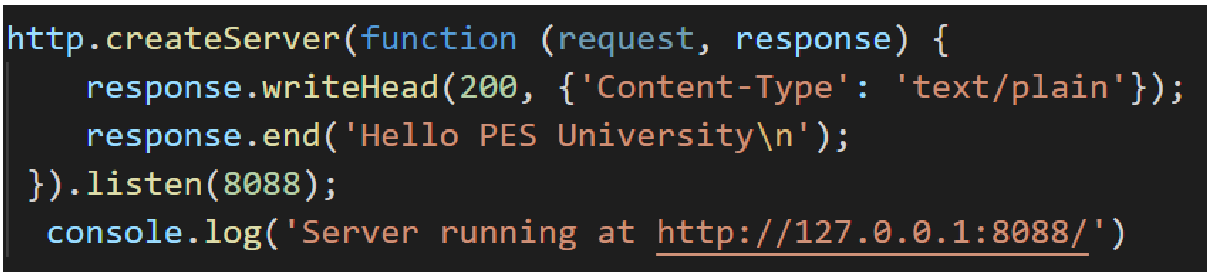
We use the require directive to load the http module and store the returned HTTP instance into an http variable as follows



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

Step 2 - Create Server

- We use the created http instance and call **http.createServer()** method to create a server instance.
- Then we bind it at port 8088 using the **listen** method associated with the server instance.



```
http.createServer(function (request, response) {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    response.end('Hello PES University\n');
 }).listen(8088);
  console.log('Server running at http://127.0.0.1:8088/')
```

Step 3 - Testing Request & Response

$node app.js

Verify the Output. Server has started.

HTTP Module – Web Server using Node

Server running at http://127.0.0.1:8088/

```
var http = require('http'); // 1 - Import Node.js core module

var server = http.createServer(function (req, res) {   // 2 - creating server

    //handle incomming requests here..

});

server.listen(5000); //3 - listen for any incoming requests

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

In the above example, we 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, to start listening to incoming requests on port 5000.

You can specify any unused port here.

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.

Ex:

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

//create a server object:

http.createServer(function (req, res) {

  res.write('Hello World!'); //write a response to the client

  res.end(); //end the response

}).listen(8080); //the server object listens on port 8080
```

The module provides some properties and methods, and some classes.

Properties

http.METHODS

This property lists all the HTTP methods supported:

```
> require('http').METHODS
```

http.STATUS_CODES

This property lists all the HTTP status codes and their description:

```
> require('http').STATUS_CODES
```

http.globalAgent

It points to the global instance of the Agent object, which is an instance of the http.Agent class.

It is used to manage connections persistence and reuse for HTTP clients, and it's a key component of Node.js HTTP networking.

Methods

http.createServer()

Return a new instance of the http.Server class.

Usage:

```
const server = http.createServer((req, res) => {
  //handle every single request with this callback
})
```

http.request()

Makes an HTTP request to a server, creating an instance of the http.ClientRequest class.

http.get()

Similar to http.request(), but automatically sets the HTTP method to GET, and calls req.end() automatically.

Classes

The HTTP module provides 5 classes:

- http.Agent
- http.ClientRequest
- http.Server
- http.ServerResponse
- http.IncomingMessage

http.Agent

Node.js creates a global instance of the http.Agent class to manage connections persistence and reuse for HTTP clients, a key component of Node.js HTTP networking.

This object makes sure that every request made to a server is queued and a single socket is reused.

It also maintains a pool of sockets. This is key for performance reasons.

http.ClientRequest

An http.ClientRequest object is created when http.request() or http.get() is called.

When a response is received, the response event is called with the response, with an http.IncomingMessage instance as argument.

The returned data of a response can be read in 2 ways:

- you can call the response.read() method
- in the response event handler you can setup an event listener for the data event, so you can listen for the data streamed into.

http.Server

This class is commonly instantiated and returned when creating a new server using http.createServer().

Once you have a server object, you have access to its methods:

- listen() starts the HTTP server and listens for connections
- close() stops the server from accepting new connections

http.ServerResponse

Created by an http.Server and passed as the second parameter to
the request event it fires.

Commonly known and used in code as res:

```
const server = http.createServer((req, res) => {
  //res is an http.ServerResponse object
})
```

The method you'll always call in the handler is end(), which closes the response, the message is complete and the server can send it to the client. It must be called on each response.

These methods are used to interact with HTTP headers:

- getHeaderNames() get the list of the names of the HTTP headers already set
- getHeaders() get a copy of the HTTP headers already set
- setHeader('headername', value) sets an HTTP header value
- getHeader('headername') gets an HTTP header already set
- removeHeader('headername') removes an HTTP header already set
- hasHeader('headername') return true if the response has that header set
- headersSent() return true if the headers have already been sent to the client

After processing the headers you can send them to the client by calling response.writeHead(), which accepts the statusCode as the first parameter, the optional status message, and the headers object.

To send data to the client in the response body, you use write(). It will send buffered data to the HTTP response stream.

http.IncomingMessage

An http.IncomingMessage object is created by:

- http.Server when listening to the request event
- http.ClientRequest when listening to the response event

It can be used to access the response:

- status using its statusCode and statusMessage methods

- headers using its headers method or rawHeaders
- HTTP method using its method method
- HTTP version using the httpVersion method
- URL using the url method
- underlying socket using the socket method

The data is accessed using streams, since http.IncomingMessage implements the Readable Stream interface.

Class code Demonstration:

Client.js

```
var http = require('http');
var fetch = require('node-fetch');

//options to be used by request
/*
var options =
{
host: 'localhost',
port: '8081',
path : '/pes.html',
};

//callback function is used to deal with the response

var callback = function(response)
{
   var body = '';
   response.on('data',function(data)
   {

      body+= data;
   });
}
//make a request to the server
var req = http.request(options,callback);
req.end();  */


/*fetch('http://localhost:8081/sample.json', {
        method: 'GET',
        headers: { 'Content-Type': 'application/json' },
    })
    .then(res => res.json())
    .then(res => console.log(res));
*/
    fetch('http://localhost:8081/sample.json', {
        method: 'POST',
        body:    JSON.stringify({"name":"Aruna modified","col":"MIT"}),
        headers: { 'Content-Type': 'application/json' },
    })
     .then(res => console.log(res));
```

Server.js

```
// http module ccreating a web server
var url = require('url');
var http = require('http');
var fs = require('fs');
var MongoClient = require('mongodb').MongoClient;

//create a server
http.createServer(function (request, response) {

    //parse the request containing the filename
    var pathname = url.parse(request.url).pathname;


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

    if (request.method == "GET") {
        //Read the requested file content from filesystem
        // fs.readFile(pathname.substr(1), function (err, data) {
        //     if (err) {
        //         console.log(err);
        //         //HTTP Status 404 not found
        //         response.writeHead(404, { 'Content-Type': 'text/html' });
        //     }
        //     else {
        //         //page found and status of 200 has to be returned
        //         response.writeHead(200, { 'Content-Type': 'text/html' });

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

        // });
        //Read from the mongodb
        console.log('executing mongo');
        MongoClient.connect("mongodb://localhost:27017", {
            useUnifiedTopology:
                true
        }, function (err, client) {
            console.log("Connected successfully to server");
            const db = client.db("pes");
            db.collection("student").find({}).toArray(function (err, docs) {
                response.writeHead(200, { 'Content-Type': 'application/json' });
                //write the content of the file to response body
                response.write(JSON.stringify(docs));
                client.close();
                response.end();
            });
        });

    }
    else {
        let body = [];
        request.on('data', (chunk) => {
            body.push(chunk);
        }).on('end', () => {
            body = Buffer.concat(body).toString();
            // at this point, `body` has the entire request body stored in it as a string
        });

        // write to file
        // fs.writeFile(pathname.substr(1), body, (err, res) => {
        //     response.writeHead(200, { 'Content-Type': 'application/json' });
        //     response.end();
        // });
```

```
        //write to mongodb
        console.log('executing mongo');
        MongoClient.connect("mongodb://localhost:27017", {
            useUnifiedTopology:
                true
        }, function (err, client) {
            console.log("Connected successfully to server");
            const db = client.db("pes");
            db.collection("student").insertOne(JSON.parse(body)).then(r => {
                response.writeHead(200, { 'Content-Type': 'application/json' });
                //write the content of the file to response body
                client.close();
                response.end();
            })
        });
    }
}).listen(8081);
console.log('server running at the link http://localhost/8081');
```

Read the Query String

The function passed into the http.createServer() has a req argument that represents the request from the client, as an object (http.IncomingMessage object).

This object has a property called "url" which holds the part of the url that comes after the domain name:

demo_http_url.js

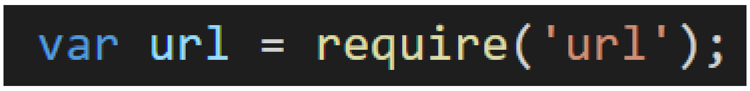
```
var http = require('http');

http.createServer(function (req, res) {

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

  res.write(req.url);   //requesting URL

  res.end();

}).listen(8080);
```

URL Module:

The URL module splits up a web address into readable parts.

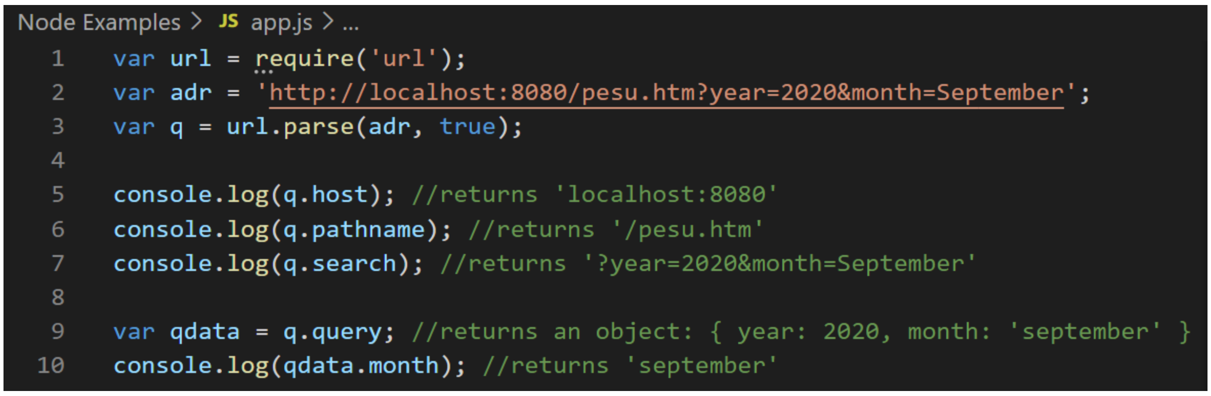
To include the URL module, use the require() method:

Parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties:



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

Parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties:



Node Examples > JS app.js > ...

```
var url = require('url');
var adr = 'http://localhost:8080/pesu.htm?year=2020&month=September';
var q = url.parse(adr, true);

console.log(q.host); //returns 'localhost:8080'
console.log(q.pathname); //returns '/pesu.htm'
console.log(q.search); //returns '?year=2020&month=September'

var qdata = q.query; //returns an object: { year: 2020, month: 'september' }
console.log(qdata.month); //returns 'september'
```

HTTP Module

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

To include the HTTP module, use the require() method:

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

To process HTTP requests in JavaScript and Node.js, we can use the built-in http module. This core module is key in leveraging Node.js networking and is extremely useful in creating HTTP servers and processing HTTP requests.

The http module comes with various methods that are useful when engaging with HTTP network requests. One of the most commonly used methods within the http module is the .createServer() method. This method is responsible for doing exactly what its namesake implies; it creates an HTTP server. To implement this method to create a server, the following code can be used:

```
const server = http.createServer((req, res) => {
  res.end('Server is running!');
});

server.listen(8080, () => {
  const { address, port } = server.address();
  console.log(`Server is listening on: http://${address}:${port}`);
})
```

The .createServer() method takes a single argument in the form of a callback function. This callback function has two primary arguments; the request (commonly written as req) and the response (commonly written as res).

The req object contains all of the information about an HTTP request ingested by the server. It exposes information such as the HTTP method (GET,POST, etc.), the pathname, headers, body, and so on. The res object contains methods and properties pertaining to the generation of a response by the HTTP server. This object contains methods such as .setHeader (sets HTTP headers on the response), .statusCode (set the status code of the response), and .end() (dispatches the response to the client who made the request). In the example above, we use the .end() method to send the string 'Server is Running!' to the client, which will display on the web page.

Once the .createServer() method has instantiated the server, it must begin listening for connections. This final step is accomplished by the .listen() method on the server instance. This method takes a port number as the first argument, which tells the server to listen for connections at the given port number. In our example above, the server has been set to listen on port 8080. Additionally, the .listen() method takes an optional callback function as a second argument, allowing it to carry out a task after the server has successfully started.

HTTP Module: Some Important Methods

http.STATUS_CODES;	A collection of all the standard HTTP response status codes, and the short description of each.
http.request(options, [callback]);	This function allows one to transparently issue requests.
http.get(options, [callback]);	Set the method to GET and calls req.end() automatically.
server = http.createServer([requestListener]);	Returns a new web server object. The requestListener is a function which is automatically added to the 'request' event.
server.listen(port, [hostname], [backlog], [callback]);	Begin accepting connections on the specified port and hostname.
server.close([callback]);	Stops the server from accepting new connections.
request.write(chunk, [encoding]);	Sends a chunk of the body.
request.end([data], [encoding]);	Finishes sending the request. If any parts of the body are unsent, it will flush them to the stream.
request.abort();	Aborts a request.
response.write(chunk, [encoding]);	This sends a chunk of the response body. If this merthod is called and response.writeHead() has

	not been called, it will switch to implicit header mode and flush the implicit headers.
response.writeHead(statusCode, [reasonPhrase], [headers]);	Sends a response header to the request.
response.getHeader(name);	Reads out a header that's already been queued but not sent to the client. Note that the name is case insensitive.
response.removeHeader(name);	Removes a header that's queued for implicit sending.
response.end([data], [encoding]);	This method signals to the server that all of the response headers and body have been sent; that server should consider this message complete. The method, response.end(), MUST be called on each response.

Handling HTTP Requests

In the following section we will learn to handle HTTP request when we open http://localhost:9000 url.

The callback function that we saw above when we created the server has two parameters namely, request and response.

The request object has url property and it holds the requested url. So, we will use it.

The response object has writeHead method which helps in setting the header. Then we have the write method that helps to write the response body. And there is end method which helps in sending the response.

Lets say, we want to print out the message Hello World! when we visit the home page at http://localhost:9000. As, it is the home page so we have to check for /.

```
const http = require('http');
const PORT = 9000;
const server = http.createServer(function(request, response) {
   // we are setting the header
   const header = {
       'Content-Type': 'text/html'
   };
   // checking for home page url /
   if (request.url === '/') {
       response.writeHead(200, header);
       response.write('Hello World!');
   }
   // if requested url is not known then prompt error response
   else {
       response.writeHead(400, header);
       response.write('Bad request!');
   }
   response.end();
});
server.listen(PORT);
console.log(`Server started and listening at port ${PORT}...`);
```

Now, stop the running server in the terminal and re-run it. Now, visit the home page http://localhost:9000 url. This time we will see the message Hello World!

This happens because the if condition is satisfied and if-block is executed.

Now, if we visit something like http://localhost:9000/some-unknown-page then we will get Bad Request message.

This happens because we are not handling the /some-unknown-page and hence the else-block is executed which gives the Bad request! message as response.

Making HTTP Requests

The request() method supports multiple function signatures. You'll use this one for the subsequent example:

```
http.request(Options_Object, Callback_Function)
```

The first argument is a string with the API endpoint. The second argument is a JavaScript object containing all the options for the request. The last argument is a callback function to handle the response.

Making HTTP Requests: Options

host	A domain name or IP address of the server to issue the request to. Defaults to 'localhost'.
hostname	To support url.parse() hostname is preferred over host
port	Port of the remote server. Defaults to 80.
method	A string specifying the HTTP request method. Defaults to 'GET'.
path	Request path. Defaults to '/'. Should include query string if any. E.G. '/index.html?page=12'
headers	An object containing request headers.
auth	Basic authentication i.e. 'user:password' to compute an Authorization header.

```
const https = require('https')
const options = {
  hostname: 'example.com',
  port: 443,
  path: '/todos',
  method: 'GET'
}

const req = https.request(options, res => {
  console.log(`statusCode: ${res.statusCode}`)

 res.on('data', d => {
    process.stdout.write(d)
  })
})

req.on('error', error => {
  console.error(error)
})

req.end()
```

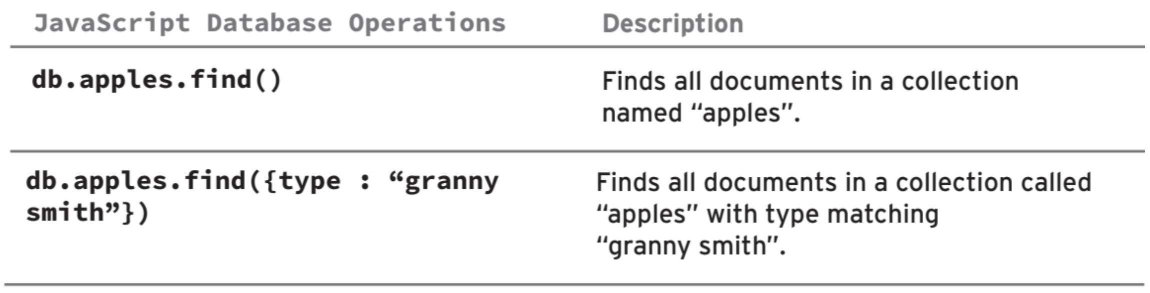
MongoDB and Its Features

MongoDB is a document-oriented NoSQL database used for high volume data storage. Instead of using tables and rows as in the traditional relational databases, MongoDB makes use of collections and documents. Documents consist of key-value pairs which are the basic unit of data in MongoDB. Collections contain sets of documents and function which is the equivalent of relational database tables. MongoDB is a database which came into light around the mid-2000s

1. Each database contains collections which in turn contains documents. Each document can be different with a varying number of fields. The size and content of each document can be different from each other.
2. The document structure is more in line with how developers construct their classes and objects in their respective programming languages. Developers will often say that their classes are not rows and columns but have a clear structure with key-value pairs.
3. The rows (or documents as called in MongoDB) doesn't need to have a schema defined beforehand. Instead, the fields can be created on the fly.
4. The data model available within MongoDB allows you to represent hierarchical relationships, to store arrays, and other more complex structures more easily.
5. Scalability – The MongoDB environments are very scalable. Companies across the world have defined clusters with some of them running 100+ nodes with around millions of documents within the database

Working with a Collection

You can perform operations using the `db` variable. Every collection has a property on the `db` variable - e.g. the `apples` collection is `db.apples`.



`JavaScript Database Operations`	Description
`db.apples.find()`	Finds all documents in a collection named "apples".
`db.apples.find({type : "granny smith"})`	Finds all documents in a collection called "apples" with type matching "granny smith".

`db.apples.remove({ bad : true})`	Removes all bad apples! Finds all apples with a property of "bad" set to true, and removes them.
`db.apples.update({ type : "granny smith"}, {$set : { price : 2.99 }})`	Updates the price of all granny smith apples.
`#don't do this! db.apples.update({ type : "granny smith"}, { price : 2.99 })`	**Important!** Note the $set syntax - this doesn't work as many would expect. Without putting the updated fields inside a $set clause, we replace the entire document.
`db.apples.insert({ type : "granny smith", "price" : 5.99 })`	Inserts a new document into the apples collection. If your apples collection doesn't exist, it will get created.
`db.apples.findOne({ _id : ObjectId("54324a5925859afb491a0000") })`	Looking up a document by ID is special. We can't just specify the ObjectID as a string - we need to cast it to an ObjectId.

MongoDB Connectivity: Create a Database

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist, and make a connection to it.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/mydb";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  console.log("Database created!");
  db.close();
});
```

MongoDB Connectivity: Create a Collection

To create a collection in MongoDB, use the createCollection() method:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";
```

```
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.createCollection("customers", function(err, res) {
    if (err) throw err;
    console.log("Collection created!");
    db.close();
  });
});
```

MongoDB Connectivity: Insert into Collection

To insert a record, or *document* as it is called in MongoDB, into a collection, we use the insertOne() method.

A **document** in MongoDB is the same as a **record** in MySQL

The first parameter of the insertOne() method is an object containing the name(s) and value(s) of each field in the document you want to insert.

It also takes a callback function where you can work with any errors, or the result of the insertion:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var myobj = { name: "Company Inc", address: "Highway 37" };
  dbo.collection("customers").insertOne(myobj, function(err, res) {
    if (err) throw err;
    console.log("1 document inserted");
    db.close();
  });
});
```

MongoDB Connectivity: Read

To select data from a table in MongoDB, we can also use the find() method. The find() method returns all occurrences in the selection. The first parameter of the find() method is a query object. In this example we use an empty query object, which selects all documents in the collection.

No parameters in the find() method gives you the same result as **SELECT *** in MySQL.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  dbo.collection("customers").find({}).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

MongoDB Connectivity: Filter results

When finding documents in a collection, you can filter the result by using a query object. The first argument of the find() method is a query object, and is used to limit the search.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb");
  var query = { address: "Park Lane 38" };
  dbo.collection("customers").find(query).toArray(function(err, result) {
    if (err) throw err;
    console.log(result);
    db.close();
  });
});
```

HTTP and MongoDB Connectivity

Node.js can be used in database applications.

One of the most popular NoSQL database is MongoDB.

Install the MongoDB Node.js Driver

The MongoDB Node.js Driver allows you to easily interact with MongoDB databases from within Node.js applications. You'll need the driver in order to connect to your database and execute the queries described in this Quick Start series.

If you don't have the MongoDB Node.js Driver installed, you can install it with the following command.

```
npm install mongodb
```

Introduction to mongodb.connect

This method is used to connect the Mongo DB server with our Node application. This is an asynchronous method from MongoDB module.

Syntax

mongodb.connect(path[, callback])

Parameters

•path – The server path where the MongoDB server is actually running along with its port.

•callback – This function will give a callback if any error occurs.

MongoDB

To be able to experiment with the code examples, you will need access to a MongoDB database.

You can download a free MongoDB database at https://www.mongodb.com.

Node.js can use this module to manipulate MongoDB databases:

```
var mongo = require('mongodb');
```

Creating a Database

To create a database in MongoDB, start by creating a MongoClient object, then specify a connection URL with the correct ip address and the name of the database you want to create.

MongoDB will create the database if it does not exist, and make a connection to it.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/pes";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  console.log("Database created!");
  db.close();
});
```

In the above example, we have imported mongodb module (native drivers) and got the reference of MongoClient object. Then we used MongoClient.connect() method to get the reference of specified MongoDB database. The specified URL "mongodb://localhost:27017/pes" points to your local MongoDB database created in MyMongoDB folder. The connect() method returns the database reference if the specified database is already exists, otherwise it creates a new database.

In MongoDB, a database is not created until it gets content!

MongoDB waits until you have created a collection (table), with at least one document (record) before it actually creates the database (and collection).

A collection in MongoDB is the same as a table in MySQL

Creating a Collection

To create a collection in MongoDB, use the `Collection()` method:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("pes");
  dbo.createCollection("students", function(err, res) {
    if (err) throw err;
    console.log("Collection created!");
    db.close();
  });
});
```

In MongoDB, a collection is not created until it gets content.

MongoDB waits until you have inserted a document before it actually creates the collection.

Insert a single document Into Collection

To insert a record, or *document* as it is called in MongoDB, into a collection, we use the insertOne() method.

A document in MongoDB is the same as a record in MySQL

The first parameter of the insertOne() method is an object containing the name(s) and value(s) of each field in the document you want to insert.

It also takes a callback function where you can work with any errors, or the result of the insertion:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("pes");
  var myobj = { name: "Ria", id: "001" };
  dbo.collection("students").insertOne(myobj, function(err, res) {
    if (err) throw err;
    console.log("1 document inserted");
    db.close();
  });
});
```

Insert Multiple Documents into a collection

To insert multiple documents into a collection in MongoDB, we use the insertMany() method.

The first parameter of the insertMany() method is an array of objects, containing the data you want to insert.

It also takes a callback function where you can work with any errors, or the result of the insertion:

```
Insert multiple documents in the "customers" collection:
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
```

```
 var dbo = db.db("pes");
 var myobj = [
  { name: Ria, id: '001'},
  { name: 'Arun', address: '002'},
  { name: 'Vivek', address: '003'},

 ];
 dbo.collection("students").insertMany(myobj, function(err, res) {
  if (err) throw err;
  console.log("Number of documents inserted: " + res.insertedCount);
  db.close();
 });
});
```

The _id Field

If you do not specify an _id field, then MongoDB will add one for you and assign a unique id for each document.

MongoDB assigns a unique _id for each document. The value must be unique for each document:

Select the documents from collection:

In MongoDB we use the **find** and **findOne** methods to find data in a collection.

Just like the **SELECT** statement is used to find data in a table in a MySQL database.

Find:

To select data from a table in MongoDB, we can also use the find() method.

The find() method returns all occurrences in the selection.

The first parameter of the find() method is a query object.

```
Find all documents in the customers collection:

var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
 if (err) throw err;
 var dbo = db.db("pes");
 dbo.collection("students").find({}).toArray(function(err, result) {
  if (err) throw err;
  console.log(result);
  db.close();
 });
});
```

<u>Examples:</u>

Example 1 – Create Collection in MongoDB via Node.js

In this example, we will connect to a MongoDB Database and create a collection named “users”.

node-js-mongodb-create-collection.js

```
// we create 'users' collection in newdb database
var url = "mongodb://localhost:27017/newdb";

// create a client to mongodb
var MongoClient = require('mongodb').MongoClient;

// make client connect to mongo service
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  // db pointing to newdb
  console.log("Switched to "+db.databaseName+" database");
  // create 'users' collection in newdb database
  db.createCollection("users", function(err, result) {
    if (err) throw err;
    console.log("Collection is created!");
    // close the connection to db when you are done with it
    db.close();
  });
});
```

Output

```
nodejs/mongodb$ node node-js-mongodb-create-collection.js
Switched to newdb database
Collection is created!
```

Example 2 – Delete Collection in MongoDB using Node.js

In this example, we will delete a MongoDB Collection using remove() method.

node-js-mongodb-delete-collection.js

```
// example : delete 'users' collection in newdb database
var url = "mongodb://localhost:27017/newdb";

// create a client to mongodb
var MongoClient = require('mongodb').MongoClient;

// make client connect to mongo service
MongoClient.connect(url, function(err, db) {
   if (err) throw err;
   // db pointing to newdb
   console.log("Switched to "+db.databaseName+" database");
   // get reference to collection
   db.collection("users", function(err, collection) {
      // handle the error if any
      if (err) throw err;
      // delete the mongodb collection
      collection.remove({}, function(err, result){
         // handle the error if any
         if (err) throw err;
         console.log("Collection is deleted! "+result);
         // close the connection to db when you are done with it
         db.close();
      });
   });
});
```

Output

```
$ node node-js-mongodb-delete-collection.js
Switched to newdb database
Collection is deleted! {"n":0,"ok":1}
```

Example 3 – insertOne() – Insert Document via Node.js

In this example, we will use insertOne() method and insert a document to MongoDB Collection via Node.js program.

node-js-mongodb-insert-document.js

```
// we create 'users' collection in newdb database
var url = "mongodb://localhost:27017/newdb";

// create a client to mongodb
var MongoClient = require('mongodb').MongoClient;

// make client connect to mongo service
```

```
MongoClient.connect(url, function(err, db) {
   if (err) throw err;
   // db pointing to newdb
   console.log("Switched to "+db.databaseName+" database");

   // document to be inserted
   var doc = { name: "Roshan", age: "22" };

   // insert document to 'users' collection using insertOne
   db.collection("users").insertOne(doc, function(err, res) {
      if (err) throw err;
      console.log("Document inserted");
      // close the connection to db when you are done with it
      db.close();
   });
});
```

Output

```
$ node node-js-mongodb-insert-document.js
Switched to newdb database
Document inserted
```

Example 4 – insertMany() – Insert Many Documents via Node.js

In this example, we will use insertMany() method and insert multiple documents to MongoDB Collection via Node.js program.

node-js-mongodb-insert-many-documents.js

```
// we create 'users' collection in newdb database
var url = "mongodb://localhost:27017/newdb";

// create a client to mongodb
var MongoClient = require('mongodb').MongoClient;

// make client connect to mongo service
MongoClient.connect(url, function(err, db) {
   if (err) throw err;
   // db pointing to newdb
   console.log("Switched to "+db.databaseName+" database");

   // documents to be inserted
   var docs = [{ name: "Udat", age: "21" },
            { name: "Karthik", age: "24" },
            { name: "Anil", age: "23" }];
```

```
    // insert multiple documents to 'users' collection using insertOne
    db.collection("users").insertMany(docs, function(err, res) {
        if (err) throw err;
        console.log(res.insertedCount+" documents inserted");
        // close the connection to db when you are done with it
        db.close();
    });
});
```

Output

```
$ node node-js-mongodb-insert-many-documents.js
Switched to newdb database
3 documents inserted
```

References:

1. https://www.tutorialkart.com/nodejs/node-js-mongodb/
2. https://www.tutorialsteacher.com/nodejs

Event Emitter Module

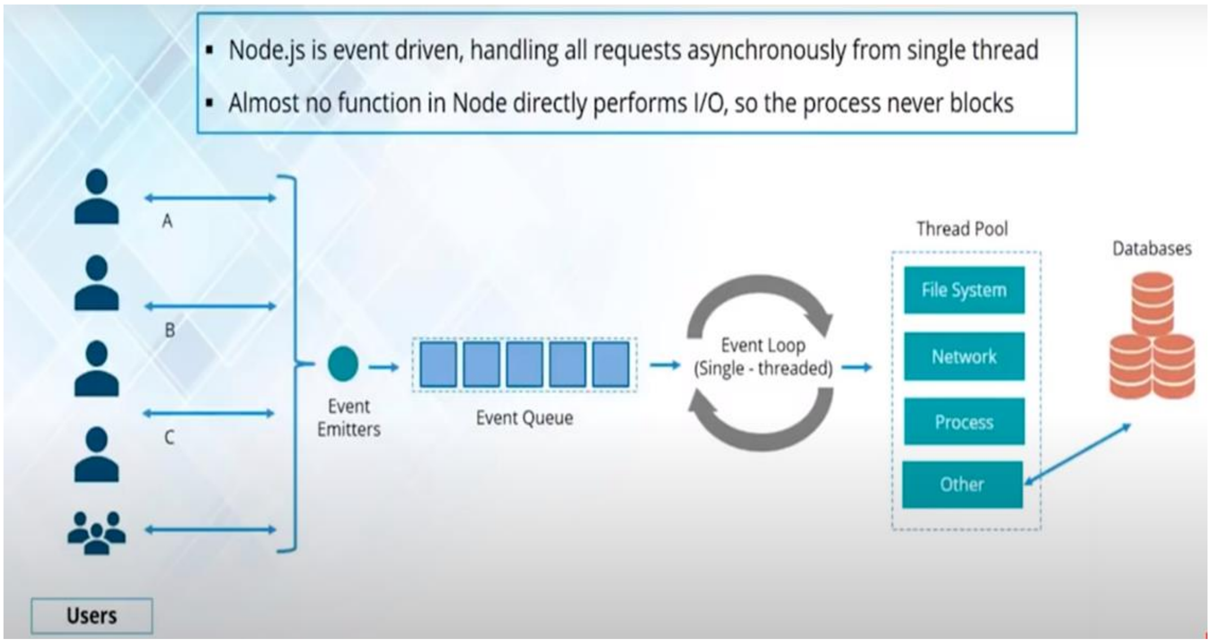
Event Loop

Node.js is a single-threaded application, but it can support concurrency via the concept of **event** and **callbacks**. Every API of Node.js is asynchronous and being single-threaded, they use **async function calls** to maintain concurrency. Node uses observer pattern. Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

Event-Driven Programming

Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies. As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.

In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.



Node.js is event driven, handling all requests asynchronously from single thread
Almost no function in Node directly performs I/O, so the process never blocks
A
B
C
Users
Event Emitters
Event Queue
Event Loop (Single - threaded)
Thread Pool
File System
Network
Process
Other
Databases

Although events look quite similar to callbacks, the difference lies in the fact that callback functions are called when an asynchronous function returns its result, whereas event handling works on the observer pattern. The functions that listen to events act as **Observers**. Whenever an event gets fired, its listener function starts executing. Node.js has multiple in-built events available through events module and EventEmitter class which are used to bind events and event-listeners as follows –

```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
Following is the syntax to bind an event handler with an event –
// Bind event and event  handler as follows
```

```
eventEmitter.on('eventName', eventHandler);
We can fire an event programmatically as follows –
// Fire an event
eventEmitter.emit('eventName');
```

Event Emitter Class:

Node.js allows us to create and handle custom events easily by using events module. Event module includes EventEmitter class which can be used to raise and handle custom events.

Many objects in a Node emit events, for example, a net.Server emits an event each time a peer connects to it, an fs.readStream emits an event when the file is opened. All objects which emit events are the instances of events.EventEmitter.

EventEmitter class lies in the events module. It is accessible via the following code –

```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
```

When an EventEmitter instance faces any error, it emits an 'error' event. When a new listener is added, 'newListener' event is fired and when a listener is removed, 'removeListener' event is fired.

EventEmitter provides multiple properties like **on** and **emit**. **on** property is used to bind a function with the event and **emit** is used to fire an event.

Example: Raise and Handle Node.js events

```
// get the reference of EventEmitter class of events module

var events = require('events');


//create an object of EventEmitter class by using above reference

var em = new events.EventEmitter();


//Subscribe for FirstEvent

em.on('FirstEvent', function (data) {

   console.log('First subscriber: ' + data);

});
```

```
// Raising FirstEvent
em.emit('FirstEvent', 'This is my first Node.js event emitter example.');
```

In the above example, we first import the 'events' module and then create an object of EventEmitter class. We then specify event handler function using on() function. The on() method requires name of the event to handle and callback function which is called when an event is raised.

The emit() function raises the specified event. First parameter is name of the event as a string and then arguments. An event can be emitted with zero or more arguments. You can specify any name for a custom event in the emit() function.

Methods

Sr.No.	Method & Description
1	**addListener(event, listener)** Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained.
2	**on(event, listener)** Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained.
3	**once(event, listener)** Adds a one time listener to the event. This listener is invoked only the next time the event is fired, after which it is removed. Returns emitter, so calls can be chained.
4	**removeListener(event, listener)** Removes a listener from the listener array for the specified event. **Caution –** It changes the array indices in the listener array behind the listener. removeListener will remove, at most, one instance of a listener from the listener array. If any single listener has been added multiple times to the listener array for the specified event, then removeListener must be called multiple times to remove each instance. Returns emitter, so calls can be chained.

5	**removeAllListeners([event])** Removes all listeners, or those of the specified event. It's not a good idea to remove listeners that were added elsewhere in the code, especially when it's on an emitter that you didn't create (e.g. sockets or file streams). Returns emitter, so calls can be chained.
6	**setMaxListeners(n)** By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event. This is a useful default which helps finding memory leaks. Obviously not all Emitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited.
7	**listeners(event)** Returns an array of listeners for the specified event.
8	**emit(event, [arg1], [arg2], [...])** Execute each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise.

Class Methods

Sr.No.	Method & Description
1	**listenerCount(emitter, event)** Returns the number of listeners for a given event.

Events

Sr.No.	Events & Description
1	**newListener** • **event** − String: the event name • **listener** − Function: the event handler function This event is emitted any time a listener is added. When this event is triggered, the listener may not yet have been added to the array of listeners for the event.
2	**removeListener** • **event** − String The event name

	• **listener** – Function The event handler function This event is emitted any time someone removes a listener. When this event is triggered, the listener may not yet have been removed from the array of listeners for the event.

Common Patterns for EventEmitters:

There are two common patterns that can be used to raise and bind an event using EventEmitter class in Node.js.

1. Return EventEmitter from a function
2. Extend the EventEmitter class

Return EventEmitter from a function

In this pattern, a constructor function returns an EventEmitter object, which was used to emit events inside a function. This EventEmitter object can be used to subscribe for the events.

Extend EventEmitter Class

In this pattern, we can extend the constructor function from EventEmitter class to emit the events so that you can use EventEmitter class to raise and handle custom events in Node.js.

Example:

```
// Import events module
var events = require('events');
// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();

//create an event handler
var handler = function connected()
{
   console.log('connection successful');
}
eventEmitter.on('connection',handler);
eventEmitter.on('received_data',function()
{
   console.log('received the data!!');
});

//Fire the event
eventEmitter.emit('connection');
eventEmitter.emit('received_data');
```

References:

1. https://www.tutorialsteacher.com/nodejs/nodejs-eventemitter

React Router

Routing is a process in which a user is directed to different pages based on their action or request. ReactJS Router is mainly used for developing Single Page Web Applications. React Router is used to define multiple routes in the application. When a user types a specific URL into the browser, and if this URL path matches any 'route' inside the router file, the user will be redirected to that particular route.

React Router is a standard library system built on top of the React and used to create routing in the React application using React Router Package. It provides the synchronous URL on the browser with data that will be displayed on the web page. It maintains the standard structure and behavior of the application and mainly used for developing single page web applications.

Need of React Router

React Router plays an important role to display multiple views in a single page application. Without React Router, it is not possible to display multiple views in React applications. Most of the social media websites like Facebook, Instagram uses React Router for rendering multiple views.

Components in React Router

There are two types of router components:

- <BrowserRouter>: It is used for handling the dynamic URL.
- <HashRouter>: It is used for handling the static request.

BrowserRouter

React Router needs to be both aware and in control of your app's location. The way it does this is with its BrowserRouter component. BrowserRouter uses both the history library as well as React Context. The history library helps React Router keep track of the browsing history of the application using the browser's built-in history stack, and React Context helps make history available wherever React Router needs it.

BrowserRouter, makes sure that if you're using React Router on the web, you wrap your app inside of the BrowserRouter component.

```
import ReactDOM from 'react-dom'
import * as React from 'react'
import { BrowserRouter } from 'react-router-dom'
import App from './App`

ReactDOM.render(
  <BrowserRouter>
```

```
        <App />
      </BrowserRouter>
    , document.getElementById('app))
```

Route

Route allows you to map your app’s location to different React components. For example, say we wanted to render a Dashboard component whenever a user navigated to the /dashboard path. To do so, we’d render a Route that looked like this.

```
<Route path='/dashboard' element={<Dashboard />} />
```

You can render as many Routes as you’d like.

```
<Route path="/" element={<Home />} />
<Route path="/about" element={<About />} />
<Route path="/settings" element={<Settings />} />
```

With our Route elements in this configuration, it’s possible for multiple routes to match on a single URL. You might want to do that sometimes, but most often you want React Router to only render the route that matches best. Fortunately, we can easily do that with Routes.

Routes

Whenever you have one or more Routes, you’ll most likely want to wrap them in a Routes.

```
import {
  Routes,
  Route
} from 'react-router-dom'

function App () {
  return (
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
      <Route path="/settings" element={<Settings />} />
      <Route path="*" element={<NotFound />} />
    </Routes>
  )
}
```

The reason for this is because it’s Routes job is to understand all of its children Route elements, and intelligently choose which ones are the best to render. Once we start adding more complex Routes to our application, Routes will start to do more work like enabling intelligent rendering and relative paths.

Link
The next step is being able to navigate between them. This is the purpose of the Link component.

To tell Link what path to take the user to when clicked, you pass it a to prop.

```
<nav>
  <Link to="/">Home</Link>
  <Link to='/about'>About</Link>
  <Link to="/settings">Settings</Link>
</nav>
```

If you need more control over Link, you can also pass to as an object. Doing so allows you to add a query string via the search property or pass along any data to the new route via state.

```
<nav>
  <Link to='/'>Home</Link>
  <Link to='/about'>About</Link>
  <Link to={{
    pathname: '/settings',
    search: '?sort=date',
    state: { fromHome: true },
  }}>Settings</Link>
</nav>
```

We'll cover state, Query Strings, and how React Router supports relative paths in more depth later on in this post.

At this point we've covered both the history and the absolute fundamentals of React Router, but one thing should already be clear - by embracing composition, React Router is truly a router for React. I believe React will make you a better JavaScript developer and React Router will make you a better React developer.

Now, instead of just walking you through the rest of the API, we'll take a more practical approach by breaking down all of the common use cases you'll need when using React Router.

URL Parameters
Like function parameters allow you to declare placeholders when you define a function, URL Parameters allow you to declare placeholders for portions of a URL. For example, When you visit a topic on Wikipedia, you'll notice that the URL pattern is always the same, wikipedia.com/wiki/{topicId}.

Instead of defining a route for every topic on the site, they can declare one route with a placeholder for the topic's id. The way you tell React Router that a certain portion of the URL is a placeholder (or URL Parameter), is by using a : in the Route's path prop.

```
<Route path='/wiki/:topicId' element={<Article />} />
```

Example

Step-1: In our project, we will create two more components along with **App.js**, which is already present.

About.js

```
import React from 'react'
class About extends React.Component {
  render() {
    return <h1>About</h1>
  }
}
export default About
```

Contact.js

```
import React from 'react'
class Contact extends React.Component {
  render() {
    return <h1>Contact</h1>
  }
}
export default Contact
```

App.js

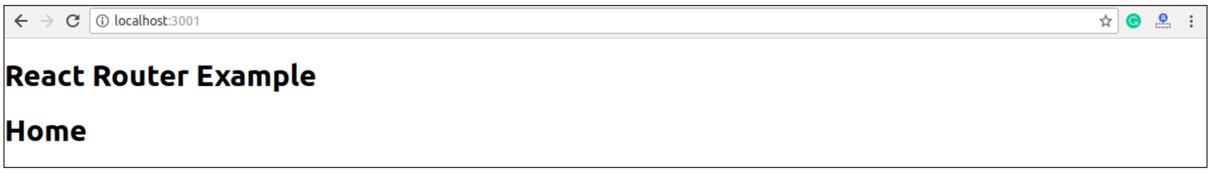
```
import React from 'react'
class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Home</h1>
      </div>
    )
  }
}
export default App
```

Step-2: For Routing, open the index.js file and import all the three component files in it. Here, you need to import line: **import { Route, Link, BrowserRouter as Router } from 'react-router-dom'** which helps us to implement the Routing. Now, our index.js file looks like below.

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Route, Link, BrowserRouter as Router } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'

const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <Route path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </div>
  </Router>
)
ReactDOM.render(routing, document.getElementById('root'));
```

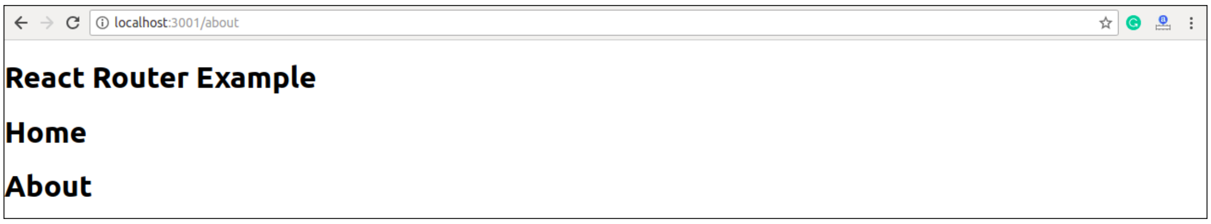
Step-3: Open **command prompt**, go to your project location, and then type **npm start**. You will get the following screen.



React Router Example

Home

Now, if you enter **manually** in the browser: **localhost:3000/about**, you will see **About** component is rendered on the screen.



React Router Example

Home

About

References:

1. https://ui.dev/react-router-tutorial/#routes
2. https://www.javatpoint.com/react-router

React Router

Routing is a process in which a user is directed to different pages based on their action or request. ReactJS Router is mainly used for developing Single Page Web Applications. React Router is used to define multiple routes in the application. When a user types a specific URL into the browser, and if this URL path matches any 'route' inside the router file, the user will be redirected to that particular route.

React Router is a standard library system built on top of the React and used to create routing in the React application using React Router Package. It provides the synchronous URL on the browser with data that will be displayed on the web page. It maintains the standard structure and behavior of the application and mainly used for developing single page web applications.

Need of React Router

React Router plays an important role to display multiple views in a single page application. Without React Router, it is not possible to display multiple views in React applications. Most of the social media websites like Facebook, Instagram uses React Router for rendering multiple views.

Components in React Router

There are two types of router components:

- <BrowserRouter>: It is used for handling the dynamic URL.
- <HashRouter>: It is used for handling the static request.

BrowserRouter

React Router needs to be both aware and in control of your app's location. The way it does this is with its BrowserRouter component. BrowserRouter uses both the history library as well as React Context. The history library helps React Router keep track of the browsing history of the application using the browser's built-in history stack, and React Context helps make history available wherever React Router needs it.

BrowserRouter, makes sure that if you're using React Router on the web, you wrap your app inside of the BrowserRouter component.

```
import ReactDOM from 'react-dom'
import * as React from 'react'
import { BrowserRouter } from 'react-router-dom'
import App from './App`

ReactDOM.render(
  <BrowserRouter>
```

```
        <App />
      </BrowserRouter>
    , document.getElementById('app))
```

Route

Route allows you to map your app's location to different React components. For example, say we wanted to render a Dashboard component whenever a user navigated to the /dashboard path. To do so, we'd render a Route that looked like this.

```
<Route path='/dashboard' element={<Dashboard />} />
```

You can render as many Routes as you'd like.

```
<Route path="/" element={<Home />} />
<Route path="/about" element={<About />} />
<Route path="/settings" element={<Settings />} />
```

With our Route elements in this configuration, it's possible for multiple routes to match on a single URL. You might want to do that sometimes, but most often you want React Router to only render the route that matches best. Fortunately, we can easily do that with Routes.

Routes

Whenever you have one or more Routes, you'll most likely want to wrap them in a Routes.

```
import {
  Routes,
  Route
} from 'react-router-dom'

function App () {
  return (
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
      <Route path="/settings" element={<Settings />} />
      <Route path="*" element={<NotFound />} />
    </Routes>
  )
}
```

The reason for this is because it's Routes job is to understand all of its children Route elements, and intelligently choose which ones are the best to render. Once we start adding more complex Routes to our application, Routes will start to do more work like enabling intelligent rendering and relative paths.

Link
The next step is being able to navigate between them. This is the purpose of the Link component.

To tell Link what path to take the user to when clicked, you pass it a to prop.

```
<nav>
  <Link to="/">Home</Link>
  <Link to='/about'>About</Link>
  <Link to="/settings">Settings</Link>
</nav>
```

If you need more control over Link, you can also pass to as an object. Doing so allows you to add a query string via the search property or pass along any data to the new route via state.

```
<nav>
  <Link to='/'>Home</Link>
  <Link to='/about'>About</Link>
  <Link to={{
    pathname: '/settings',
    search: '?sort=date',
    state: { fromHome: true },
  }}>Settings</Link>
</nav>
```

We'll cover state, Query Strings, and how React Router supports relative paths in more depth later on in this post.

At this point we've covered both the history and the absolute fundamentals of React Router, but one thing should already be clear - by embracing composition, React Router is truly a router for React. I believe React will make you a better JavaScript developer and React Router will make you a better React developer.

Now, instead of just walking you through the rest of the API, we'll take a more practical approach by breaking down all of the common use cases you'll need when using React Router.

URL Parameters
Like function parameters allow you to declare placeholders when you define a function, URL Parameters allow you to declare placeholders for portions of a URL. For example, When you visit a topic on Wikipedia, you'll notice that the URL pattern is always the same, wikipedia.com/wiki/{topicId}.

Instead of defining a route for every topic on the site, they can declare one route with a placeholder for the topic's id. The way you tell React Router that a certain portion of the URL is a placeholder (or URL Parameter), is by using a : in the Route's path prop.

```
<Route path='/wiki/:topicId' element={<Article />} />
```

Example

Step-1: In our project, we will create two more components along with **App.js**, which is already present.

About.js

```
import React from 'react'
class About extends React.Component {
  render() {
    return <h1>About</h1>
  }
}
export default About
```

Contact.js

```
import React from 'react'
class Contact extends React.Component {
  render() {
    return <h1>Contact</h1>
  }
}
export default Contact
```

App.js

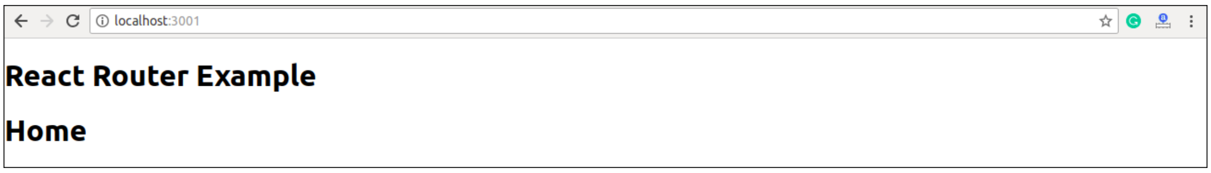
```
import React from 'react'
class App extends React.Component {
  render() {
    return (
      <div>
        <h1>Home</h1>
      </div>
    )
  }
}
export default App
```

Step-2: For Routing, open the index.js file and import all the three component files in it. Here, you need to import line: **import { Route, Link, BrowserRouter as Router } from 'react-router-dom'** which helps us to implement the Routing. Now, our index.js file looks like below.

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Route, Link, BrowserRouter as Router } from 'react-router-dom'
import './index.css';
import App from './App';
import About from './about'
import Contact from './contact'

const routing = (
  <Router>
    <div>
      <h1>React Router Example</h1>
      <Route path="/" component={App} />
      <Route path="/about" component={About} />
      <Route path="/contact" component={Contact} />
    </div>
  </Router>
)
ReactDOM.render(routing, document.getElementById('root'));
```

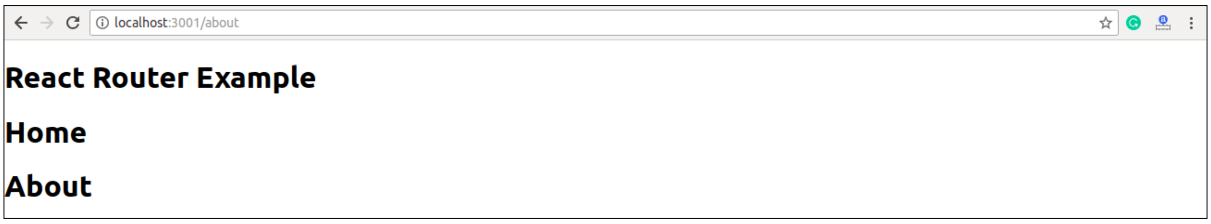
Step-3: Open **command prompt**, go to your project location, and then type **npm start**. You will get the following screen.



React Router Example

Home

Now, if you enter **manually** in the browser: **localhost:3000/about**, you will see **About** component is rendered on the screen.



React Router Example

Home

About

References:

1. https://ui.dev/react-router-tutorial/#routes
2. https://www.javatpoint.com/react-router