

Nodejs



Nodejs

- Node.js is an open-source and cross-platform JavaScript runtime environment.
- Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser.
- This allows Node.js to be very performant.
- A Node.js app runs in a single process, without creating a new thread for every request.
- It provides an event driven, non-blocking (asynchronous) I/O and cross-platform

- 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.
- Mainly used to build network programs like web servers, similar to PHP, Java, or ASP.NET.

Advantages of Node.js

- 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).)
- Uses JavaScript to build entire server side application.
- Lightweight framework that includes bare minimum modules.
- Other modules can be included as per the need of an application.
- Asynchronous by default. So it performs faster than other frameworks.
- Cross-platform framework that runs on

Features of Node.js

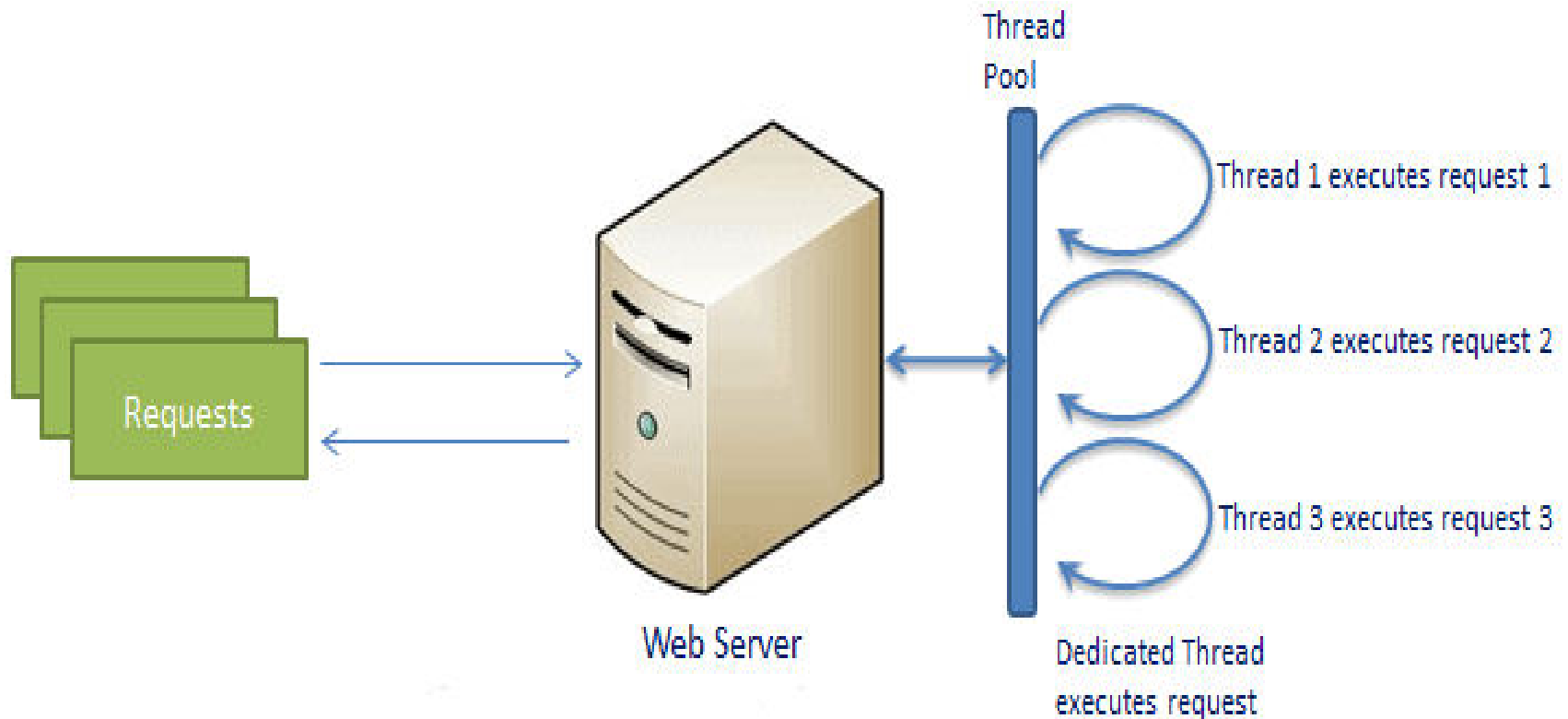
- **Extremely fast:** Node.js is built on Google Chrome's V8 JavaScript Engine, so its library is very fast in code execution.
- **I/O is Asynchronous and Event Driven:** All APIs of Node.js library are asynchronous i.e. non-blocking. So a Node.js based server never waits for an API to return data. The 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. It is also a reason that it is very fast.
- **Single threaded:** Node.js follows a single

- **Highly Scalable:** Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.
- **No buffering:** Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.
- **Open source:** Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications.

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.

Traditional Web Server Model



Node.js Process Model

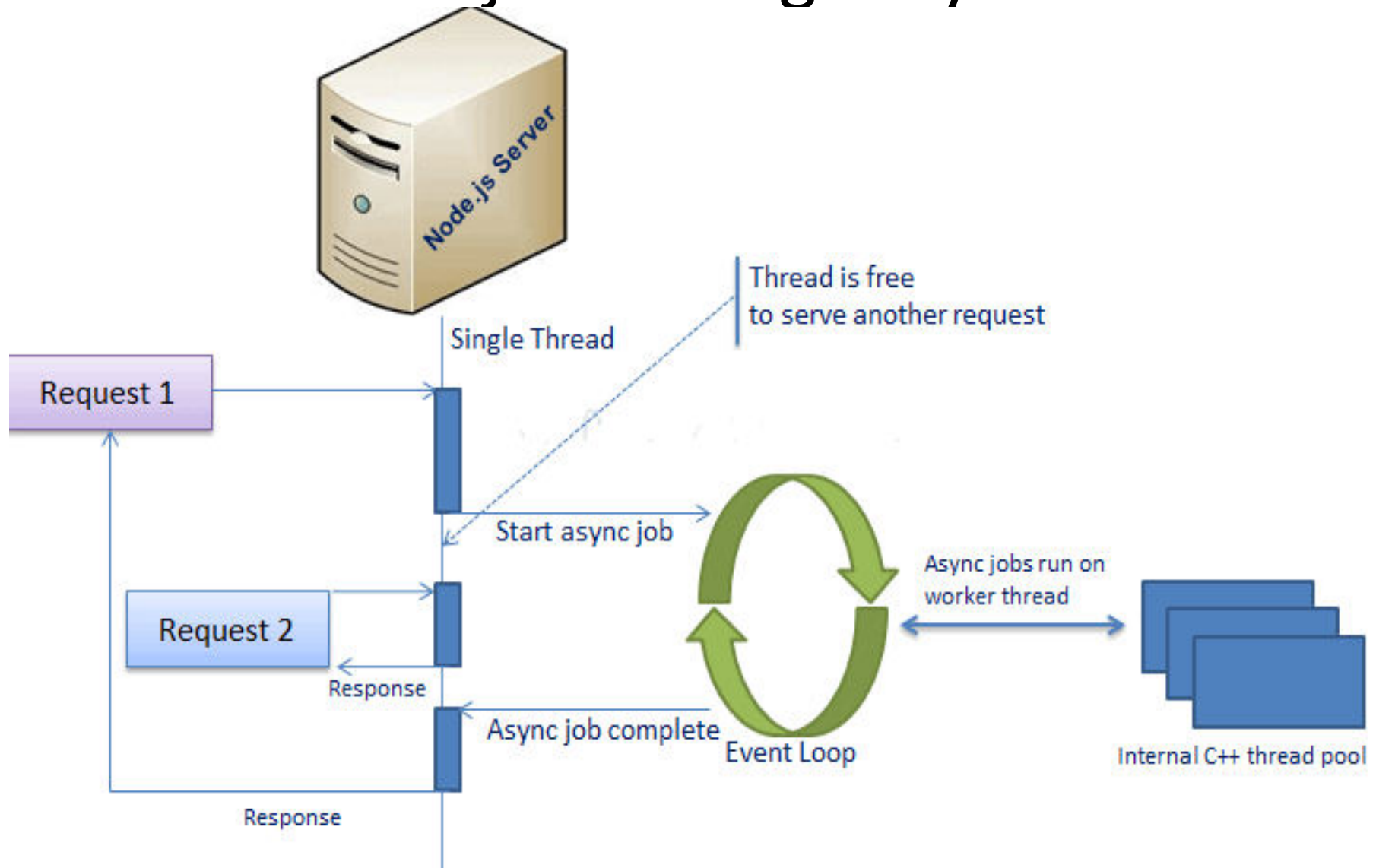
- Node.js processes user requests differently when compared to a traditional web server model.
- Node.js runs in a single process and the application code runs in a single thread and thereby needs less resources than other platforms.
- All the user requests to your web application will be handled by 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

- 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 libuv (Libuv is a multi-platform C library that provides support for asynchronous event loops.) for the event loop. Node.js then uses internal C++ APIs to provide asynchronous I/O.



libuv

Nodejs working- Async



Nodejs- Async

- When Node.js performs an I/O operation, like reading from the network, accessing a database or the filesystem, instead of blocking the thread and wasting CPU cycles waiting, Node.js will resume the operations when the response comes back.
- This allows Node.js to handle thousands of concurrent connections with a single server without introducing the burden of managing thread concurrency, which could be a significant source of bugs.

Nodejs

- Node.js has a unique advantage because millions of frontend developers that write JavaScript for the browser are now able to write the server-side code in addition to the client-side code without the need to learn a completely different language.
- In Node.js the new ECMAScript standards can be used without problems, as you don't have to wait for all your users to update their browsers - you are in charge of deciding which ECMAScript version to use by changing the Node.js version, and you can

Environment Setup

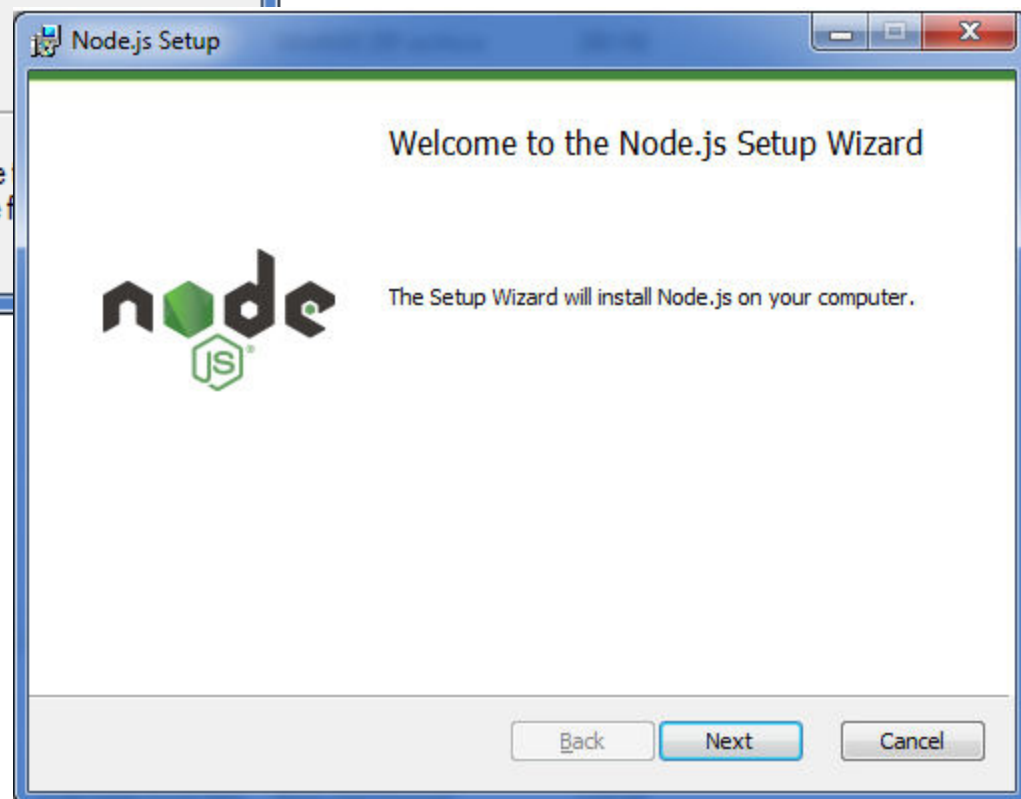
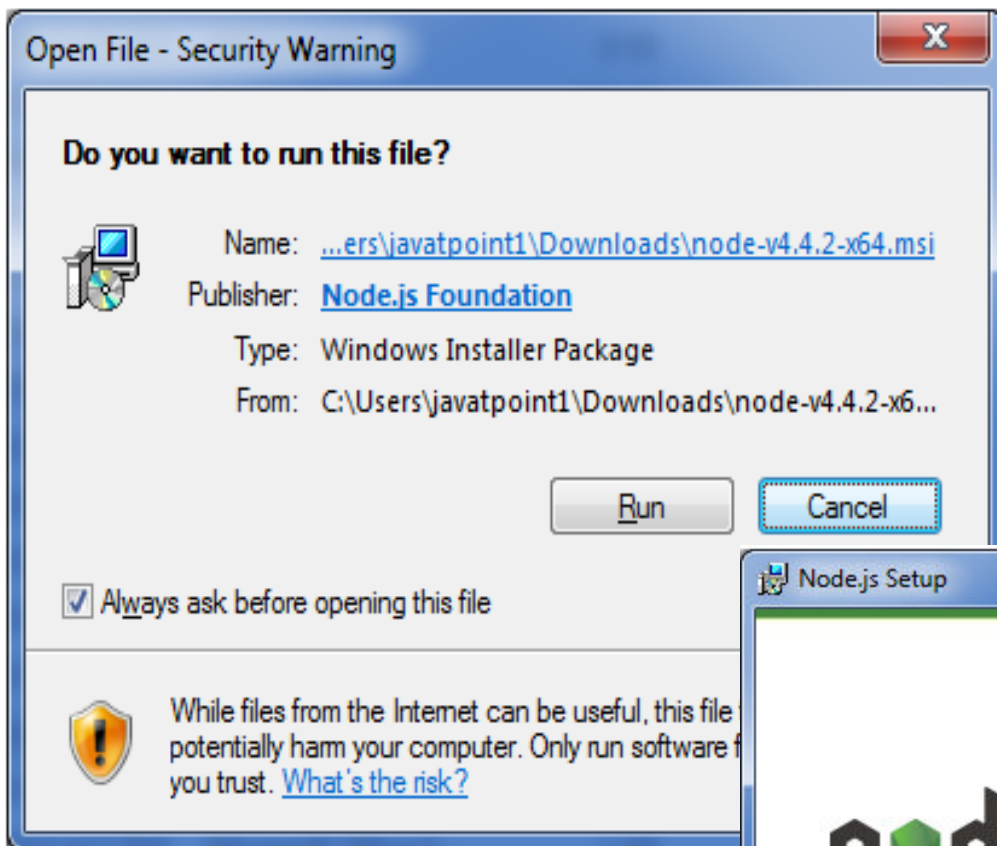
- Node.js development environment can be setup in Windows, Mac, Linux and Solaris.
- The following tools/SDK are required for developing a Node.js application on any platform.
- Node.js
- Node Package Manager (NPM)
- IDE (Integrated Development Environment) or TextEditor
- *NPM (Node Package Manager) is included in Node.js installation since Node version 0.6.0.,*

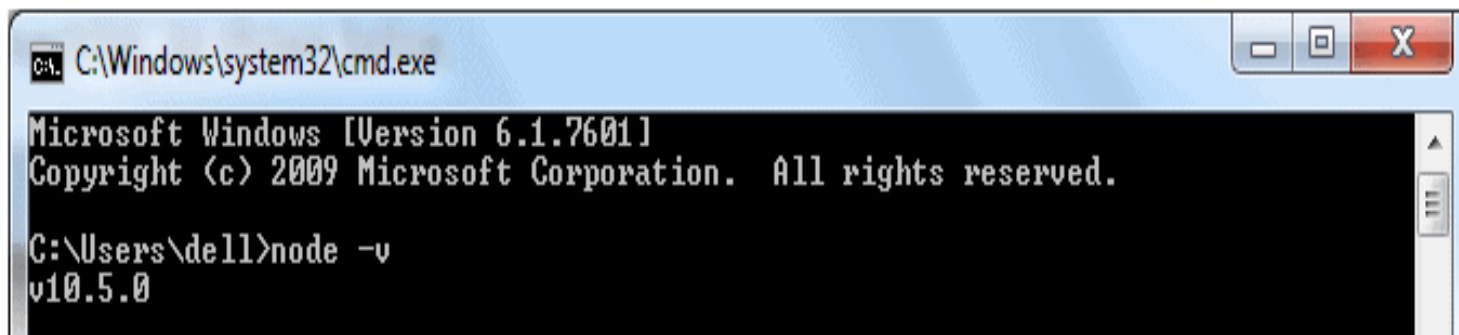
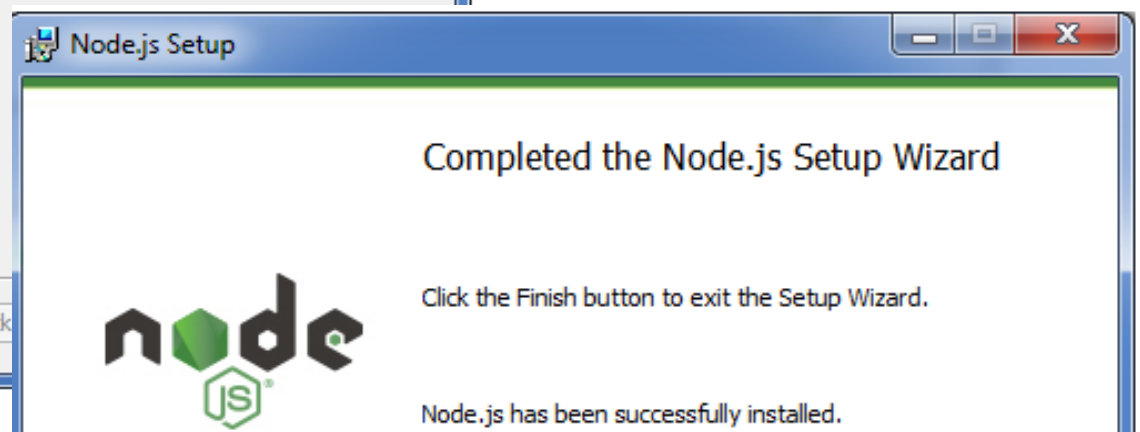
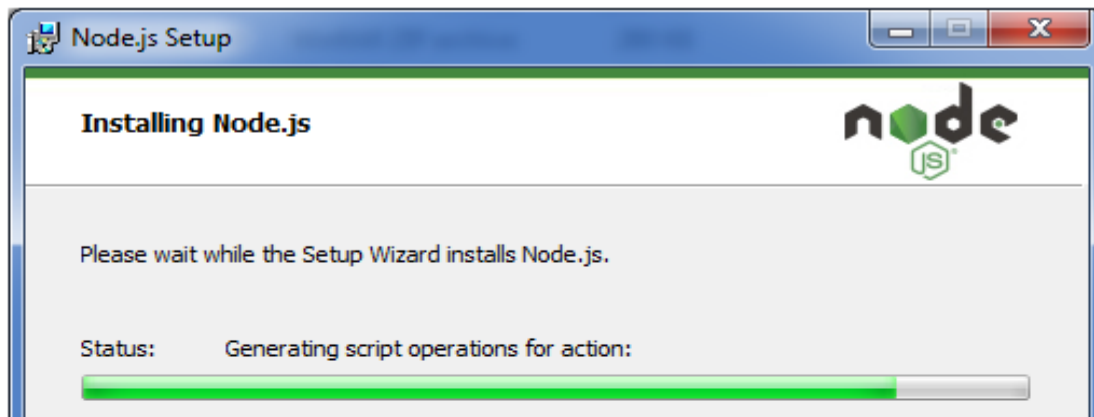
How to install Node.js

- Official packages for all the major platforms are available at
- <https://nodejs.org/en/download/>.
- One very convenient way to install Node.js is through a package manager.



Here, you deploy the installation of node-v4.4.2 LTS recommended for most users.





Verify Node.js Installation

Node.js web-based Example

- A node.js web application contains the following three parts:
- **Import required modules:** The "require" directive is used to load a Node.js module.
- **Create server:** You have to establish a server which will listen to client's request similar to Apache HTTP Server.
- **Read request and return response:** Server created in the second step will read HTTP request made by client which can be a browser or console and return the response

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

```
http.createServer(function (request, response) {
```

```
    // Send the HTTP header
```

```
    // HTTP Status: 200 : OK
```

```
    // Content Type: text/plain
```

```
    response.writeHead(200, {'Content-Type': 'text/plain'});
```

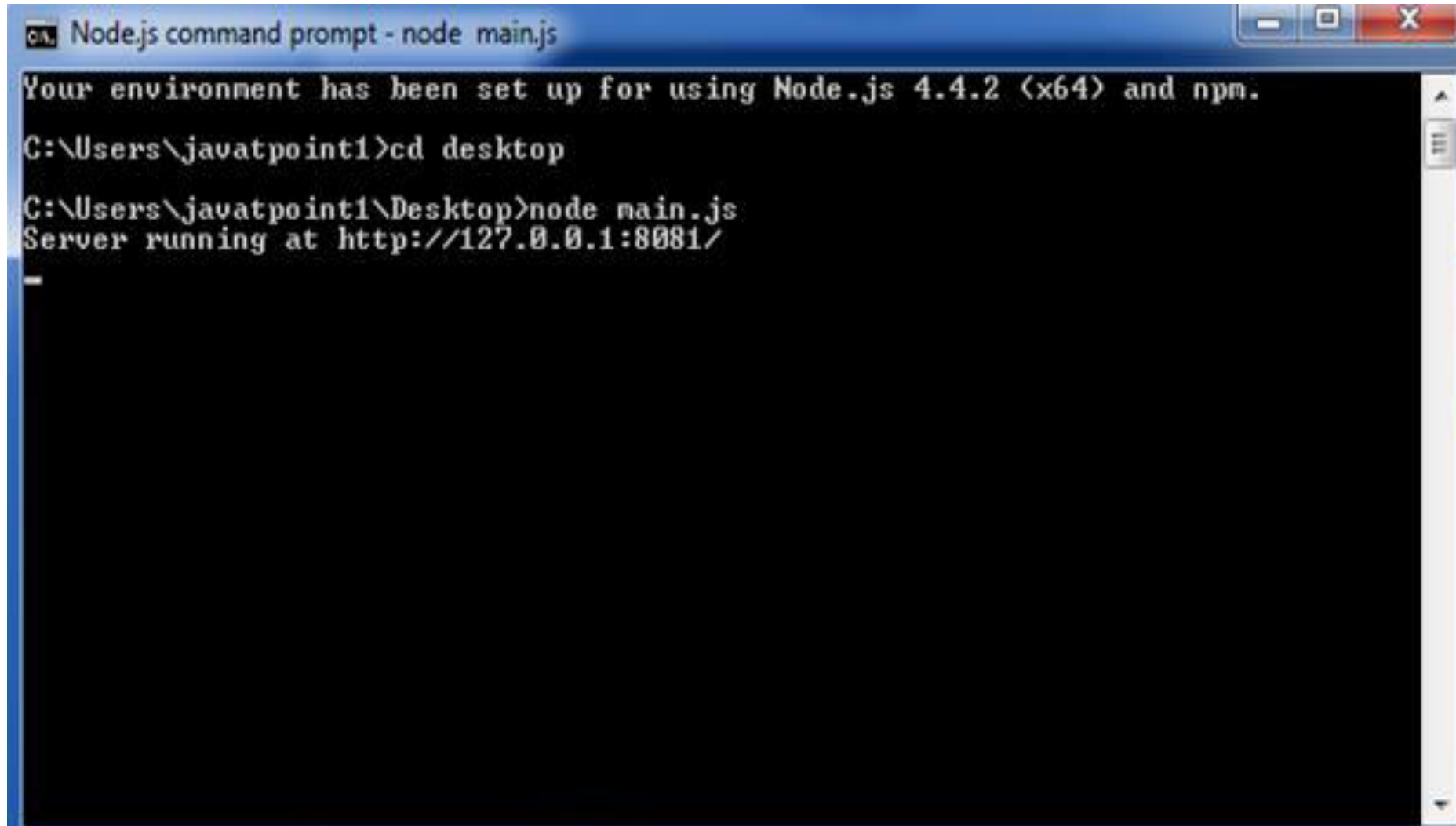
```
    // Send the response body as "Hello World"
```

```
    response.end('Hello World\n');
```

```
}).listen(8081);
```

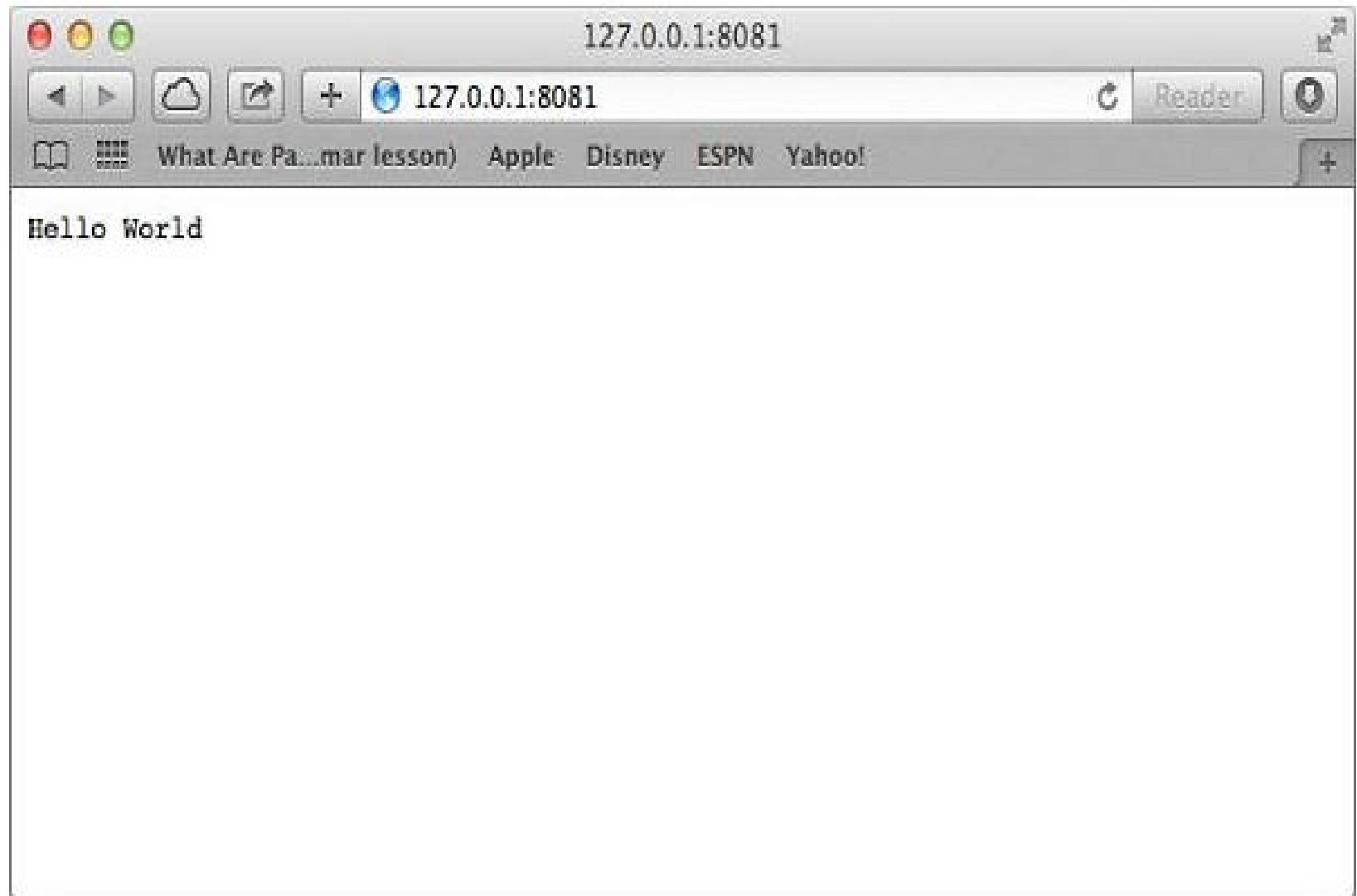
Main.js

run



```
Node.js command prompt - node main.js
Your environment has been set up for using Node.js 4.4.2 (x64) and npm.
C:\Users\javatpoint1>cd desktop
C:\Users\javatpoint1\Desktop>node main.js
Server running at http://127.0.0.1:8081/
-
```

Open `http://127.0.0.1:8081/` in any browser and observe the following result.



A Vast Number of Libraries

- npm with its simple structure helped the ecosystem of Node.js proliferate, and now the npm registry hosts over 1,000,000 open source packages you can freely use.

Differences between Node.js and the Browser

- Both the browser and Node.js use JavaScript as their programming language.
- Building apps that run in the browser is a completely different thing than building a Node.js application.
- From the perspective of a frontend developer who extensively uses JavaScript, Node.js apps bring with them a huge advantage: the comfort of programming everything - the frontend and the backend - in a single language.

- In the browser, most of the time what you are doing is interacting with the DOM, or other Web Platform APIs like Cookies.
- Those do not exist in Node.js.
- You don't have the document, window and all the other objects that are provided by the browser.
- And in the browser, we don't have all the nice APIs that Node.js provides

- Another big difference is that in Node.js you control the environment.
- Unless you are building an open source application that anyone can deploy anywhere, you know which version of Node.js you will run the application on.
- Compared to the browser environment, where you don't get the luxury to choose what browser your visitors will use, this is very

REPL- Read Eval Print Loop

- REPL stands for Read Eval Print Loop and it represents a computer environment like a Windows console or Unix/Linux shell where a command is entered and the system responds with an output in an interactive mode.
- Node.js or Node comes bundled with a REPL environment.
- It performs the following tasks –
- Read – Reads user's input, parses the input into JavaScript data-structure, and stores in memory.
- Eval – Takes and evaluates the data structure.

- The REPL feature of Node is very useful in experimenting with Node.js codes and to debug JavaScript codes.
- REPL can be started by simply running node on shell/console without any arguments as follows.
- `$ node`
- You will see the REPL Command prompt `>` where you can type any Node.js command –

```
$ node
```

```
>
```

Simple Expression

Let's try a simple mathematics at the Node.js
REPL command prompt –

```
$ node
```

```
> 1 + 3
```

```
4
```

```
> 1 + ( 2 * 3 ) - 4
```

```
3
```

```
>
```

Use Variables

You can make use variables to store values and print later like any conventional script.

If var keyword is not used, then the value is stored in the variable and printed. Whereas if var keyword is used, then the value is stored but not printed.

You can print variables using `console.log()`.

```
$ node
```

```
> x = 10
```

```
10
```

```
> var y = 10
```

```
20
```

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

```
Hello World
```

```
undefined
```

Multiline Expression

Node REPL supports multiline expression similar to JavaScript. Let's check the following do-while loop in action –

```
$ node
```

```
> var x = 0
```

```
undefined
```

```
> do {
```

```
... x++;
```

```
... console.log("x: " + x);
```

```
... }
```

Underscore Variable

You can use underscore (_) to get the last result –

```
$ node
```

```
> var x = 10
```

```
undefined
```

```
> var y = 20
```

```
undefined
```

```
> x + y
```

```
30
```

```
> var sum = _
```

```
undefined
```

```
> console.log(su
```

```
30
```

```
undefined
```

```
>
```


REPL Commands

- `ctrl + c` – terminate the current command.
- `ctrl + c` twice – terminate the Node REPL.
- `ctrl + d` – terminate the Node REPL.
- Up/Down Keys – see command history and modify previous commands.
- tab Keys – list of current commands.
- `.help` – list of all commands.
- `.break` – exit from multiline expression.
- `.clear` – exit from multiline expression.
- `.save filename` – save the current Node REPL session to a file.

What is Callback?

- 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

Eg- Blocking code

- *Create a text file named **input.txt** with the following content*
- Web is giving self learning content to teach the world in simple and easy way!!!!
- *Create a js file named **main.js** with the following code –*

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

```
var data = fs.readFileSync('input.txt');
```

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

```
console.log("Program Ended");
```

- Now run the main.js to see the result –

o/p

- Web is giving self learning content to teach the world in simple and easy way!!!!

Program Ended

example shows that the program blocks until it reads the file and then only it proceeds to end the program.

Non-Blocking Code Example

- *Create a text file named input.txt with the following content.*
- Web is giving self learning content to teach the world in simple and easy way!!!!
- *Update main.js to have the following code –*

```
var fs = require("fs");  
fs.readFile('input.txt', function (err, data)  
{ if (err) return console.error(err);  
  console.log(data.toString());  
});  
console.log("Program Ended");
```

o/p

- Program Ended
- Web is giving self learning content to teach the world in simple and easy way!!!!

The second example shows that the program does not wait for file reading and proceeds to print "Program Ended" and at the same time, the program without blocking continues reading the file.

- 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 "Program Ended" 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-

Node.js File System (FS)

- The Node.js file system module allows you to work with the file system on your computer.
- To handle file operations like creating, reading, deleting, etc., Node.js provides an inbuilt module called FS (File System).
- The fs module is responsible for all the asynchronous or synchronous file I/O operations.
- To include the File System module, use the `require()` method:

Node.js FS Reading File

- Every method in fs module has synchronous and asynchronous forms.
- Asynchronous methods take a last parameter as completion function callback.
- Asynchronous method is preferred over synchronous method because it never blocks the program execution where as the synchronous method blocks.

- Create a text file named "input.txt" having the following content.
- *File: input.txt*

IP is a one of the best online tutorial website to learn different technologies in a very easy and efficient manner.

File: main.js

```
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());  
});
```

// Synchronous read

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

Node.js Open a file

Syntax:

to open a file in asynchronous mode:

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

path: This is a string having file name including path.

flags: Flag specifies the behavior of the file to be opened. All possible values have been mentioned below.

mode: This sets the file mode (permission and sticky bits), but only if the file was created. It defaults to 0666, readable and writeable

Node.js Flags for Read/Write

Flag	Description
r	open file for reading. an exception occurs if the file does not exist.
r+	open file for reading and writing. an exception occurs if the file does not exist.
rs	open file for reading in synchronous mode.
rs+	open file for reading and writing, telling the os to open it synchronously.
w	open file for writing. the file is created (if it does not exist) or truncated (if it exists).
wx	like 'w' but fails if path exists.

Flag	Description
w+	open file for reading and writing. the file is created (if it does not exist) or truncated (if it exists).
wx+	like 'w+' but fails if path exists.
a	open file for appending. the file is created if it does not exist.
ax	like 'a' but fails if path exists.
a+	open file for reading and appending. the file is created if it does not exist.
ax+	open file for reading and appending. the file is created if it does not exist.

File: main.js

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

Writing a File

- Syntax
- `fs.writeFile(filename, data[, options], callback)`
- This method will over-write the file if the file already exists.
- **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


```
var fs = require("fs");  
console.log("Going to write into existing file");  
fs.writeFile('input.txt', 'Simply Easy Learning!',  
  function(err) {  
    if (err) {  
      return console.error(err);  
    }  
    console.log("Data written successfully!");  
  }  
}
```

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

Reading a File

- Syntax
- `fs.read(fd, buffer, offset, length, position, callback)`
- This method will use file descriptor to read the file.
- **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

```
var fs = require("fs");
var buf = new Buffer(1024);
console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
  console.log("Going to read the file");
  fs.read(fd, buf, 0, buf.length, 0, function(err,
bytes){
  if (err){
    console.log(err);
```

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

Closing a File

- Syntax
- `fs.close(fd, callback)`
- **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.

```
var fs = require("fs");  
var buf = new Buffer(1024);  
console.log("Going to open an existing file");  
fs.open('input.txt', 'r+', function(err, fd) {  
    if (err) {  
        return console.error(err);  
    }  
    console.log("File opened successfully!");  
    console.log("Going to read the file");
```

```
fs.read(fd, buf, 0, buf.length, 0, function(err,
  bytes) {
    if (err) {
      console.log(err);    }
    // Print only read bytes to avoid junk.
    if(bytes > 0) {
      console.log(buf.slice(0, bytes).toString());
    }

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


Delete a File

- Syntax
- `fs.unlink(path, callback)` Parameters
- **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.

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

console.log("Going to delete an existing file");
fs.unlink('input.txt', function(err) {
  if (err) {
    return console.error(err);
  }
  console.log("File deleted successfully!");
});
```

Create a Directory

Syntax

```
fs.mkdir(path[, mode], callback)
```

path – This is the directory name including path.

mode – This is the directory permission to be set. Defaults to 0777.

callback – This is the callback function. No arguments other than a possible exception are given to the completion callback.

```
var fs = require("fs");
console.log("Going to create directory
/tmp/test");
fs.mkdir('/tmp/test',function(err) {
  if (err) {
    return console.error(err);
  }
  console.log("Directory created
successfully!");
});
```

Node.js Buffers

- Node.js provides Buffer class to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.
- Buffer class is used because pure JavaScript is not suitable to binary data.
- So, when dealing with TCP streams or the file system, it's necessary to handle octet streams.
- Buffer class is a global class. It can be accessed in application without importing buffer module.

There are many ways to construct a Node buffer.

- 3 most widely used methods:

- Create an uninitiated buffer:

Following is the syntax of creating an uninitiated buffer of 10 octets:

- `var buf = new Buffer(10);`

- Create a buffer from array:
- Following is the syntax to create a Buffer from a given array:
- `var buf = new Buffer([10, 20, 30, 40, 50]);`

- Create a buffer from string:
- Following is the syntax to create a Buffer from a given string and optionally encoding type:
- `var buf = new Buffer("Simply Easy Learning", "utf-8");`

Node.js Writing to buffers

- **Syntax:**
- `buf.write(string[, offset][, length][, encoding])`
- **string:** It specifies the string data to be written to buffer.
- **offset:** It specifies the index of the buffer to start writing at. Its default value is 0.
- **length:** It specifies the number of bytes to write. Defaults to `buffer.length`
- **encoding:** Encoding to use. 'utf8' is the default encoding.

Return values from writing buffers:

- This method is used to return number of octets written.
- In the case of space shortage for buffer to fit the entire string, it will write a part of the string.

File: main.js

```
buf = new Buffer(256);  
len = buf.write("Simply Easy Learning TEIT");  
console.log("Octets written : "+ len);
```

Node.js Reading from buffers

- Syntax:

`buf.toString([encoding][, start][, end])`

- encoding: It specifies encoding to use. 'utf8' is the default encoding
- start: It specifies beginning index to start reading, defaults to 0.
- end: It specifies end index to end reading, defaults is complete buffer.

- Return values reading from buffers:
- This method decodes and returns a string from buffer data encoded using the specified character set encoding.

```
buf = new Buffer(26);  
for (var i = 0 ; i < 26 ; i++) {  
    buf[i] = i + 97;  
}  
console.log( buf.toString('ascii'));  
console.log( buf.toString('ascii',0,5));  
console.log( buf.toString('utf8',0,5));  
console.log( buf.toString(undefined,0,5));
```

Convert Buffer to JSON

- Syntax

`buf.toJSON()`

- Return Value
- This method returns a JSON-representation of the Buffer instance.

- Example

```
var buf = new Buffer('Simply Easy Learning  
TEIT');
```

```
var json = buf.toJSON(buf);
```

```
console.log(json);
```

Concatenate Buffers

Syntax

`Buffer.concat(list[, totalLength])`

Where ,

list – Array List of Buffer objects to be concatenated.

totalLength – This is the total length of the buffers when concatenated.

Return Value

This method returns a Buffer instance.

Eg

```
var buffer1 = new Buffer('Hello TEIT ');  
var buffer2 = new Buffer('Enjoy Learning');  
var buffer3 = Buffer.concat([buffer1,buffer2]);  
  
console.log("buffer3 content: " +  
    buffer3.toString());
```

Compare Buffers

Syntax

```
buf.compare(otherBuffer);
```

Where,

otherBuffer – This is the other buffer which will be compared with **buf**

Return Value

Returns a number indicating whether it comes before or after or is the same as the **otherBuffer** in sort order.

```
var buffer1 = new Buffer('ABC');  
var buffer2 = new Buffer('ABCD');  
var result = buffer1.compare(buffer2);  
if(result < 0) {  
    console.log(buffer1 +" comes before " +  
    buffer2);  
} else if(result === 0) {  
    console.log(buffer1 +" is same as " +  
    buffer2);  
} else {  
    console.log(buffer1 +" comes after " +  
    buffer2);  
}
```

Copy Buffer

- Syntax

```
buf.copy(targetBuffer[, targetStart][,  
         sourceStart][, sourceEnd])
```

- targetBuffer – Buffer object where buffer will be copied.
- targetStart – Number, Optional, Default: 0
- sourceStart – Number, Optional, Default: 0
- sourceEnd – Number, Optional, Default: buffer.length
- Return Value
- No return value. Copies data from a region of this buffer to a region in the target buffer

- If undefined, the targetStart and sourceStart parameters default to 0, while sourceEnd defaults to buffer.length.

- Example

```
var buffer1 = new Buffer('ABC');
```

```
//copy a buffer
```

```
var buffer2 = new Buffer(3);
```

```
buffer1.copy(buffer2);
```

```
console.log("buffer2 content: " +  
    buffer2.toString());
```

Slice Buffer

- Syntax

`buf.slice([start][, end])`

- start – Number, Optional, Default: 0
- end – Number, Optional, Default: `buffer.length`
- Return Value
- Returns a new buffer which references the same memory as the old one, but offset and cropped by the start (defaults to 0) and end (defaults to `buffer.length`) indexes.
- Negative indexes start from the end of the

```
var    buffer1    =    new    Buffer('Internet  
    Programming');  
//slicing a buffer  
var buffer2 = buffer1.slice(0,9);  
console.log("buffer2    content:    "    +  
    buffer2.toString());
```

Buffer Length

Syntax

```
buf.length;
```

Return Value

Returns the size of a buffer in bytes.

```
var buffer = new Buffer('Internet  
Programming');  
//length of the buffer  
console.log("buffer length: " +  
    buffer.length);
```


Node.js Streams

- Streams are the objects that facilitate you to read data from a source and write data to a destination.
- There are four types of streams in Node.js:
- **Readable:** This stream is used for read operations.
- **Writable:** This stream is used for write operations.
- **Duplex:** This stream can be used for both read and write operations.
- **Transform:** It is type of duplex stream

- Each type of stream is an Event emitter instance and throws several events at different times.
- Following are some commonly used events:
- **Data:** This event is fired when there is data available to read.
- **End:** This event is fired when there is no more data available to read.
- **Error:** This event is fired when there is any error receiving or writing data.
- **Finish:** This event is fired when all data has been flushed to underlying system.

Node.js Reading from stream

- Create a text file named `input.txt` having the following content:
- Chaining stream is a mechanism of creating a chain of multiple stream operations by connecting output of one stream to another stream. It is generally used with piping operation.

File: main.js

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

```
var data = "";
```

```
// Create a readable stream
```

```
var readerStream =
```

```
    fs.createReadStream('input.txt');
```

```
// Set the encoding to be utf8.
```

```
readerStream.setEncoding('UTF8');
```

```
// Handle stream events -->
```

```
data, end, and error
```

```
readerStream.on('data', function(chunk) {
```

```
    data += chunk;
```

```
});
```

```
readerStream.on('end',function(){  
    console.log(data);  
});  
readerStream.on('error', function(err){  
    console.log(err.stack);  
});  
console.log("Program Ended");
```

File:
main.js

Node.js Writing to stream

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

```
var data = 'A Solution of all Technology written  
in text file';
```

```
// Create a writable stream
```

```
var writerStream =  
fs.createWriteStream('output.txt');
```

```
// Write the data to stream with encoding  
to be utf8
```

```
writerStream.write(data,'UTF8');
```

```
// Mark the end of file
```

```
writerStream.end();
```

// Handle stream events --> finish, and error

```
writerStream.on('finish', function() {  
    console.log("Write completed.");  
});  
writerStream.on('error', function(err){  
    console.log(err.stack);  
});  
console.log("Program Ended");
```

Now, you can see that a text file named "output.txt" is created where you had saved "input.txt" and "main.js" file. In my case, it is on desktop.

Node.js Piping Streams

- Piping is a mechanism where output of one stream is used as input to another stream. There is no limit on piping operation.
- File: main.js

```
var fs = require("fs");  
                                // Create a readable stream  
var                                readerStream                                =  
    fs.createReadStream('input.txt');  
                                // Create a writable stream  
var                                writerStream                                =  
    fs.createWriteStream('output.txt');  
                                // Pipe the read and write operations  
                                // read input.txt and write data to output.txt  
readerStream.pipe(writerStream);
```


- Now, you can see that a text file named "output.txt" is created where you had saved main.js file.

Chaining the Streams

- Chaining is a mechanism to connect the output of one stream to another stream and create a chain of multiple stream operations.
- It is normally used with piping operations.
- Now use piping and chaining to first compress a file and then decompress the same.

Main.js

```
var fs = require("fs");  
var zlib = require('zlib');  
    // Compress the file input.txt to input.txt.gz  
fs.createReadStream('input.txt')  
    .pipe(zlib.createGzip())  
  
    .pipe(fs.createWriteStream('input.txt.gz'));  
console.log("File Compressed.");
```

You will find that input.txt has been compressed and it created a file input.txt.gz in the current directory.

decompress the same file

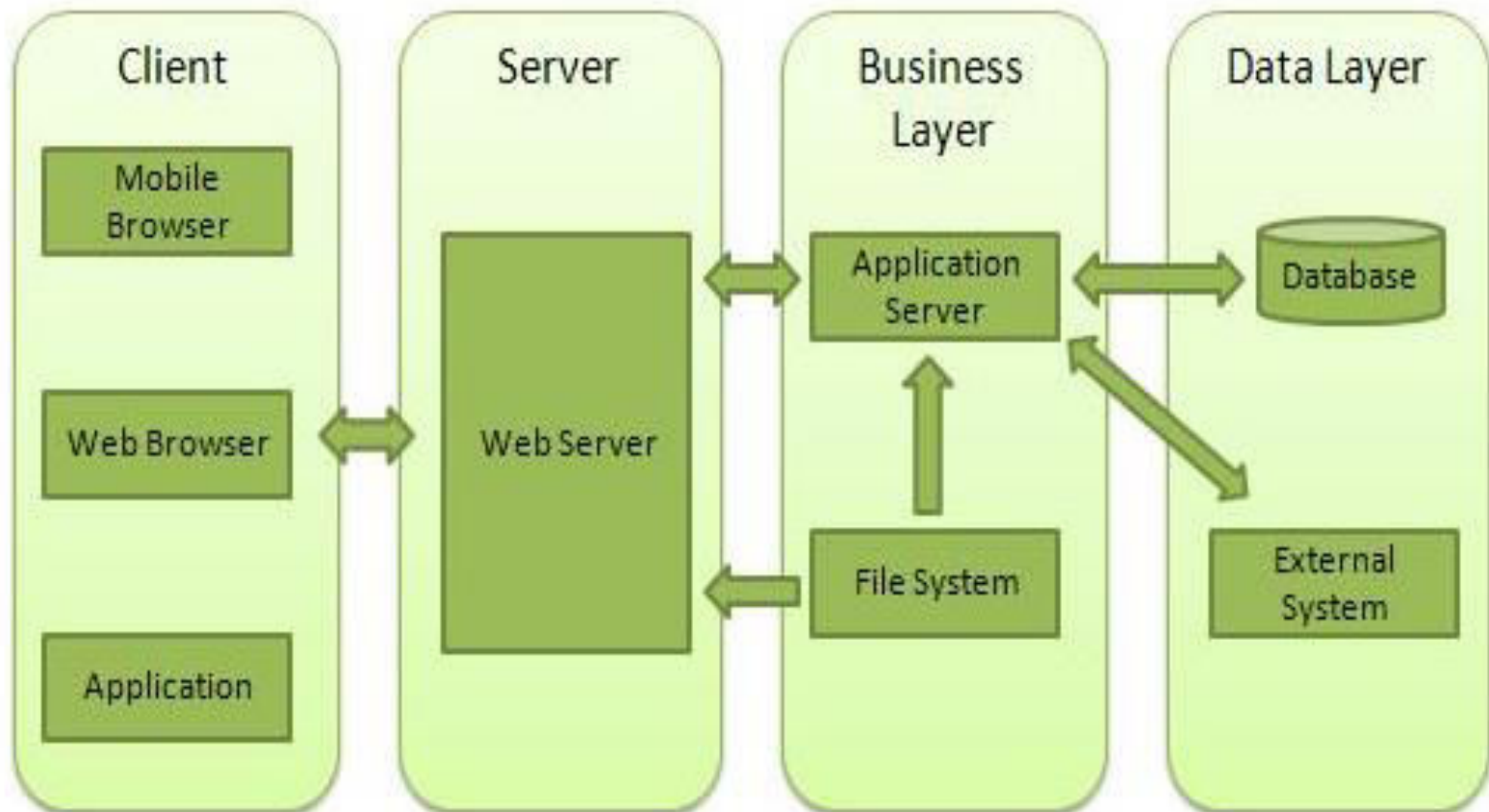
```
var fs = require("fs");  
var zlib = require('zlib');  
// Decompress the file input.txt.gz to input.txt  
fs.createReadStream('input.txt.gz')  
  .pipe(zlib.createGunzip())  
  .pipe(fs.createWriteStream('input.txt'));  
console.log("File Decompressed.");
```

Node.js - Web Module

- **What is a Web Server?**
- 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

Web Application Architecture

- 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.

Creating a Web Server using Node

- Node.js provides an **http** module which can be used to create an HTTP client of a server.
- Following is the bare minimum structure of the HTTP server which listens at 8081 port.

Create a js file named server.js


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

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

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

```
// Create a server
```

```
http.createServer( function (request, response) {
```

```
    // Parse the request containing file name
```

```
    var pathname =
```

```
    url.parse(request.url).pathname;
```

```
    // Print the name of the file for which request is made.
```

```
    console.log("Request for " + pathname + "  
received.");
```

// Read the requested file content from file system

```
fs.readFile(pathname.substr(1), function (err,  
data) {
```

```
  if (err) {
```

```
    console.log(err);
```

```
    // HTTP Status: 404 : NOT FOUND
```

```
    // Content Type: text/plain
```

```
    response.writeHead(404, {'Content-Type':  
'text/html'});
```

```
  } else {
```

```
//Page found
```

```
// HTTP Status: 200 : OK
```

```
// Content Type: text/plain
```

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

```
// Write the content of the file to response body
```

```
response.write(data.toString());
```

```
}
```

// Send the response body

```
response.end();
```

```
});
```

```
}).listen(8081);
```

// Console will print the message

```
console.log('Server running at  
http://127.0.0.1:8081/');
```

Next create the following html file named **index.html** in the same directory where you created server.js.

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

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

Now run server.js

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

Creating Web client using Node

- A web client can be created using **http** module.
- Create a js file named client.js –

Client.js

```
var http = require('http');  
// Options to be used by request  
var options = {  
  host: 'localhost',  
  port: '8081',  
  path: '/index.html'  
};
```

// Callback function is used to deal with response

```
var callback = function(response) {
```

// Continuously update stream with data

```
var body = "";
```

```
response.on('data', function(data) {
```

```
    body += data;
```

```
});
```

```
response.on('end', function() {
```

// Data received completely. *// Make a request to the server*

```
    console.log(body);
```

```
});
```

```
var req =
```

```
http.request(options,  
callback);
```


- Now run the client.js from a different command terminal other than server.js to see the result

Verify the Output.

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

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

Verify the Output at server end.

Server running at <http://127.0.0.1:8081/>
Request for /index.htm received.

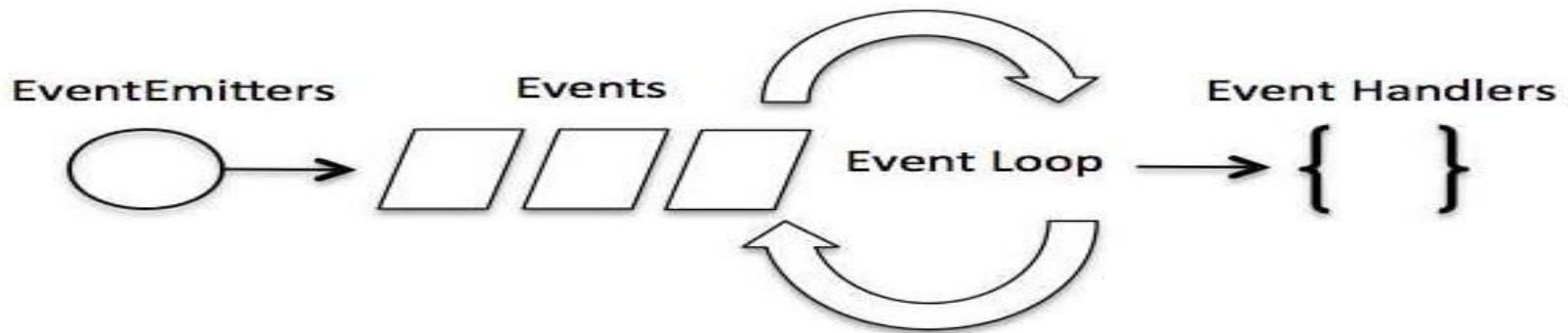
Node.js - Event Loop

- 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

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

Event loop



- 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.

```
// 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);
```

```
// Fire an event
```

```
    EventEmitter.emit('eventName');
```

Example

main.js

```
// Import events module
var events = require('events');
// Create an EventEmitter object
var EventEmitter = new events.EventEmitter();
// Create an event handler as follows
var connectHandler = function connected() {
  console.log('connection succesful.');
  // Fire the data_received event
  EventEmitter.emit('data_received');
}
```

```
// Bind the connection event with the handler  
eventEmitter.on('connection',  
    connectHandler);
```

```
// Bind the data_received event with the  
    anonymous function
```

```
eventEmitter.on('data_received', function()  
    {  
        console.log('data received succesfully.');  
});
```

```
// Fire the connection event
```

```
eventEmitter.emit('connection');
```

```
console.log("Program Ended.");
```


run the above program and check its output

–

```
$ node main.js
```

o/p

connection successful.

data received successfully.

Program Ended.

How Node Applications Work?

In Node Application, any async function accepts a callback as the last parameter and a callback function accepts an error as the first parameter.

Create a text file named **input.txt** with the following content.

Web is giving self learning content to teach the world in simple and easy way!!!!

Create a js file named **main.js** having the following code –

```
var fs = require("fs");
fs.readFile('input.txt', function (err, data) {
  if (err) {
    console.log(err.stack);
    return;
  }
  console.log(data.toString());
});
console.log("Program Ended");
```

Here `fs.readFile()` is a async function whose purpose is to read a file.

If an error occurs during the read operation, then the `err` object will contain the corresponding error, else `data` will contain the contents of the file.

`readFile` passes `err` and `data` to the callback function after the read operation is complete, which finally prints the content.

Program Ended

Web is giving self learning content
to teach the world in simple and easy

Node.js EventEmitter

- 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.

Events

- **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.

- **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.

```
var events = require('events');
var EventEmitter = new events.EventEmitter();
// listener #1
var listner1 = function listner1() {
    console.log('listner1 executed.');
```



```
}
// listener #2
var listner2 = function listner2() {
    console.log('listner2 executed.');
```



```
}
// Bind the connection event with the listner1 function
eventEmitter.addListener('connection', listner1);
```



```
// Bind the connection event with the listner2 function
eventEmitter.on('connection', listner2);
var eventListeners =
    require('events').EventEmitter.listenerCount
    (eventEmitter,'connection');
console.log(eventListeners + " Listner(s) listening to
    connection event");
// Fire the connection event
eventEmitter.emit('connection');
// Remove the binding of listner1 function
eventEmitter.removeListener('connection', listner1);
console.log("Listner1 will not listen now.");
// Fire the connection event
```

```
eventListeners =  
    require('events').EventEmitter.listenerCount(eventEmitter, 'connection');  
console.log(eventListeners + " Listener(s)  
    listening to connection event");  
  
console.log("Program Ended.");
```