# LEARNING UNIT THREE: NODEJS

Contents

## CHAPTER ONE: INTRODUCTION

### 1.1 Introduction

#### 1.1.1 What is nodejs?

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. Visit Wikipedia to know the history of Node.js.

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

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

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

Initially implemented as a client-side scripting language. Nowadays, it is used to execute JavaScript code and scripts that run server-side to create dynamic web pages. The latest version of Node.js is 10.10.0.

Node.js is based on an event-driven architecture and a non-blocking Input/Output API that is designed to optimize an application's throughput and scalability for real-time web applications.

Over a long period of time, the framework available for web development were all based on a stateless model. A stateless model is where the data generated in one session (such as information about user settings and events that occurred) is not maintained for usage in the next session with that user.
A lot of work had to be done to maintain the session information between requests for a user. But with Node.js there is finally a way for web applications to have a real-time, two-way connections, where both the client and server can initiate communication, allowing them to exchange data freely.

#### 1.1.2 Why use Node.js?

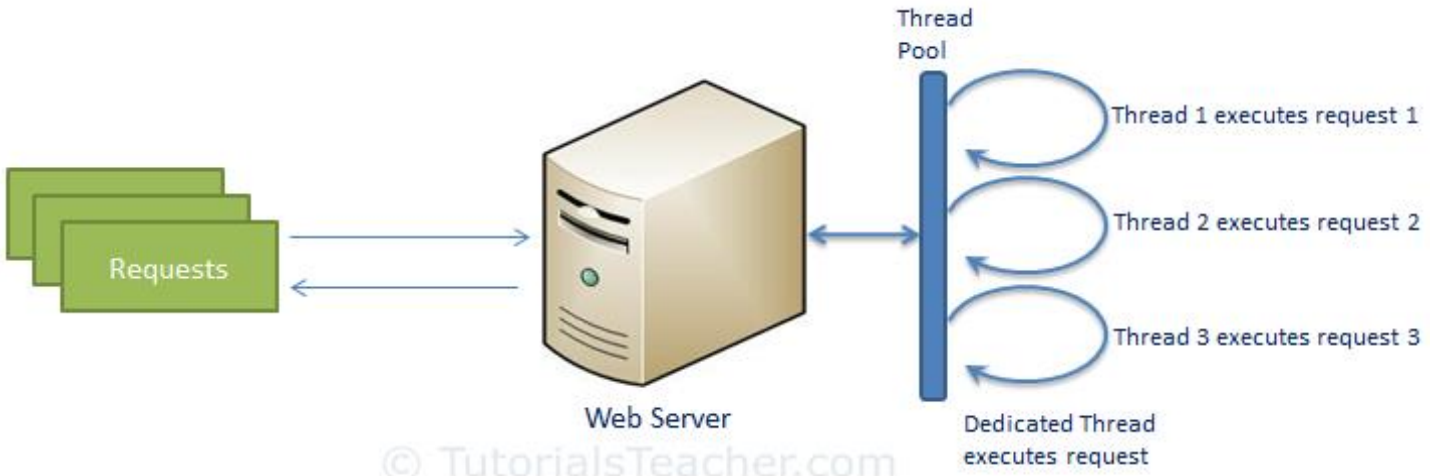
1. Node.js is an open-source framework under MIT license. (MIT license is a free software license originating at the Massachusetts Institute of Technology (MIT).)
2. Uses JavaScript to build entire server side application.
3. Lightweight framework that includes bare minimum modules. Other modules can 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

## 1.2 Processing model

In this section, we will learn about the Node.js process model and understand why we should use Node.js.

### 1.2.1 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.

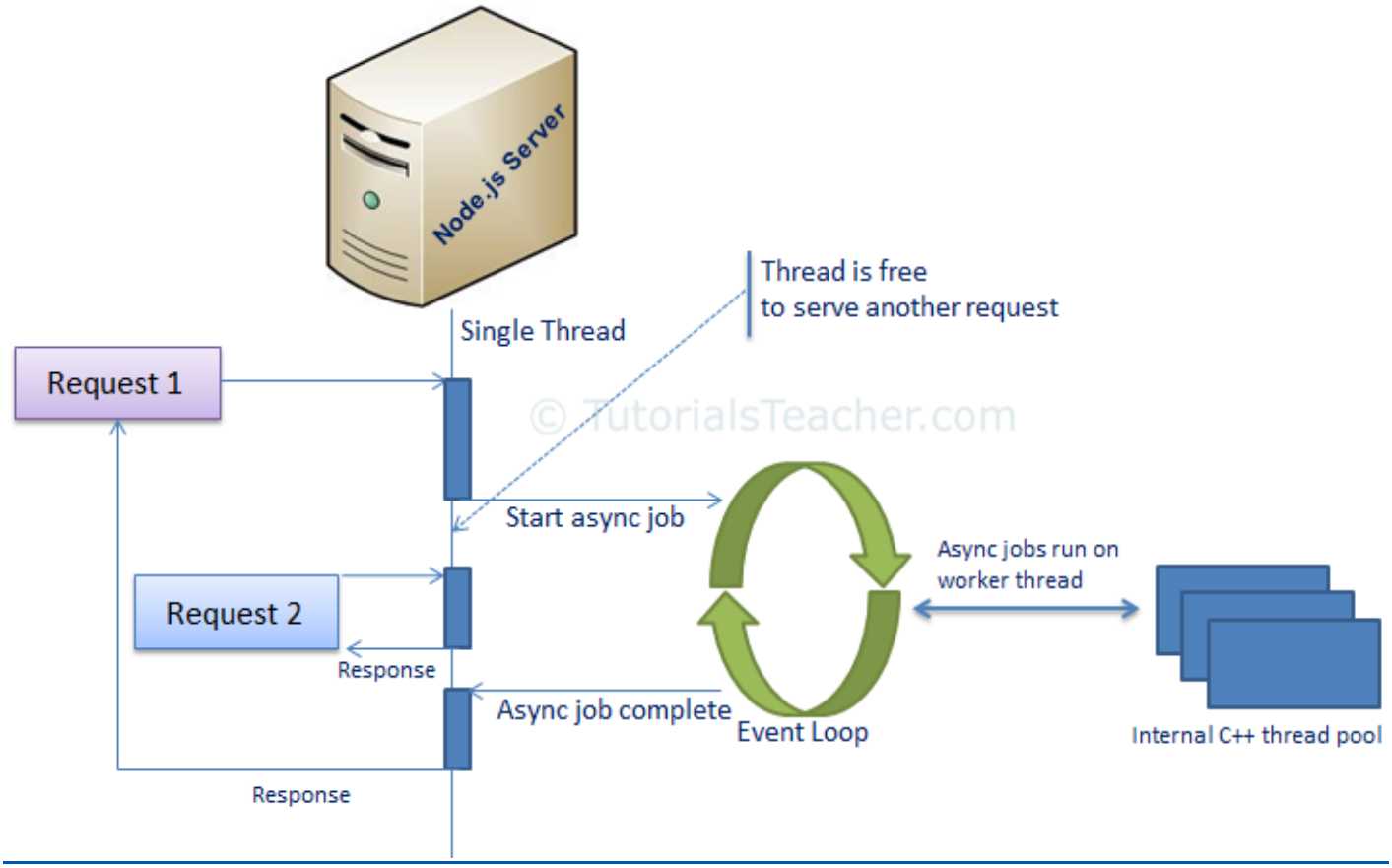



### 1.2.2 Node.js Process Model

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 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.

### 1.3 Summary of Node.js features

1. Asynchronous event driven IO helps concurrent request handling – This is probably the biggest selling points of Node.js. This feature basically means that if a request is received by Node for some Input/Output operation, it will execute the operation in the background and continue with processing other requests.
This is quite different from other programming languages. A simple example of this is given in the code below

```
var fs = require('fs');
        fs.readFile("Sample.txt",function(error,data)
        {
            console.log("Reading Data completed");
    });
```

- The above code snippet looks at reading a file called Sample.txt. In other programming languages, the next line of processing would only happen once the entire file is read.
- But in the case of Node.js the important fraction of code to notice is the declaration of the function ('function(error,data)'). This is known as a callback function.
- So what happens here is that the file reading operation will start in the background. And other processing can happen simultaneously while the file is being read. Once the file read operation is

completed, this anonymous function will be called and the text "Reading Data completed" will be written to the console log.

2. Node uses the V8 JavaScript Runtime engine, the one which is used by Google Chrome. Node has a wrapper over the JavaScript engine which makes the runtime engine much faster and hence processing of requests within Node also become faster.
3. Handling of concurrent requests – Another key functionality of Node is the ability to handle concurrent connections with a very minimal overhead on a single process.
4. The Node.js library used JavaScript – This is another important aspect of development in Node.js. A major part of the development community are already well versed in javascript, and hence, development in Node.js becomes easier for a developer who knows javascript.
5. There are an Active and vibrant community for the Node.js framework. Because of the active community, there are always keys updates made available to the framework. This helps to keep the framework always up-to-date with the latest trends in web development.

### 1.4 Who uses Node.js

Node.js is used by a variety of large companies. Below is a list of a few of them.

- Paypal – A lot of sites within Paypal have also started the transition onto Node.js.
- LinkedIn - LinkedIn is using Node.js to power their Mobile Servers, which powers the iPhone, Android, and Mobile Web products.
- Mozilla has implemented Node.js to support browser APIs which has half a billion installs.
- Ebay hosts their HTTP API service in Node.js

### 1.5 When to Use Node.js

Node.js is best for usage in streaming or event-based real-time applications like

1. Chat applications
2. Game servers – Fast and high-performance servers that need to processes thousands of requests at a time, then this is an ideal framework.
3. Good for collaborative environment – This is good for environments which manage document. In document management environment you will have multiple people who post their documents and do constant changes by checking out and checking in documents. So Node.js is good for these environments because the event loop in Node.js can be triggered whenever documents are changed in a document managed environment.
4. Advertisement servers – Again here you could have thousands of request to pull advertisements from the central server and Node.js can be an ideal framework to handle this.
5. Streaming servers – Another ideal scenario to use Node is for multimedia streaming servers wherein clients have request's to pull different multimedia contents from this server.

Node.js is good when you need high levels of concurrency but less amount of dedicated CPU time.

Best of all, since Node.js is built on javascript, it's best suited when you build client-side applications which are based on the same javascript framework.

### 1.6 When to not use Node.js

Node.js can be used for a lot of applications with various purpose, the only scenario where it should not be used is if there are long processing times which is required by the application.
Node is structured to be single threaded. If any application is required to carry out some long running calculations in the background. So if the server is doing some calculation, it won't be able to process any other requests. As discussed above, Node.js is best when processing needs less dedicated CPU time.

### 1.7 Setup Node.js Development Environment

In this section, you will learn about the tools required and steps to setup development environment to develop a Node.js application.

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.

1. Node.js
2. Node Package Manager (NPM)
3. IDE (Integrated Development Environment) or TextEditor

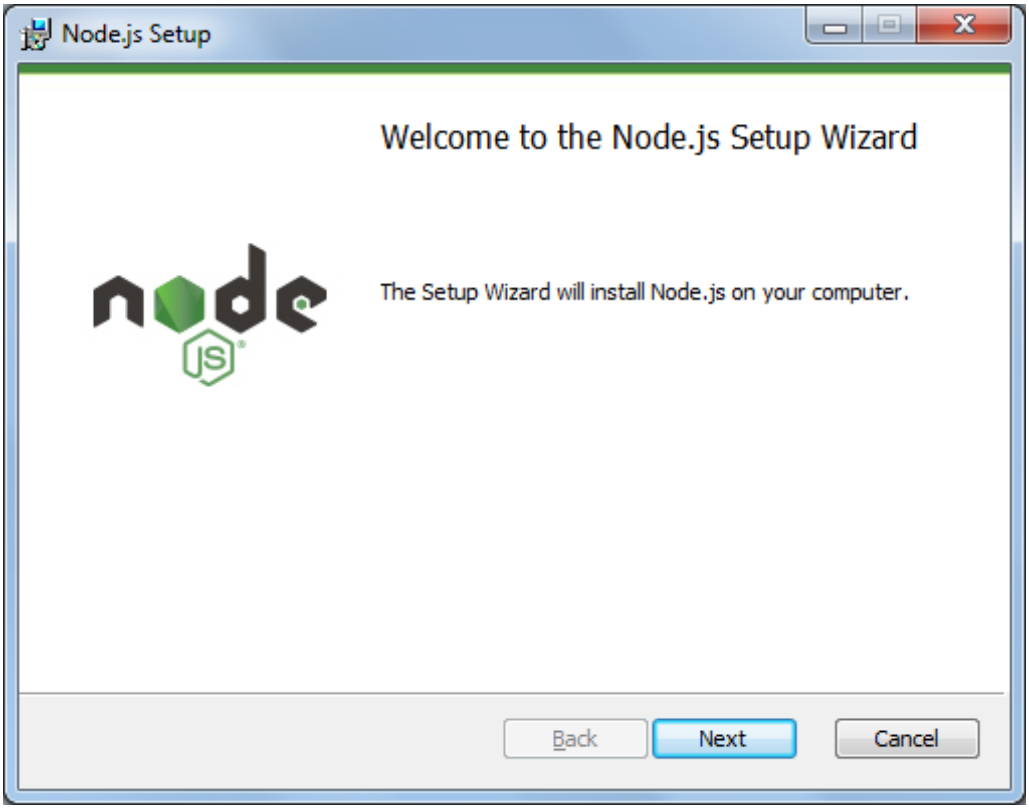
NPM (Node Package Manager) is included in Node.js installation since Node version 0.6.0., so there is no need to install it separately.

#### 1.7.1 Install Node.js on Windows

Visit Node.js official web site https://nodejs.org. It will automatically detect OS and display download link as per your Operating System. For example, it will display following download link for 64 bit Windows OS.


Download node MSI for windows by clicking on 8.11.3 LTS or 10.5.0 Current button. We have used the latest version 10.5.0 for windows through out these tutorials.

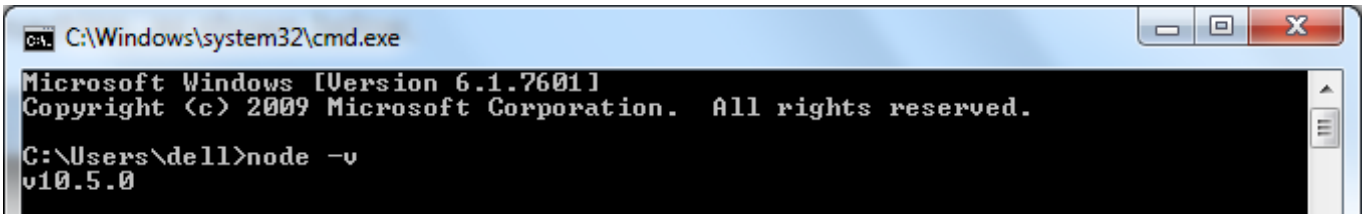
After you download the MSI, double-click on it to start the installation as shown below.



Click Next to read and accept the License Agreement and then click Install. It will install Node.js quickly on your computer. Finally, click finish to complete the installation.

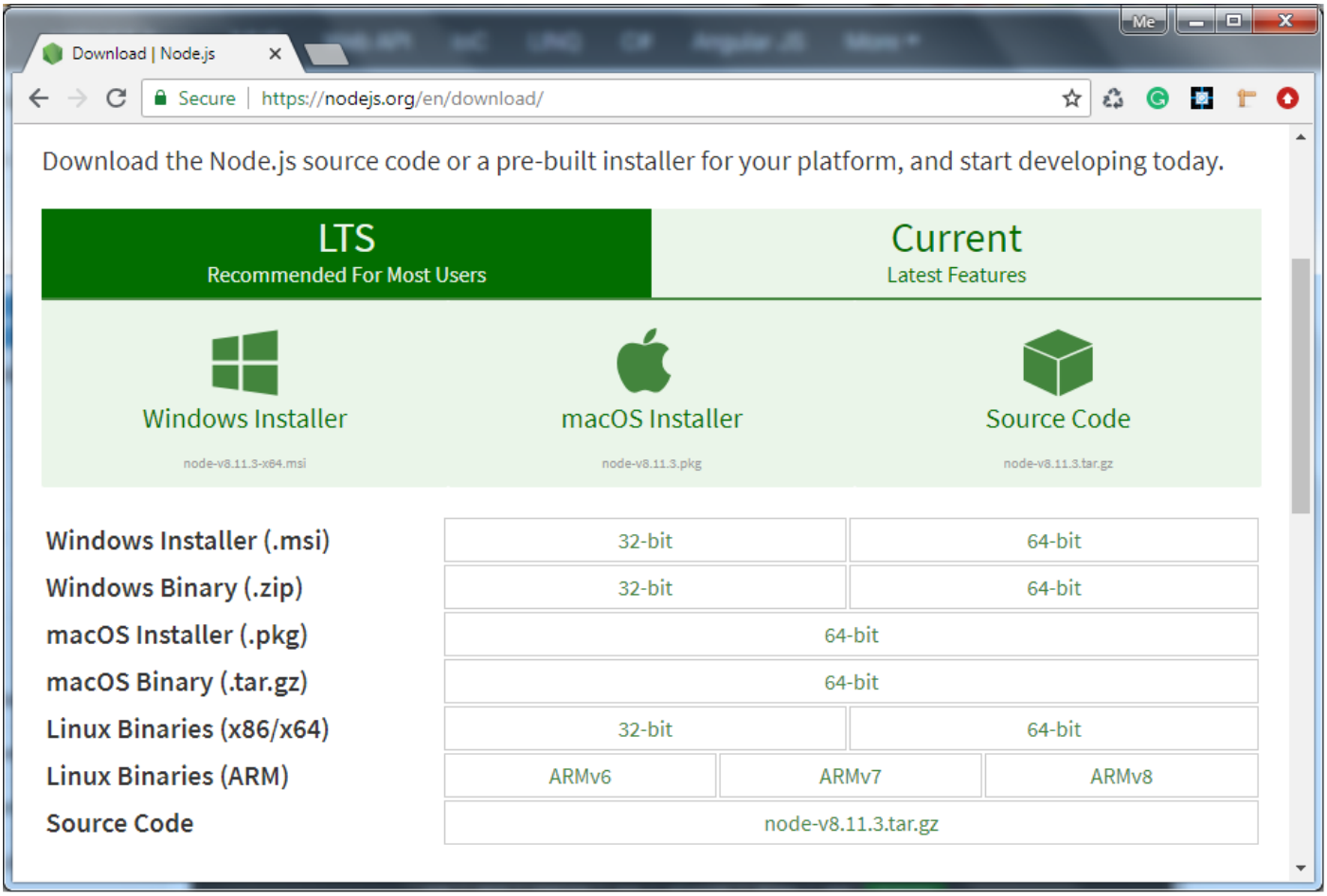
### 1.7.2 Verify Installation

Once you install Node.js on your computer, you can verify it by opening the command prompt and typing `node -v`. If Node.js is installed successfully then it will display the version of the Node.js installed on your machine, as shown below.

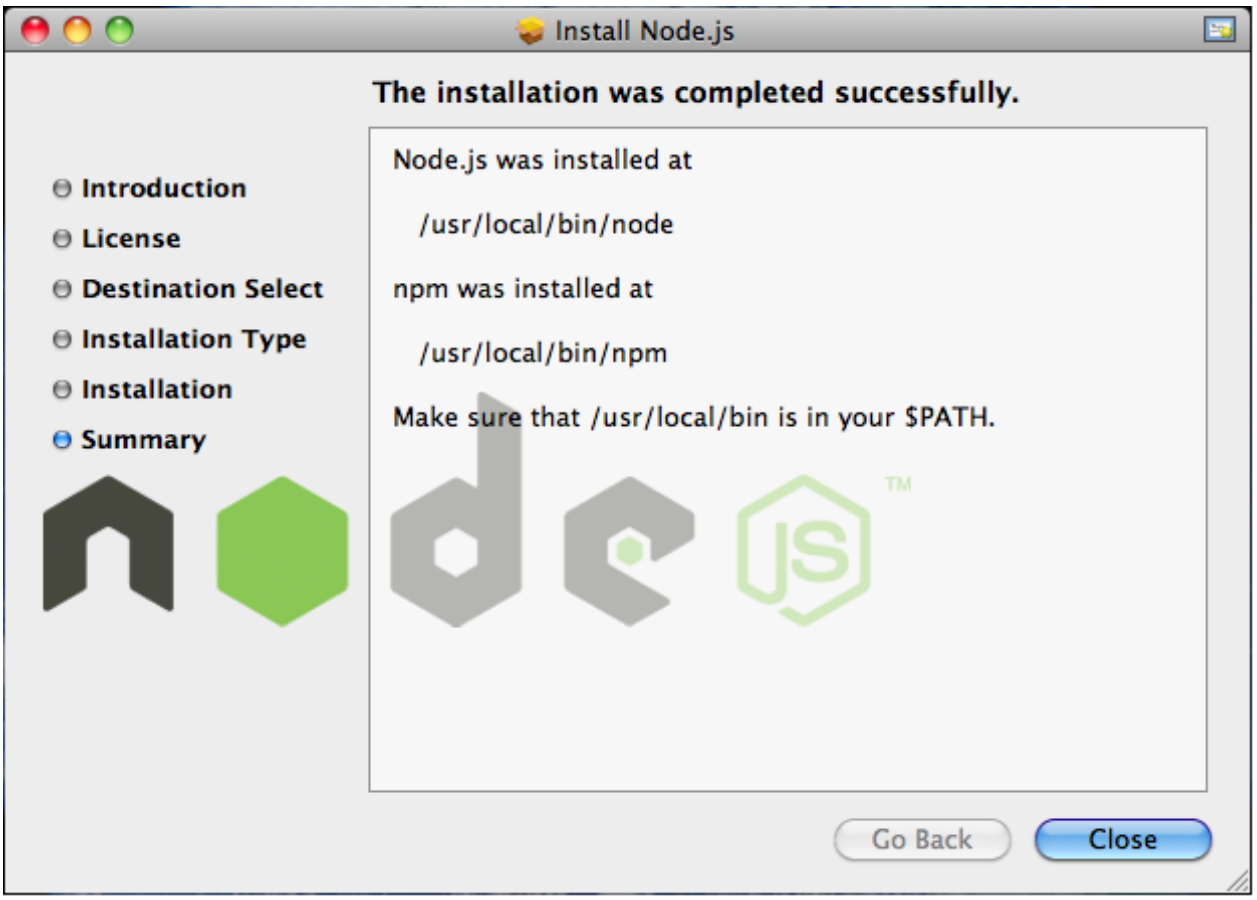



### 1.7.3 Install Node.js on Mac/Linux

Visit Node.js official web site https://nodejs.org/en/download page. Click on the appropriate installer for Mac (.pkg or .tar.gz) or Linux to download the Node.js installer.




Once downloaded, click on the installer to start the Node.js installation wizard. Click on **Continue** and follow the steps. After successful installation, it will display summary of installation about the location where it installed Node.js and NPM.




After installation, verify the Node.js installation using terminal window and enter the following command. It will display the version number of Node.js installed on your Mac.

```
$ node -v
```

Optionally, for Mac or Linux users, you can directly install Node.js from the command line using Homebrew package manager for Mac OS or Linuxbrew package manager for Linux Operating System. For Linux, you will need to install additional dependencies, viz. Ruby version 1.8.6 or higher and GCC version 4.2 or higher before installing node.

```
$ brew install node
```

### 1.7.4 IDE

Node.js application uses JavaScript to develop an application. So, you can use any IDE or texteditor tool that supports JavaScript syntax. However, an IDE that supports auto complete features for Node.js API is recommended e.g. Visual Studio, Sublime text, Eclipse, Aptana etc.

Node.js Console - REPL

Node.js comes with virtual environment called REPL (aka Node shell). REPL stands for Read-Eval-Print-Loop. It is a quick and easy way to test simple Node.js/JavaScript code.

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


You can now test pretty much any Node.js/JavaScript expression in REPL. For example, if your write "10 + 20" then it will display result 30 immediately in new line.

```
> 10 + 20
30
```

The + operator also concatenates strings as in browser's JavaScript.

```
> "Hello" + "World"
Hello World
```

You can also define variables and perform some operation on them.

```
> var x = 10, y = 20;
> x + y
30
```

If you need to write multi line JavaScript expression or function then just press **Enter** whenever you want to write something in the next line as a continuation of your code. The REPL terminal will display three dots (...), it means you can continue on next line. Write .break to get out of continuity mode.

For example, you can define a function and execute it as shown below.


You can execute an external JavaScript file by writing `node fileName` command. For example, assume that node-example.js is on C drive of your PC with following code.

node-example.js

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

Now, you can execute node-exampel.js from command prompt as shown below.


To exit from the REPL terminal, press Ctrl + C twice or write .exit and press Enter.


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

Note: ECMAScript implementation in Node.js and browsers is slightly different. For example, {}+{} is '[object Object][object Object]' in Node.js REPL, whereas the same code is NaN in the Chrome console because of the automatic semicolon insertion feature. However, mostly Node.js REPL and the Chrome/Firefox consoles are similar.

The following table lists important REPL commands.

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

| REPL Command | Description |
| --- | --- |
| .clear | Exit from multiline expression. |

## CHAPTER TWO: BASICS OF NODE.JS

### 2.0 Node.js Basics

Node.js supports JavaScript. So, JavaScript syntax on Node.js is similar to the browser's JavaScript syntax.

Visit JavaScript section to learn about JavaScript syntax in detail.

### 2.1 Primitive Types

Node.js includes following primitive types:

- String
- Number
- Boolean
- Undefined
- Null
- RegExp

Everything else is an object in Node.js.

### 2.2 Global Objects

Node.js global objects are global in nature and available in all modules. You don't need to include these objects in your application; rather they can be used directly. These objects are modules, functions, strings and object etc. Some of these objects aren't actually in the global scope but in the module scope.
A list of Node.js global objects are given below:

- __dirname
- __filename
- Console
- Process
- Buffer
- setImmediate(callback[, arg][, ...])
- setInterval(callback, delay[, arg][, ...])
- setTimeout(callback, delay[, arg][, ...])
- clearImmediate(immediateObject)
- clearInterval(intervalObject)
- clearTimeout(timeoutObject)

All above objects are accessed via window or global objects. Window object is common in javascript but nodejs uses global object instead.

Different from javascript, nodejs global object can not access custom objects, as a result none of the below functions will print the message on the screen. SayHelloWindow() will not run because window object is not known in nodejs and sayHelloGlobal will fail as well because global.message is not defined.

```
var message ="good morning"

function sayHelloWindow(){
console.log(window.message)
}
sayHello()

function sayHelloGlobal(){
    console.log(global.message)
}
sayHelloGlobal()
```

### 2.3 Node.js Module

#### 2.3.1 Introduction

We have learnt that in javascript browser when a variable is declared, it is added to window object and becomes global. This is dangerous as objects can override each other. Node.js provides concept of module as a solution to that draw back of javascript. Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.

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


Fig: How each module avoid global pollution

Note: In Node.js each file is a module, variables and functions defined in that file have a scope of private to that module only and if you want those objects inside a specific module to be accessible outside, you have to export the module. We will see how to export module and its objects shortly. Module may seem to be a member of

global object but it's not. When you console.log(module) you see that it's not a global member though it has a global influence.

### 2.3.2 Node.js Module Types

Node.js includes three types of modules:

1. Core Modules
2. Local Modules
3. Third Party Modules

### 2.3.3 Node.js Core Modules

Node.js is a light weight framework. The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts. However, you need to import the core module first in order to use it in your application.

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

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

**Loading core modules**

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

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

As per above syntax, specify the module name in the require() function. The require() function will return an object, function, property or any other JavaScript type, depending on what the specified module returns.

The following example demonstrates how to use Node.js http module to create a web server.

Example: Load and Use Core http Module

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

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

In this way, you can load and use Node.js core modules in your application. We will be using core modules in sections to follow.

### 2.3.4 Node.js Local Module

Local modules are modules created locally in your Node.js application. These modules include different functionalities of your application in separate files and folders. You can also package it and distribute it via NPM, so that Node.js community can use it. For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

#### *Writing Simple Module*

Let's write simple logging module which logs the information, warning or error to the console.

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

Log.js

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

Now, let's see how to use the above logging module in our application.

#### *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.

app.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()

Run the above example using command prompt (in Windows) as shown below.

```
C:\> node app.js
Info: Node.js started
```

Thus, you can create a local module using module.exports and use it in your application.

### Export Module in Node.js

In the previous section, you learned how to write a local module using module.exports. In this section, you will learn how to expose different types as a module using module.exports.

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

Let's see how to expose different types as a module using module.exports.

### Export Literals

As mentioned above, *exports* is an object. So it exposes whatever you assigned to it as a module. For example, if you assign a string literal then it will expose that string literal as a module.

The following example exposes simple string message as a module in Message.js.

Message.js

```
module.exports = 'Hello world';
//or
exports = 'Hello world';
```

Now, import this message module and use it as shown below.

app.js

```
var msg = require('./Messages.js');
console.log(msg);
```

Run the above example and see the result as shown below.

```
C:\> node app.js
Hello World
```

Note: You must specify './' as a path of root folder to import a local module. However, you do not need to specify path to import Node.js core module or NPM module in the require() function.

Export Object

*exports* is an object. So, you can attach properties or methods to it. The following example exposes an object with a string property in Message.js file.

Message.js

```
exports.SimpleMessage = 'Hello world';
//or
module.exports.SimpleMessage = 'Hello world';
```

In the above example, we have attached a property "SimpleMessage" to the exports object. Now, import and use this module as shown below.

app.js

```
var msg = require('./Messages.js');
console.log(msg.SimpleMessage);
```

In the above example, require() function will return an object { SimpleMessage : 'Hello World'} and assign it to the msg variable. So, now you can use msg.SimpleMessage.

Run the above example by writing node app.js in the command prompt and see the output as shown below.

```
C:\> node app.js
Hello World
```

The same way as above, you can expose an object with function. The following example exposes an object with log function as a module.

Log.js

```
module.exports.log = function (msg) {
    console.log(msg);
};
```

The above module will expose an object- { log : function(msg){ console.log(msg); } } . Use the above module as shown below.

app.js

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

msg.log('Hello World');
```

Run and see the output in command prompt as shown below.

```
C:\> node app.js
Hello World
```

You can also attach an object to module.exports as shown below.

data.js

```
module.exports = {
    firstName: 'James',
    lastName: 'Bond'
```

```
}
```

app.js

```
var person = require('./data.js');
console.log(person.firstName + ' ' + person.lastName);
```

Run the above example and see the result as shown below.

```
C:\> node app.js
James Bond
```

### Export Function

You can attach an anonymous function to exports object as shown below.

Log.js

```
module.exports = function (msg) {
    console.log(msg);
};
```

Now, you can use the above module as below.

app.js

```
var msg = require('./Log.js');
msg('Hello World');
```

The msg variable becomes function expression in the above example. So, you can invoke the function using parenthesis (). Run the above example and see the output as shown below.

```
C:\> node app.js
Hello World
```

### Export function as a class

In the JavaScript, a function can be treated like a class. The following example exposes a function which can be used like a class.

Person.js

```
module.exports = function (firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.fullName = function () {
        return this.firstName + ' ' + this.lastName;
    }
}
```

The above module can be used as shown below.

app.js

```
var Person = require('./Person.js');
var person1 = new Person('James', 'Bond');
```

```
console.log(person1.fullName());
```

As you can see, we have created a person object using new keyword. Run the above example as below.

```
C:\> node app.js
James Bond
```

In this way, you can export and import a local module created in a separate file under root folder.

Node.js also allows you to create modules in sub folders. Let's see how to load module from sub folders.

### Load Module from Separate Folder

Use the full path of a module file where you have exported it using module.exports. For example, if log module in the log.js is stored under "utility" folder under the root folder of your application then import it as shown below.

app.js

```
var log = require('./utility/log.js');
```

In the above example, **.** is for root folder and then specify exact path of your module file. Node.js also allows us to specify the path to the folder without specifying file name. For example, you can specify only utility folder without specifing log.js as shown below.

app.js

```
var log = require('./utility');
```

In the above example, Node will search for a package definition file called package.json inside utility folder. This is because Node assumes that this folder is a package and will try to look for a package definition. The package.json file should be in a module directory. The package.json under utility folder specifies the file name using "main" key as below.

./utility/package.json

```
{
   "name" : "log",
   "main" : "./log.js"
}
```

Now, Node.js will find log.js file using *main* entry in package.json and import it.

Note:

If package.json file does not exist then it will look for index.js file as a module file by default.

## 2.4 Some of important modules

### 2.4.1 Path module

The path module of Node.js provides useful functions to interact with file paths. The path 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 core, it can be used by requiring it:

*const path = require('path')*

Methods

| Sr.No. | Method & Description |
|---|---|
| 1 | **path.normalize(p)**<br>Normalize a string path, taking care of '..' and '.' parts. |
| 2 | **path.join([path1][, path2][, ...])**<br>Join all the arguments together and normalize the resulting path. |
| 3 | **path.resolve([from ...], to)**<br>Resolves to an absolute path. |
| 4 | **path.isAbsolute(path)**<br>Determines whether the path is an absolute path. An absolute path will always resolve to the same location, regardless of the working directory. |
| 5 | **path.relative(from, to)**<br>Solve the relative path from from to to. |
| 6 | **path.dirname(p)**<br>Return the directory name of a path. Similar to the Unix dirname command. |
| 7 | **path.basename(p[, ext])**<br>Return the last portion of a path. Similar to the Unix basename command. |
| 8 | **path.extname(p)** |

| | |
|---|---|
| | Return the extension of the path, from the last '.' to end of string in the last portion of the path. If there is no '.' in the last portion of the path or the first character of it is '.', then it returns an empty string. |
| 9 | **path.parse(pathString)**<br>Returns an object from a path string. |
| 10 | **path.format(pathObject)**<br>Returns a path string from an object, the opposite of path.parse above. |

Properties

| Sr.No. | Property & Description |
|---|---|
| 1 | **path.sep**<br>The platform-specific file separator. '\\' or '/'. |
| 2 | **path.delimiter**<br>The platform-specific path delimiter, ; or ':'. |
| 3 | **path.posix**<br>Provide access to aforementioned path methods but always interact in a posix compatible way. |
| 4 | **path.win32**<br>Provide access to aforementioned path methods but always interact in a win32 compatible way. |

Example

Create a js file named main.js with the following code −

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

// Normalization
console.log('normalization : ' + path.normalize('/test/test1//2slashes/1slash/tab/..'));

// Join
console.log('joint path : ' + path.join('/test', 'test1', '2slashes/1slash', 'tab', '..'));

// Resolve
console.log('resolve : ' + path.resolve('main.js'));

// extName
console.log('ext name : ' + path.extname('main.js'));
```

Now run the main.js to see the result −

$ node main.js

Verify the Output.

```
normalization : /test/test1/2slashes/1slash
joint path : /test/test1/2slashes/1slash
resolve : /web/com/1427176256_27423/main.js
ext name : .js
```

### 2.4.2 OS module

Node.js **os** module provides a few basic operating-system related utility functions. This module can be imported using the following syntax.

```
var os = require("os")
```

Methods

| Sr.No. | Method & Description |
|---|---|
| 1 | **os.tmpdir()**<br>Returns the operating system's default directory for temp files. |

| | |
|---|---|
| 2 | **os.endianness()**<br>Returns the endianness of the CPU. Possible values are "BE" or "LE". |
| 3 | **os.hostname()**<br>Returns the hostname of the operating system. |
| 4 | **os.type()**<br>Returns the operating system name. |
| 5 | **os.platform()**<br>Returns the operating system platform. |
| 6 | **os.arch()**<br>Returns the operating system CPU architecture. Possible values are "x64", "arm" and "ia32". |
| 7 | **os.release()**<br>Returns the operating system release. |
| 8 | **os.uptime()**<br>Returns the system uptime in seconds. |
| 9 | **os.loadavg()**<br>Returns an array containing the 1, 5, and 15 minute load averages. |
| 10 | **os.totalmem()**<br>Returns the total amount of system memory in bytes. |
| 11 | **os.freemem()** |

| | |
|---|---|
| | Returns the amount of free system memory in bytes. |
| 12 | **os.cpus()**<br>Returns an array of objects containing information about each CPU/core installed: model, speed (in MHz), and times (an object containing the number of milliseconds the CPU/core spent in: user, nice, sys, idle, and irq). |
| 13 | **os.networkInterfaces()**<br>Get a list of network interfaces. |

Properties

| Sr.No. | Property & Description |
|---|---|
| 1 | **os.EOL**<br>A constant defining the appropriate End-of-line marker for the operating system. |

Example

The following example demonstrates a few OS methods. Create a js file named main.js with the following code.

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

// Endianness
console.log('endianness : ' + os.endianness());

// OS type
console.log('type : ' + os.type());

// OS platform
console.log('platform : ' + os.platform());

// Total system memory
console.log('total memory : ' + os.totalmem() + " bytes.");

// Total free memory
console.log('free memory : ' + os.freemem() + " bytes.");
```

Now run the main.js to see the result −

$ node main.js

Verify the Output.

```
endianness : LE
type : Linux
platform : linux
total memory : 25103400960 bytes.
free memory : 20676710400 bytes.
```

### 2.4.3 File system module

Node.js includes **fs** module to access physical file system. The fs module is responsible for all the asynchronous or synchronous file I/O operations.

Let's see some of the common I/O operation examples using fs module.

#### *Reading File*

Use fs.readFile() method to read the physical file asynchronously.

fs.readFile(fileName [,options], callback)

Parameter Description:

- filename: Full path and name of the file as a string.
- options: The options parameter can be an object or string which can include encoding and flag. The default encoding is utf8 and default flag is "r".
- callback: A function with two parameters err and fd. This will get called when readFile operation completes.

The following example demonstrates reading existing TestFile.txt asynchronously.

Example: Reading File

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

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

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

The above example reads TestFile.txt (on Windows) asynchronously and executes callback function when read operation completes. This read operation either throws an error or completes successfully. The err parameter contains error information if any. The data parameter contains the content of the specified file.

The following is a sample TextFile.txt file.

TextFile.txt

```
This is test file to test fs module of Node.js
```

Now, run the above example and see the result as shown below.

```
C:\> node server.js
This is test file to test fs module of Node.js
```

Use fs.readFileSync() method to read file synchronously as shown below.

Example: Reading File Synchronously

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

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

### *Writing File*

Use fs.writeFile() method to write data to a file. If file already exists then it overwrites the existing content otherwise it creates a new file and writes data into it.

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

Parameter Description:

- filename: Full path and name of the file as a string.
- Data: The content to be written in a file.
- options: The options parameter can be an object or string which can include encoding, mode and flag. The default encoding is utf8 and default flag is "r".
- callback: A function with two parameters err and fd. This will get called when write operation completes.

The following example creates a new file called test.txt and writes "Hello World" into it asynchronously.

Example: Creating & Writing File

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

fs.writeFile('test.txt', 'Hello World!', function (err) {
                if (err)
    console.log(err);
                else
    console.log('Write operation complete.');
});
```

In the same way, use fs.appendFile() method to append the content to an existing file.

Example: Append File Content

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

fs.appendFile('test.txt', 'Hello World!', function (err) {
                if (err)
    console.log(err);
                else
    console.log('Append operation complete.');
});
```

### Open File

Alternatively, you can open a file for reading or writing using fs.open() method.

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

Parameter Description:

- path: Full path with name of the file as a string.
- Flag: The flag to perform operation
- Mode: The mode for read, write or readwrite. Defaults to 0666 readwrite.
- callback: A function with two parameters err and fd. This will get called when file open operation completes.

## Flags

The following table lists all the flags which can be used in read/write operation.

| 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. See notes for 'rs' about using this with caution. |
| 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. |
| 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+ | Like 'a+' but fails if path exists. |

The following example opens an existing file and reads its content.

Example:File open and read

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

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

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

                        var buffr = new Buffer(1024);

   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 File

Use fs.unlink() method to delete an existing file.

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

The following example deletes an existing file.

Example:File Open and Read

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

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

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

});
```

Important method of fs module

| Method | Description |
|---|---|
| fs.readFile(fileName [,options], callback) | Reads existing file. |
| fs.writeFile(filename, data[, options], callback) | Writes to the file. If file exists then overwrite the content otherwise creates new file. |
| fs.open(path, flags[, mode], callback) | Opens file for reading or writing. |
| fs.rename(oldPath, newPath, callback) | Renames an existing file. |
| fs.chown(path, uid, gid, callback) | Asynchronous chown. |
| fs.stat(path, callback) | Returns fs.stat object which includes important file statistics. |
| fs.link(srcpath, dstpath, callback) | Links file asynchronously. |
| fs.symlink(destination, path[, type], callback) | Symlink asynchronously. |
| fs.rmdir(path, callback) | Renames an existing directory. |
| fs.mkdir(path[, mode], callback) | Creates a new directory. |
| fs.readdir(path, callback) | Reads the content of the specified directory. |
| fs.utimes(path, atime, mtime, callback) | Changes the timestamp of the file. |

| Method | Description |
|---|---|
| fs.exists(path, callback) | Determines whether the specified file exists or not. |
| fs.access(path[, mode], callback) | Tests a user's permissions for the specified file. |
| fs.appendFile(file, data[, options], callback) | Appends new content to the existing file. |

## Get File Information

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

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

## Syntax

Following is the syntax of the method to get the information about a file −

fs.stat(path, callback)

## Parameters

Here is the description of the parameters used −

- **path** − This is the string having file name including path.
- **callback** − This is the callback function which gets two arguments (err, stats) where **stats** is an object of fs.Stats type which is printed below in the example.

Apart from the important attributes which are printed below in the example, there are several useful methods available in **fs.Stats** class which can be used to check file type. These methods are given in the following table.

| Sr.No. | Method & Description |
|---|---|
| 1 | **stats.isFile()**<br>Returns true if file type of a simple file. |
| 2 | **stats.isDirectory()**<br>Returns true if file type of a directory. |

| | |
|---|---|
| 3 | **stats.isBlockDevice()**<br><br>Returns true if file type of a block device. |
| 4 | **stats.isCharacterDevice()**<br><br>Returns true if file type of a character device. |
| 5 | **stats.isSymbolicLink()**<br><br>Returns true if file type of a symbolic link. |
| 6 | **stats.isFIFO()**<br><br>Returns true if file type of a FIFO. |
| 7 | **stats.isSocket()**<br><br>Returns true if file type of asocket. |

### Example

Let us create a js file named **main.js** with the following code −

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

console.log("Going to get file info!");
fs.stat('input.txt', function (err, stats) {
   if (err) {
      return console.error(err);
   }
   console.log(stats);
   console.log("Got file info successfully!");

   // Check file type
   console.log("isFile ? " + stats.isFile());
   console.log("isDirectory ? " + stats.isDirectory());
});
```

Now run the main.js to see the result −

```
$ node main.js
```

Verify the Output.

```
Going to get file info!
{
  dev: 1792,
  mode: 33188,
  nlink: 1,
  uid: 48,
  gid: 48,
  rdev: 0,
  blksize: 4096,
  ino: 4318127,
  size: 97,
  blocks: 8,
  atime: Sun Mar 22 2015 13:40:00 GMT-0500 (CDT),
  mtime: Sun Mar 22 2015 13:40:57 GMT-0500 (CDT),
  ctime: Sun Mar 22 2015 13:40:57 GMT-0500 (CDT)
}
Got file info successfully!
isFile ? true
isDirectory ? false
```

### 2.4.4 Events module

**EventEmitter** is the implementation of **Node.js's pub-sub** design patterns. Node.js core API has built on an **asynchronous event-driven** architecture. In an **asynchronous event architecture,** certain kinds of objects (called "emitters") periodically emit named events that cause **Function objects** ("listeners") to be called.

Each object that emits **events** are instances of an **EventEmitter** class.

We are going to see one example with ES5 Syntax.

### Node.js Events EventEmitter

Example #1

```
// ES5 Event Emitter Example


var EventEmitter = require('events');
```

```
var emitter = new EventEmitter();


emitter.on('newEvent', function(user){
    console.log(user);
});


emitter.emit('newEvent', "Krunal");
```

In above example, first, we are going to import **events** object, and then we get **emitter** object.

**The event emitter** class has two methods.

1. On
2. emit

So, if we make an object of **EventEmitter** class, then we have access to this methods.

```
emitter.on('newEvent', function(user){
    console.log(user);
});
```

Here, I have defined an event and till now, not called just determine.

**On()** method takes an event name and a call back function, which describes the logic and payload of the function. As we know, Node.js has the event-driven architecture, so it first occurs the events and then related to that event, one callback function is returned EMIT.

```
emitter.emit('newEvent', "Krunal");
```

Here, I have emitted the event, so related to that event, the callback function is invoked, and that function will execute. The first parameter is event name and second is payload.

The output will be following.

```
Krunal
```

Example #2

```
var EventEmitter = require('events');
var util = require('util');

var User = function(username){
    this.username = username;
}
util.inherits(User, EventEmitter);
var user = new User('Krunal Lathiya');
user.on('nuevent', function(props){
    console.log(props);
});
user.emit('nuevent', 'dancing');
```

In this example, first I have created an **EventEmitter** object and then also create **User function constructor**.

Then, import the Node.js's core module **util** and inherits the base functionality from EventEmitter module to the newly created User module.

So, now User has all the methods and properties of the **EventEmitter** module, and we can use two methods on it.

Now, user object's behavior is same as EventEmitter, and we can define events on it and emit the events.

The output will be "**dancing**."

Example #3

```
const EventEmitter = require('events');

class MyEmitter extends EventEmitter {}

const myEmitter = new MyEmitter();
```

```
myEmitter.on('event', () => {
  console.log('an event occurred!');
});
myEmitter.emit('event');
```

Above is an **ES6** example of **Node.js events and event emitter**.

First, we have included the **EventEmitter** class, and then we define our class and extends the base **EventEmitter** class.

Now, make an object of the newly created class, so we have access all the methods of the **EventEmitter**class. Rest is all the same.

### 2.4.5 HTTP module

In this section, we will learn how to create a simple Node.js web server and handle HTTP requests.
To access web pages of any web application, you need a web server. The web server will handle all the http requests for the web application e.g IIS is a web server for ASP.NET web applications and Apache is a web server for PHP or Java web applications.
Node.js provides capabilities to create your own web server which will handle HTTP requests asynchronously. You can use IIS or Apache to run Node.js web application but it is recommended to use Node.js web server.

#### Create Node.js Web Server

Node.js makes it easy to create a simple web server that processes incoming requests asynchronously.
The following example is a simple Node.js web server contained in server.js file.
server.js

```
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.

Run the above web server by writing node server.js command in command prompt or terminal window and it will display message as shown below.

```
C:\> node server.js
Node.js web server at port 5000 is running..
```

This is how you create a Node.js web server using simple steps. Now, let's see how to handle HTTP request and send response in Node.js web server.

### Handle HTTP Request

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

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

server.js

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

var server = http.createServer(function (req, res) {   //create web server
    if (req.url == '/') { //check the URL of the current request

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

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

    }
    else if (req.url == "/student") {

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

    }
    else if (req.url == "/admin") {

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

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

});

server.listen(5000); //6 - listen for any incoming requests

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

In the above example, req.url is used to check the url of the current request and based on that it sends the response. To send a response, first it sets the response header using writeHead() method and then writes a string as a response body using write() method. Finally, Node.js web server sends the response using end() method.

Now, run the above web server as shown below.

```
C:\> node server.js
Node.js web server at port 5000 is running..
```

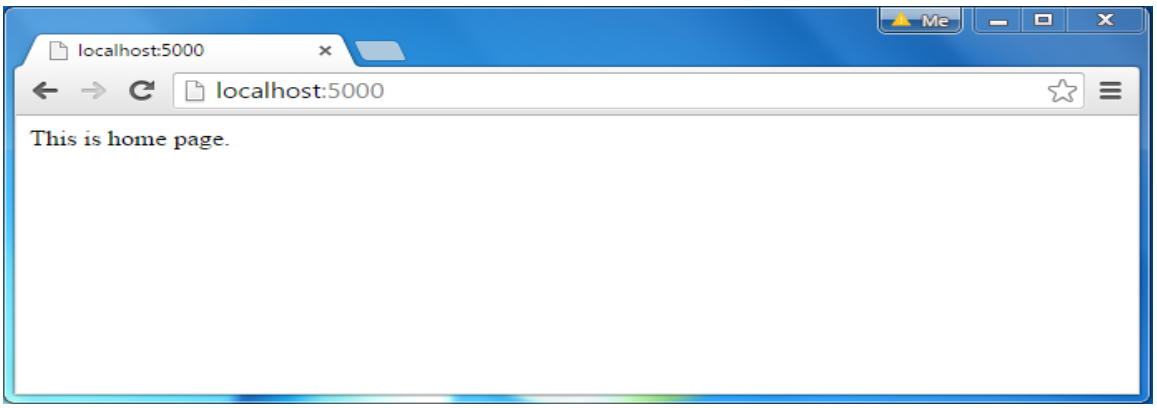
To test it, you can use the command-line program curl, which most Mac and Linux machines have pre-installed.

```
curl -i http://localhost:5000
```

You should see the following response.

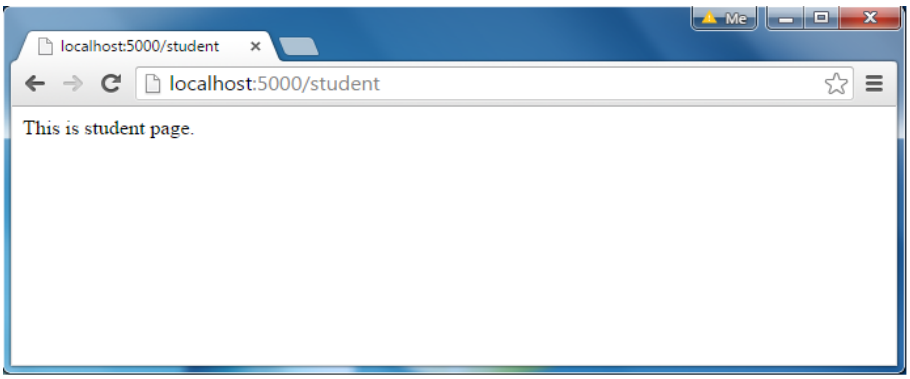
```
HTTP/1.1 200 OK
Content-Type: text/plain
Date: Tue, 8 Sep 2015 03:05:08 GMT
Connection: keep-alive
This is home page.
```

For Windows users, point your browser to *http://localhost:5000* and see the following result.




Node.js Web Server Response

The same way, point your browser to *http://localhost:5000/student* and see the following result.




Node.js Web Server Response

It will display "Invalid Request" for all requests other than the above URLs.

### Sending JSON Response

The following example demonstrates how to serve JSON response from the Node.js web server.

server.js

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

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

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

server.listen(5000);

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

So, this way you can create a simple web server that serves different responses.

### 2.4.6 Node.js URL Module

URL is a nodejs built-in Module. The URL module splits up a web address into readable parts.

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

```
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:

Example

Split a web address into readable parts:

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

var adr = 'http://localhost:8080/default.htm?year=2017&month=february';

var q = url.parse(adr, true);

console.log(q.host); //returns 'localhost:8080'

console.log(q.pathname); //returns '/default.htm'

console.log(q.search); //returns '?year=2017&month=february'

var qdata = q.query; //returns an object: { year: 2017, month: 'february' }

console.log(qdata.month); //returns 'february'
```

### Example combining http, fs and URL modules

Now we know how to parse the query string, and in the previous chapter we learned how to make Create two html files and save them in the same folder as your node.js files.

**summer.html**

<!DOCTYPE html>

<html>

<body>

<h1>Summer</h1>

I love the sun!

</body>

</html>

**winter.html**

<!DOCTYPE html>

<html>

<body>

<h1>Winter</h1>

I love the snow!

</body>

</html>

Create a Node.js file that opens the requested file and returns the content to the client. If anything goes wrong, throw a 404 error:

**demo_fileserver.js**

```
var http = require('http');
var url = require('url');
var fs = require('fs');
http.createServer(function  (req, res) {
  var q = url.parse(req.url, true);
  var filename = "." + q.pathname;
  fs.readFile(filename, function(err, data) {
    if (err) {
      res.writeHead(404, {'Content-Type': 'text/html'});
      return res.end("404 Not Found");
    }
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    return res.end();
  });
}).listen(8080);
```

### 2.5 Node package manager (npm)

#### 2.5.1 Introduction

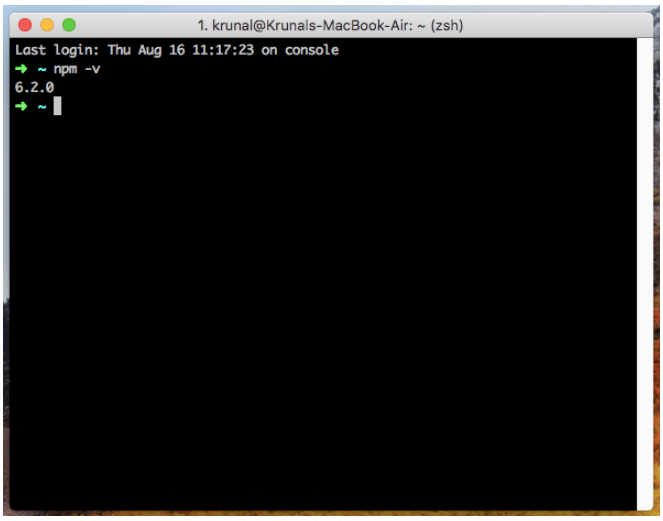
**NPM** is the biggest repository for any programming language, and it has almost every package that you need in a web or mobile development project. **npm** means the **node package manager**. In January 2017 over**350,000** packages were reported being listed in the npm registry, making it the most significant single language code repository on the planet. At the starting point **Node Package Manager** is only for **Node.js** develop-

ment and nothing else, but as Javascript community grows, it becomes the backbone of almost all the latest **JavaScript** Frameworks and Web development. The **npm** registry hosts the world's most extensive collection of free, reusable code.

**NPM** is written entirely in **JavaScript**(JS) and was developed by **Isaac Z. Schlueter** with inspiration from the shortcomings of other similar projects such as PHP) and CPAN (Perl).

If you have installed **Node.js** in your machine then by default **NPM** has already been installed. The-**Node.js** comes with **NPM**.

You can check the version of your **NPM** using the following command.




All of your dependencies are written in one file inside your project root called **package.json.** You can create the **package.json** file using the following command below.

```
npm init
```

If you have already the **package.json** file, but dependencies are not installed previously then you can install all the dependencies defined on your package.json file using the following command.

```
npm install
```

If you want to install development dependency, then you can hit the following command with the **–dev** flag.

```
npm install <package-name> --save-dev

# or
```

```
npm install -D <package-name>
```

The difference between **dependency** and **devDependencies** are usually development tools, like a testing library, while **dependencies** are bundled with the app in production.

If you do not want to write **install**, then you can use **i**, and it will do the work for you.

```
npm i <package-name>
```

If you want to save the dependency inside the **package.json** file**,** then you need to add the **–save** flag.

```
npm i <package-name> --save
```

**NPM** will generate one folder called **node_modules** when you install any library from the **Node Package Manager** Repository.

**2.5.2 npm Usage**

**NPM** can manage packages that are local dependencies of a particular project, as well as globally-installed JavaScript tools. We can use as a dependency manager for our local **project**. You can update **NPM** packages using the following command.

```
npm update
```

You can also update the specific package using the following command.

```
npm update <package-name>
```

If you are running an old version of NPM, then you can update it to the latest release by the following command from root.

```
sudo npm install npm -g
```

**2.5.3 NPM Versioning.**

In addition to plain downloads, **node package manager** also manages the **versioning** so that you can specify any specific version of a package.

If you specify the exact version of the **NPM** libraries, then it also helps to keep everyone on the same version of a library, so that the whole team runs the same version and no conflicts occur until the **package.json** file is updated.

If you are using the Git version control system, then you need to upload the **package.json** file and not **a node_modules** folder. So when another developer downloads the project, it has already the **package.json** file, and he only needs to hit the **npm install** command to up and running with the project.

If we do not specify any version, then it will install the latest version of the particular package.

**Semantic Versioning**

If there's one great thing in Node.js packages, is that all agreed on using Semantic Versioning for their version numbering.The Semantic Versioning concept is simple: all versions have 3 digits**: x.y.z**.

- the first digit is the major version
- the second digit is the minor version
- the third digit is the patch version

When you make a new release, you don't just **up** a number as you please, but you have rules:

- you up the major version when you make incompatible API changes
- you up the minor version when you add functionality in a backward-compatible manner
- you up the patch version when you make backward-compatible bug fixes

The convention is adopted all across programming languages, and it is very important that every npm package adheres to it, because the whole system depends on that.

**Why is that so important?**

Because npm set some rules we can use in the package.json file to choose which versions it can update our packages to, when we run npm update.

The rules use those symbols:

- **^**
- **~**
- **>**
- **>=**
- **<**

- **<=**
- **=**
- **-**
- **||**

Let's see those rules in detail:

- ^: if you write ^0.13.0 when running npm update it can update to patch and minor releases: 0.13.1, 0.14.0 and so on.
- ~: if you write ~0.13.0, when running npm update it can update to patch releases: 0.13.1 is ok, but 0.14.0 is not.
- >: you accept any version higher than the one you specify
- >=: you accept any version equal to or higher than the one you specify
- <=: you accept any version equal or lower to the one you specify
- <: you accept any version lower to the one you specify
- =: you accept that exact version
- -: you accept a range of versions. Example: 2.1.0 - 2.6.2
- ||: you combine sets. Example: < 2.1 || > 2.6

You can combine some of those notations, for example use 1.0.0 || >=1.1.0 <1.2.0 to either use 1.0.0 or one release from 1.1.0 up, but lower than 1.2.0.

**There are other rules, too:**

- **no symbol**: you accept only that specific version you specify (1.2.1)
- **latest**: you want to use the latest version available

### 2.5.4 Global vs. Local Installation.

By default, **NPM** installs any dependency in the local mode. Here local mode refers to the package installation in the **node_modules** directory inside our working project. Locally deployed packages are accessible via require() method. For example, when we installed an express module, it created a**node_modules** directory in the current directory where it installed the express module.

```
$ npm install express

Now you can use this module in your js file as following

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

Globally installed packages are stored in the system directory. Such dependencies can be used in **CLI** (Command Line Interface) function of any **node.js** but cannot be imported using **require**() in Node application directly. Now let's try installing the express module using global installation.

```
npm install express -g
```

It will produce a similar result, but the module will be installed globally.

**Uninstalling a Module**

Use the following command to uninstall a Node.js module.

```
npm uninstall <package-name>
```

**Search a Module**

Search a package name using **NPM**.

```
npm search express
```

**Running Tasks**

The package.json file supports a format for specifying command line tasks that can be run by using

```
npm <task-name>
```

For example:

```
{
```

```
  "scripts": {
    "start-dev": "node lib/server-development",
    "start": "node lib/server-production"
  },
}
```

It's very common to use this feature to run **Webpack**:

```
{
  "scripts": {
    "watch": "webpack --watch --progress --colors --config webpack.conf.js",
    "dev": "webpack --progress --colors --config webpack.conf.js",
    "prod": "NODE_ENV=production webpack -p --config webpack.conf.js",
  },
}
```

So instead of typing those extended commands, which are easy to forget or mistype, you can run

```
$ npm watch
```

```
$ npm dev
```

**2.5.5 Publishing custom modules**

Using npm we can publish our our modules, follow this steps to publish you module:

Step1: Create project and export some functions or object
Step2: run npm init – answer prompts questions accordingly

Step3: run npm login – signup if you do not have an account

Step4: run npm publish – Append some numbers to you module name if you get an error

Step5: Create another project and run npm install your-module and use it in your new project

Later if you make changes and want to publish update version run ***npm version*** {***major*** or ***minor*** or ***patch***} depending on the changes made, then run ***npm publish***

### 2.5.6 Version Control

Version control offers these benefits:

**Documentation**

Being able to go back through the history of a project to see the decisions that were made and the order in which components were developed can be valuable documentation. Having a technical history of your project can be quite useful.

**Attribution**

If you work on a team, attribution can be hugely important. Whenever you find something in code that is opaque or questionable, knowing who made that change can save you many hours. It could be that the comments associated with the change are sufficient to answer your questions, and if not, you'll know who to talk to.

**Experimentation**

A good version control system enables experimentation. You can go off on a tangent, trying something new, without fear of affecting the stability of your project. If the experiment is successful, you can fold it back into the project, and if it is not successful, you can abandon it.

We will be using Git, but you are welcome to substitute it with any other version control of your choice, e.g svn, mercurial,...

**Importing a new project**

Assume you have a project folder with your initial work. You can place it under Git revision control as follows.

```
$ mkdir project

$ cd project

$ git init
```

Git will reply

```
Initialized empty Git repository in .git/
```

You've now initialized the working directory—you may notice a new directory created, named ".git".

Next, tell Git to take a snapshot of the contents of all files under the current directory (note the **.**), with **git add**:

```
$ git add .
```

This snapshot is now stored in a temporary staging area which Git calls the "index". You can permanently store the contents of the index in the repository with **git commit**:

```
$ git commit
```

This will prompt you for a commit message. You've now stored the first version of your project in Git.

**Making changes**

Modify some files, then add their updated contents to the index:

```
$ git add file1 file2 file3
```

You are now ready to commit. You can see what is about to be committed using **git diff** with the --cached option:

```
$ git diff --cached
```

(Without --cached, **git diff** will show you any changes that you've made but not yet added to the index.) You can also get a brief summary of the situation with **git status**:

```
$ git status

On branch master

Changes to be committed:

Your branch is up to date with 'origin/master'.

 (use "git restore --staged <file>..." to unstage)

            modified:   file1

            modified:   file2

            modified:   file3
```

If you need to make any further adjustments, do so now, and then add any newly modified content to the index. Finally, commit your changes with:

```
$ git commit
```

This will again prompt you for a message describing the change, and then record a new version of the project.

Alternatively, instead of running **git add** beforehand, you can use

```
$ git commit -a
```

which will automatically notice any modified (but not new) files, add them to the index, and commit, all in one step.

A note on commit messages: Though not required, it's a good idea to begin the commit message with a single short (less than 50 character) line summarizing the change, followed by a blank line and then a more thorough description. The text up to the first blank line in a commit message is treated as the commit title, and that title is used throughout Git.

**Git tracks content not files**

Many revision control systems provide an add command that tells the system to start tracking changes to a new file. Git's add command does something simpler and more powerful: **git add** is used both for new and newly modified files, and in both cases it takes a snapshot of the given files and stages that content in the index, ready for inclusion in the next commit.

**Viewing project history**

At any point you can view the history of your changes using

```
$ git log
```

If you also want to see complete diffs at each step, use

```
$ git log -p
```

Often the overview of the change is useful to get a feel of each step

```
$ git log --stat --summary
```

**Managing branches**

A single Git repository can maintain multiple branches of development. To create a new branch named "experimental", use

```
$ git branch experimental
```

If you now run

```
$ git branch
```

you'll get a list of all existing branches:

```
  experimental
* master
```

The "experimental" branch is the one you just created, and the "master" branch is a default branch that was created for you automatically. The asterisk marks the branch you are currently on; type

```
$ git switch experimental
```

to switch to the experimental branch. Now edit a file, commit the change, and switch back to the master branch:

```
(edit file)
$ git commit -a
$ git switch master
```

Check that the change you made is no longer visible, since it was made on the experimental branch and you're back on the master branch.

You can make a different change on the master branch:

```
(edit file)
$ git commit -a
```

at this point the two branches have diverged, with different changes made in each. To merge the changes made in experimental into master, run

```
$ git merge experimental
```

If the changes don't conflict, you're done. If there are conflicts, markers will be left in the problematic files showing the conflict;

```
$ git diff
```

will show this. Once you've edited the files to resolve the conflicts,

```
$ git commit -a
```

will commit the result of the merge. Finally,

```
$ gitk
```

will show a nice graphical representation of the resulting history.

At this point you could delete the experimental branch with

```
$ git branch -d experimental
```

This command ensures that the changes in the experimental branch are already in the current branch.

If you develop on a branch crazy-idea, then regret it, you can always delete the branch with

```
$ git branch -D crazy-idea
```

Branches are cheap and easy, so this is a good way to try something out.

**Using Git for collaboration**

Suppose that Alice has started a new project with a Git repository in /home/alice/project, and that Bob, who has a home directory on the same machine, wants to contribute.

Bob begins with:

```
bob$ git clone /home/alice/project myrepo
```

This creates a new directory "myrepo" containing a clone of Alice's repository. The clone is on an equal footing with the original project, possessing its own copy of the original project's history.

Bob then makes some changes and commits them:

```
(edit files)

bob$ git commit -a
```

```
(repeat as necessary)
```

When he’s ready, he tells Alice to pull changes from the repository at /home/bob/myrepo. She does this with:

```
alice$ cd /home/alice/project

alice$ git pull /home/bob/myrepo master
```

This merges the changes from Bob’s "master" branch into Alice’s current branch. If Alice has made her own changes in the meantime, then she may need to manually fix any conflicts.

The "pull" command thus performs two operations: it fetches changes from a remote branch, then merges them into the current branch.

Note that in general, Alice would want her local changes committed before initiating this "pull". If Bob’s work conflicts with what Alice did since their histories forked, Alice will use her working tree and the index to resolve conflicts, and existing local changes will interfere with the conflict resolution process (Git will still perform the fetch but will refuse to merge --- Alice will have to get rid of her local changes in some way and pull again when this happens).

Alice can peek at what Bob did without merging first, using the "fetch" command; this allows Alice to inspect what Bob did, using a special symbol "FETCH_HEAD", in order to determine if he has anything worth pulling, like this:

```
alice$ git fetch /home/bob/myrepo master

alice$ git log -p HEAD..FETCH_HEAD
```

This operation is safe even if Alice has uncommitted local changes. The range notation "HEAD..FETCH_HEAD" means "show everything that is reachable from the FETCH_HEAD but exclude anything that is reachable from HEAD". Alice already knows everything that leads to her current state (HEAD), and reviews what Bob has in his state (FETCH_HEAD) that she has not seen with this command.

If Alice wants to visualize what Bob did since their histories forked she can issue the following command:

```
$ gitk HEAD..FETCH_HEAD
```

This uses the same two-dot range notation we saw earlier with **git log**.

Alice may want to view what both of them did since they forked. She can use three-dot form instead of the two-dot form:

```
$ gitk HEAD...FETCH_HEAD
```

This means "show everything that is reachable from either one, but exclude anything that is reachable from both of them".

Please note that these range notation can be used with both gitk and "git log".

After inspecting what Bob did, if there is nothing urgent, Alice may decide to continue working without pulling from Bob. If Bob's history does have something Alice would immediately need, Alice may choose to stash her work-in-progress first, do a "pull", and then finally unstash her work-in-progress on top of the resulting history.

When you are working in a small closely knit group, it is not unusual to interact with the same repository over and over again. By defining **remote** repository shorthand, you can make it easier:

```
alice$ git remote add bob /home/bob/myrepo
```

With this, Alice can perform the first part of the "pull" operation alone using the **git fetch** command without merging them with her own branch, using:

```
alice$ git fetch bob
```

Unlike the longhand form, when Alice fetches from Bob using a remote repository shorthand set up with **git remote**, what was fetched is stored in a remote-tracking branch, in this case bob/master. So after this:

```
alice$ git log -p master..bob/master
```

shows a list of all the changes that Bob made since he branched from Alice's master branch.

After examining those changes, Alice could merge the changes into her master branch:

```
alice$ git merge bob/master
```

This merge can also be done by **pulling from her own remote-tracking branch**, like this:

```
alice$ git pull . remotes/bob/master
```

Note that git pull always merges into the current branch, regardless of what else is given on the command line.

Later, Bob can update his repo with Alice's latest changes using

```
bob$ git pull
```

Note that he doesn’t need to give the path to Alice’s repository; when Bob cloned Alice’s repository, Git stored the location of her repository in the repository configuration, and that location is used for pulls:

```
bob$ git config --get remote.origin.url

/home/alice/project
```

(The complete configuration created by **git clone** is visible using git config -l, and the git-config[1] man page explains the meaning of each option.)
Git also keeps a pristine copy of Alice’s master branch under the name "origin/master":

```
bob$ git branch -r

  origin/master
```

If Bob later decides to work from a different host, he can still perform clones and pulls using the ssh protocol:

```
bob$ git clone alice.org:/home/alice/project myrepo
```

**Exploring history**

Git history is represented as a series of interrelated commits. We have already seen that the **git log** command can list those commits. Note that first line of each git log entry also gives a name for the commit:

```
$ git log

commit c82a22c39cbc32576f64f5c6b3f24b99ea8149c7

Author: Junio C Hamano <junkio@cox.net>

Date:   Tue May 16 17:18:22 2006 -0700


    merge-base: Clarify the comments on post processing.
```

We can give this name to **git show** to see the details about this commit.

```
$ git show c82a22c39cbc32576f64f5c6b3f24b99ea8149c7
```

But there are other ways to refer to commits. You can use any initial part of the name that is long enough to uniquely identify the commit:

```
$ git show c82a22c39c       # the first few characters of the name are
                            # usually enough
$ git show HEAD             # the tip of the current branch
$ git show experimental     # the tip of the "experimental" branch
```

Every commit usually has one "parent" commit which points to the previous state of the project:

```
$ git show HEAD^  # to see the parent of HEAD
$ git show HEAD^^ # to see the grandparent of HEAD
$ git show HEAD~4 # to see the great-great grandparent of HEAD
```

Note that merge commits may have more than one parent:

```
$ git show HEAD^1 # show the first parent of HEAD (same as HEAD^)
$ git show HEAD^2 # show the second parent of HEAD
```

You can also give commits names of your own; after running

```
$ git tag v2.5 1b2e1d63ff
```

you can refer to 1b2e1d63ff by the name "v2.5". If you intend to share this name with other people (for example, to identify a release version), you should create a "tag" object, and perhaps sign it; see git-tag[1]for details.

Any Git command that needs to know a commit can take any of these names. For example:

```
$ git diff v2.5 HEAD        # compare the current HEAD to v2.5
$ git branch stable v2.5 # start a new branch named "stable" based
                         # at v2.5
```

```
$ git reset --hard HEAD^ # reset your current branch and working
                         # directory to its state at HEAD^
```

Be careful with that last command: in addition to losing any changes in the working directory, it will also remove all later commits from this branch. If this branch is the only branch containing those commits, they will be lost. Also, don't use **git reset** on a publicly-visible branch that other developers pull from, as it will force needless merges on other developers to clean up the history. If you need to undo changes that you have pushed, use **git revert** instead.

The **git grep** command can search for strings in any version of your project, so

```
$ git grep "hello" v2.5
```

searches for all occurrences of "hello" in v2.5.

If you leave out the commit name, **git grep** will search any of the files it manages in your current directory. So

```
$ git grep "hello"
```

is a quick way to search just the files that are tracked by Git.

Many Git commands also take sets of commits, which can be specified in a number of ways. Here are some examples with **git log**:

```
$ git log v2.5..v2.6            # commits between v2.5 and v2.6
$ git log v2.5..                # commits since v2.5
$ git log --since="2 weeks ago" # commits from the last 2 weeks
$ git log v2.5.. Makefile       # commits since v2.5 which modify
                                # Makefile
```

You can also give **git log** a "range" of commits where the first is not necessarily an ancestor of the second; for example, if the tips of the branches "stable" and "master" diverged from a common commit some time ago, then

```
$ git log stable..master
```

will list commits made in the master branch but not in the stable branch, while

```
$ git log master..stable
```

will show the list of commits made on the stable branch but not the master branch.

The **git log** command has a weakness: it must present commits in a list. When the history has lines of development that diverged and then merged back together, the order in which **git log** presents those commits is meaningless.

Most projects with multiple contributors (such as the Linux kernel, or Git itself) have frequent merges, and **gitk** does a better job of visualizing their history. For example,

```
$ gitk --since="2 weeks ago" drivers/
```

allows you to browse any commits from the last 2 weeks of commits that modified files under the "drivers" directory. (Note: you can adjust gitk's fonts by holding down the control key while pressing "-" or "+".)

Finally, most commands that take filenames will optionally allow you to precede any filename by a commit, to specify a particular version of the file:

```
$ git diff v2.5:Makefile HEAD:Makefile.in
```

You can also use **git show** to see any such file:

```
$ git show v2.5:Makefile
```

**CHAPTER THREE: Express.js**

**3.1 Frameworks for Node.js**

In previous lectures we saw that we need to write lots of low level code ourselves to create a web application using Node.js.

There are various third party open-source frameworks available in Node Package Manager which makes Node.js application development faster and easy. You can choose an appropriate framework as per your application requirements.

The following table lists frameworks for Node.js.

| Open-Source Framework | Description |
|---|---|
| Express.js | Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. This is the most popular framework as of now for Node.js. |
| Geddy | Geddy is a simple, structured web application framework for Node.js based on MVC architecture. |
| Locomotive | Locomotive is MVC web application framework for Node.js. It supports MVC patterns, RESTful routes, and convention over configuration, while integrating seamlessly with any database and template engine. Locomotive builds on Express, preserving the power and simplicity you've come to expect from Node. |
| Koa | Koa is a new web framework designed by the team behind Express, which aims to be a smaller, more expressive, and more robust foundation for web applications and APIs. |
| Total.js | Totaljs is free web application framework for building web sites and web applications using JavaScript, HTML and CSS on Node.js |
| Hapi.js | Hapi is a rich Node.js framework for building applications and services. |
| Keystone | Keystone is the open source framework for developing database-driven websites, applications and APIs in Node.js. Built on Express and MongoDB. |
| Derbyjs | Derby support single-page apps that have a full MVC structure, including a model provided byÂ Racer, a template and styles based view, and controller code with application logic and routes. |
| Sails.js | Sails makes it easy to build custom, enterprise-grade Node.js apps. It is designed to emulate the familiar MVC pattern of frameworks like Ruby on Rails, but with support for the requirements of modern apps: data-driven APIs with a scalable, service-oriented architecture. It's especially good for building chat, realtime dashboards, or multiplayer games; but you can use it for any web application project - top to bottom. |
| Meteor | Meteor is a complete open source platform for building web and mobile apps in pure |

| Open-Source Framework | Description |
|---|---|
| | JavaScript. |
| Mojito | This HTML5 framework for the browser and server from Yahoo offers direct MVC access to the server database through the local routines. One clever feature allows the code to migrate. If the client can't run JavaScript for some reason, Mojito will run it on the server – a convenient way to handle very thin clients. |
| Restify | Restify is a node.js module built specifically to enable you to build correct REST web services. |
| Loopback | Loopback is an open-source Node.js API framework. |
| ActionHero | actionhero.js is a multi-transport Node.JS API Server with integrated cluster capabilities and delayed tasks. |
| Frisby | Frisby is a REST API testing framework built on node.js and Jasmine that makes testing API endpoints easy, fast, and fun. |
| Chocolate.js | Chocolate is a simple webapp framework built on Node.js using Coffeescript. |

### 3.2 Express.js introduction

Express is a fast, unopinionated minimalist web framework for Node.js - official web site: Expressjs.com

Express.js is a web application framework for Node.js. It provides various features that make web application development fast and easy which otherwise takes more time using only Node.js.

Express.js is based on the Node.js middleware module called ***connect*** which in turn uses **http** module. So, any middleware which is based on connect will also work with Express.js.

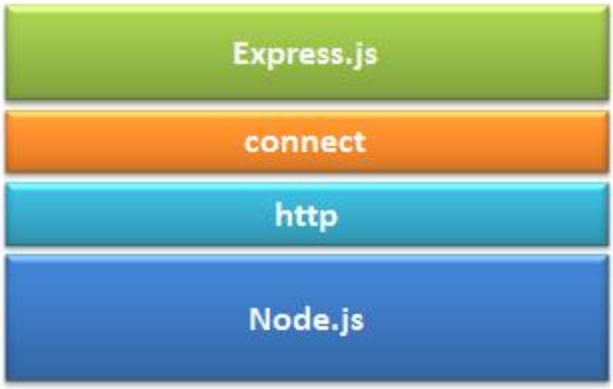



Fig: Express.js

### 3.3 Advantages of Express.js

1. Makes Node.js web application development fast and easy.
2. Easy to configure and customize.

3. Allows you to define routes of your application based on HTTP methods and URLs.
4. Includes various middleware modules which you can use to perform additional tasks on request and response.
5. Easy to integrate with different template engines like Jade, Vash, EJS etc.
6. Allows you to define an error handling middleware.
7. Easy to serve static files and resources of your application.
8. Allows you to create REST API server.
9. Easy to connect with databases such as MongoDB, Redis, MySQL

**3.4 Install Express.js**

You can install express.js using npm. The following command will install latest version of express.js globally on your machine so that every Node.js application on your machine can use it.

```
npm install -g express
```

The following command will install latest version of express.js local to your project folder.

```
C:\MyNodeJSApp> npm install express --save
```

As you know, -- save will update the package.json file by specifying express.js dependency.

**3.5 Express.js Web Application**

In this section, you will learn how to create a web application using Express.js.

Express.js provides an easy way to create web server and render HTML pages for different HTTP requests by configuring routes for your application.

**3.5.2 Web Server**

First of all, import the Express.js module and create the web server as shown below.

app.js: Express.js Web Server

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

// define routes here..

var server = app.listen(5000, function () {
    console.log('Node server is running..');
});
```

In the above example, we imported Express.js module using require() function. The express module returns a function. This function returns an object which can be used to configure Express application (app in the above example).

The app object includes methods for routing HTTP requests, configuring middleware, rendering HTML views and registering a template engine.

The app.listen() function creates the Node.js web server at the specified host and port. It is identical to Node's http.Server.listen() method.

Run the above example using `node app.js` command and point your browser to *http://localhost:5000*. It will display **Cannot GET /** because we have not configured any routes yet.

### 3.5.3 Configure Routes

Use app object to define different routes of your application. The app object includes get(), post(), put() and delete() methods to define routes for HTTP GET, POST, PUT and DELETE requests respectively.

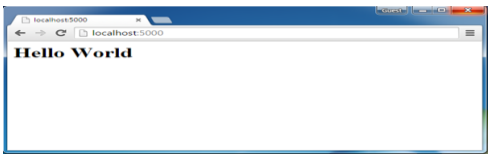
The following example demonstrates configuring routes for HTTP requests.

Example: Configure Routes in Express.js

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

app.get('/', function (req, res) {
    res.send('<html><body><h1>Hello World</h1></body></html>');
});

app.post('/submit-data', function (req, res) {
    res.send('POST Request');
});

app.put('/update-data', function (req, res) {
    res.send('PUT Request');
});

app.delete('/delete-data', function (req, res) {
    res.send('DELETE Request');
});

var server = app.listen(5000, function () {
    console.log('Node server is running..');
});
```

In the above example, app.get(), app.post(), app.put() and app.delete() methods define routes for HTTP GET, POST, PUT, DELETE respectively. The first parameter is a path of a route which will start after base URL. The callback function includes request and response object which will be executed on each request.

Run the above example using `node server.js` command, and point your browser to *http://localhost:5000* and you will see the following result.




Express.js Web Application

### 3.5.4 Handle POST Request

Here, you will learn how to handle HTTP POST request and get data from the submitted form.

First, create Index.html file in the root folder of your application and write the following HTML code in it.

Example: Configure Routes in Express.js

```
<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta charset="utf-8" />
    <title></title>
</head>
<body>
    <form action="/submit-student-data" method="post">
        First Name: <input name="firstName" type="text" /> <br />
        Last Name: <input name="lastName" type="text" /> <br />
        <input type="submit" />
    </form>
</body>
</html>
```

**Body Parser**

To handle HTTP POST request in Express.js version 4 and above, you need to install middleware module called body-parser. The middleware was a part of Express.js earlier but now you have to install it separately.

This body-parser module parses the JSON, buffer, string and url encoded data submitted using HTTP POST request. Install body-parser using NPM as shown below.

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

Now, import body-parser and get the POST request data as shown below.

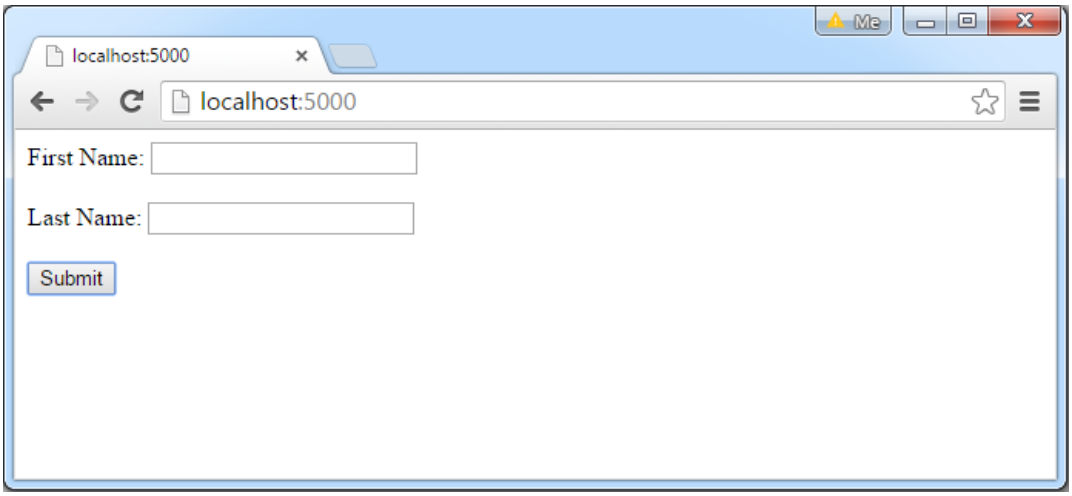
app.js: Handle POST Route in Express.js

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

var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
```

```
app.get('/', function (req, res) {
   res.sendFile('index.html');
});

app.post('/submit-student-data', function (req, res) {
   var name = req.body.firstName + ' ' + req.body.lastName;
   res.send(name + ' Submitted Successfully!');
});

var server = app.listen(5000, function () {
   console.log('Node server is running..');
});
```

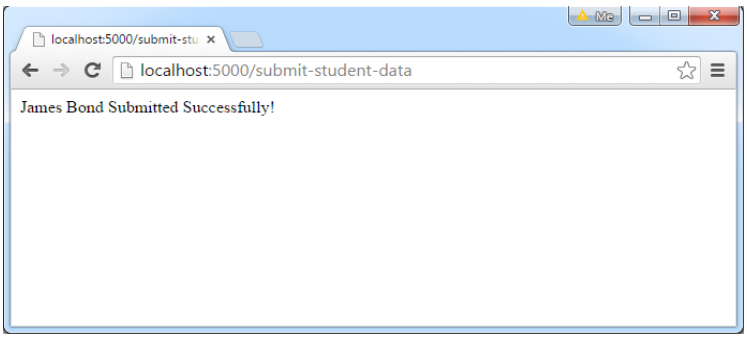
In the above example, POST data can be accessed using req.body. The req.body is an object that includes properties for each submitted form. Index.html contains firstName and lastName input types, so you can access it using req.body.firstName and req.body.lastName. Third party libraries can be used to validate user input, example is **joi**.

Now, run the above example using `node server.js` command, point your browser to *http://localhost:5000* and see the following result.




HTML Form to submit POST request

Fill the First Name and Last Name in the above example and click on **submit**. For example, enter "James" in First Name textbox and "Bond" in Last Name textbox and click the submit button. The following result is displayed.




Response from POST request

### 3.6 Serving Static Resources in Node.js

In this section, you will learn how to serve static resources like images, css, JavaScript or other static files using **Express.js** and **node-static** module.

#### 3.6.2 Serve Static Resources using Express.js

It is easy to serve static files using built-in middleware in Express.js called express.static. Using express.static() method, you can server static resources directly by specifying the folder name where you have stored your static resources.

The following example serves static resources from the public folder under the root folder of your application.

server.js

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

//setting middleware
app.use(express.static(__dirname + 'public')); //Serves resources from public folder
var server = app.listen(5000);
```

Note: Specify absolute path in express.static() by prepending __dirname. This will not break your application even if you run the express app from another directory.

In the above example, app.use() method mounts the middleware express.static for every request. The express.static middleware is responsible for serving the static assets of an Express.js application. The express.static() method specifies the folder from which to serve all static resources.

Now, run the above code using `node server.js` command and point your browser to *http://localhost:5000/myImage.jpg* and it will display myImage.jpg from the public folder (public folder should have myImage.jpg).

If you have different folders for different types of resources then you can set express.static middleware as shown below.

Example: Serve resources from different folders

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

```
app.use(express.static('public'));

//Serves all the request which includes /images in the url from Images folder
app.use('/images', express.static(__dirname + '/Images'));

var server = app.listen(5000);
```

In the above example, app.use() method mounts the express.static middleware for every request that starts with "/images". It will serve images from images folder for every HTTP requests that starts with "/images". For example, HTTP request http://localhost:5000/images/myImage.png will get myImage.png as a response. All other resources will be served from public folder.

Now, run the above code using `node server.js` and point your browser to *http://localhost:5000/images/myImage.jpg* and it will display myImage.jpg from the **images** folder, whereas *http://localhost:5000/myJSFile.js* request will be served from public folder. (images folder must include myImage.png and public folder must include myJSFile.js)

You can also create a virtual path in case you don't want to show actual folder name in the url.

Example: Setting virtual path

```
app.use('/resources',express.static(__dirname + '/images'));
```

So now, you can use *http://localhost:5000/resources/myImage.jpg* to serve all the images instead of *http://localhost:5000/images/myImage.jpg*.

In this way, you can use Express.js to server static resources such as images, CSS, JavaScript or other files.

### 3.6.3 Serve Static Resources using Node-static Module

In your node application, you can use node-static module to serve static resources. The node-static module is an HTTP static-file server module with built-in caching.

First of all, install node-static module using NPM as below.

```
npm install node-static
```

After installing node-static module, you can create static file server in Node.js which serves static files only.

The following example demonstrates serving static resources using node-static module.

Example: Serving static resources using node-static

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

var nStatic = require('node-static');

var fileServer = new nStatic.Server('./public');

http.createServer(function (req, res) {

    fileServer.serve(req, res);
```

```
}).listen(5000);
```

In the above example, node-static will serve static files from public folder by default. So, an URL request will automatically map to the file in the public folder and will send it as a response.

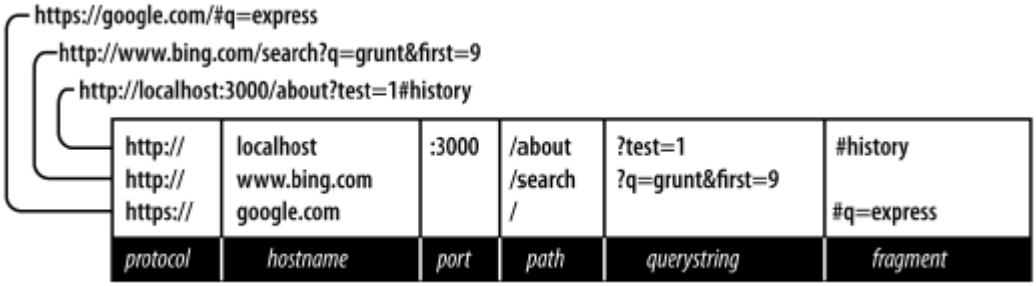
Now, run the above example using `node server.js` command and point your browser to *http://localhost:5000/myImage.jpg* (assuming that public folder includes myImage.jpg file) and it will display the image on your browser. You don't need to give "/public/myImage.jpg" because it will automatically serve all the static files from the public folder.

### 3.7 Response and Request objects

#### 3.7.1 Introduction

When you're building a web server with Express, most of what you'll be doing starts with a request object and ends with a response object. These two objects originate in Node and are extended by Express. Before we delve into what these objects offer us, let's establish a little background on how a client (a browser, usually) requests a page from a server, and how that page is returned.

The Parts of a URL




***Protocol***

The protocol determines how the request will be transmitted. We will be dealing

exclusively with http and https. Other common protocols include file and ftp.

***Host***

The host identifies the server. Servers on your computer (localhost) or a local network may simply be one word, or it may be a numeric IP address. On the Internet,the host will end in a top-level domain (TLD) like .com or .net. Additionally, there may be subdomains, which prefix the hostname. www is a very common subdomain,though it can be anything. Subdomains are optional.

***Port***

Each server has a collection of numbered ports. Some port numbers are "special," like 80 and 443. If you omit the port, port 80 is assumed for HTTP and 443 for HTTPS. In general, if you aren't using port 80 or 443, you should use a port number greater than 1023.1 It's very common to use easy-to-remember port numbers like 3000, 8080, and 8088.

***Path***

The path is generally the first part of the URL that your app cares about (it is possible to make decisions based on protocol, host, and port, but it's not good practice). The path should be used to uniquely identify pages or other resources in your app.

***Querystring***

The querystring is an optional collection of name/value pairs. The querystring

starts with a question mark (?), and name/value pairs are separated by ampersands (&). Both names and values should be URL encoded. JavaScript provides a built-in function to do that: encodeURIComponent. For example, spaces will be replaced with plus signs (+). Other special characters will be replaced with numeric character references.

***Fragment***

The fragment (or hash) is not passed to the server at all: it is strictly for use by the browser. It is becoming increasingly common for single-page applications or AJAXheavy applications to use the fragment to control the application. Originally, the fragment's sole purpose was to cause the browser to display a specific part of the document, marked by an anchor tag (<a id="chapter06">)

### 3.7.2 The Request Object

The request object (which is normally passed to a callback, meaning you can name it whatever you want: it is common to name it req or request) starts its life as an instance of http.IncomingMessage, a core Node object. Express adds additional functionality. Let's look at the most useful properties and methods of the request object (all of these methods are added by Express, except for req.headers and req.url, which originate in Node):

***req.params***

An array containing the named route parameters. We'll learn more about this in next lectures.

***req.param(name)***

Returns the named route parameter, or GET or POST parameters.

***req.query***

An object containing querystring parameters (sometimes called GET parameters) as name/value pairs. They are used to hold optional parameters

***req.body***

An object containing POST parameters. It is so named because POST parameters are passed in the body of the REQUEST, not in the URL like querystring parameters. To make req.body available, you'll need middleware that can parse the body content type, like body-parse as we saw previously.

***req.route***

Information about the currently matched route. Primarily useful for route debugging.

***req.cookies/req.signedCookies***

Objects containing containing cookie values passed from the client.

***req.headers***

The request headers received from the client.

***req.accepts([types])***

A convenience method to determine whether the client accepts a given type or types (optional types can be a single MIME type, such as application/json, a commadelimited list, or an array). This method is of primary interest to those writing public APIs; it is assumed that browsers will always accept HTML by default.

***req.ip***

The IP address of the client.

***req.path***

The request path (without protocol, host, port, or querystring).

***req.host***

A convenience method that returns the hostname reported by the client. This information can be spoofed and should not be used for security purposes. req.xhr A convenience property that returns true if the request originated from an AJAX call.

***req.protocol***

The protocol used in making this request (for our purposes, it will either be http or https).

***req.secure***

A convenience property that returns true if the connection is secure. Equivalent to req.protocol==='https'.

***req.url/req.originalUrl***

A bit of a misnomer, these properties return the path and querystring (they do not include protocol, host, or port). req.url can be rewritten for internal routing purposes, but req.originalUrl is designed to remain the original request and querystring.

***req.acceptedLanguages***

A convenience method that returns an array of the (human) languages the client prefers, in order. This information is parsed from the request header.

**3.7.3 The Request Object**

The response object (which is normally passed to a callback, meaning you can name it whatever you want: it is common to name it res, resp, or response) starts its life as an instance of http.ServerResponse, a core Node object. Express adds additional functionality. Let's look at the most useful properties and methods of the response object (all of these are added by Express):

***res.status(code)***

Sets the HTTP status code. Express defaults to 200 (OK), so you will have to use this method to return a status of 404 (Not Found) or 500 (Server Error), or any other status code you wish to use. For redirects (status codes 301, 302, 303, and 307), there is a method redirect, which is preferable.

***res.set(name, value)***

Sets a response header. This is not something you will normally be doing manually.

***res.cookie(name, value, [options]), res.clearCookie(name, [options])***

Sets or clears cookies that will be stored on the client. This requires some middleware support.

***res.redirect([status], url)***

Redirects the browser. The default redirect code is 302 (Found). In general, you should minimize redirection unless you are permanently moving a page, in which case you should use the code 301 (Moved Permanently).

***res.send(body), res.send(status, body)***

Sends a response to the client, with an optional status code. Express defaults to a content type of text/html, so if you want to change it to text/plain (for example), you'll have to call res.set('Content-Type', 'text/plain\') before calling res.send. If body is an object or an array, the response is sent as JSON instead (with the content type being set appropriately), though if you want to send JSON, I recommend doing so explicitly by calling res.json instead.

***res.json(json), res.json(status, json)***

Sends JSON to the client with an optional status code.

***res.jsonp(json), res.jsonp(status, json)***

Sends JSONP to the client with an optional status code.

***res.type(type)***

A convenience method to set the Content-Type header. Essentially equivalent to res.set('Content-Type', type), except that it will also attempt to map file extensions to an Internet media type if you provide a string without a slash in it. For example, res.type('txt') will result in a Content-Type of text/plain. There are areas where this functionality could be useful (for example, automatically serving disparate multimedia files), but in general, you should avoid it in favor of explicitly setting the correct Internet media type.

***res.format(object)***

This method allows you to send different content depending on the Accept request header. This is of primary use in APIs, and we will discuss this more in coming chapters. Here's a very simple example: res.format({'text/plain': 'hi there','text/html': 'hi there'}).

***res.attachment([filename]), res.download(path, [filename], [callback])***

Both of these methods set a response header called Content-Disposition to attachment; this will prompt the browser to download the content instead of displaying it in a browser. You may specify filename as a hint to the browser. With res.download, you can specify the file to download, whereas res.attachment just sets the header; you still have to send content to the client.

***res.sendFile(path, [options], [callback])***

This method will read a file specified by path and send its contents to the client. There should be little need for this method; it's easier to use the static middleware,and put files you want available to the client in the public directory. However, if you want to have a different resource served from the same URL depending on some condition, this method could come in handy.

***res.links(links)*** Sets the Links response header. This is a specialized header that has little use in most applications.

***res.locals, res.render(view, [locals], callback)***

res.locals is an object containing default context for rendering views. res.render will render a view using the configured templating engine.

EXAMPLES

1) Course management RESTful API

   Write All necessary CRUD APIs for course management application where each course has these fields: **id**(number), **name**(string), **authorId**(number) and **price**(number), the course depends on author object which has fields: **id**(number), **names**(string), email(string)

   Note: Implement below validation rules:

   - Course name must be string of minimum length 3 and always required for course to be either saved or updated
   - Author names must string of 3 minimum characters and always required.
   - Use Joi library for validation

1) Product shop management RESTful API

   Write back end CRUD APIs for shop management application where two objects are Products( has fields: id, name, description, categoryId) and categories(id, name)

   **Note:**

   - Expose at least below APIs:
   - All CRUD APIs for both objects
   - API to fetch products belong to same category
   - Validation rules
   - Course name must be string of minimum length 3 and always required for course to be either saved or updated
   - Author names must string of 3 minimum characters and always required.

- Use Joi library for validation

# CHAPTER FOUR: ADVANCED EXPRESS.JS

### 4.1 Middleware function

#### 4.1.0 Introduction

*Middleware* function is a function that has access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named next.

Express.js Middleware are also known as different types of functions that are invoked by the Express.js routing layer before the final request handler. As the name specified, Middleware appears in the middle between an initial request and final intended route. In stack, middleware functions are always invoked in the order in which they are added.

**Middleware** function can perform the following tasks.

Can execute any code.

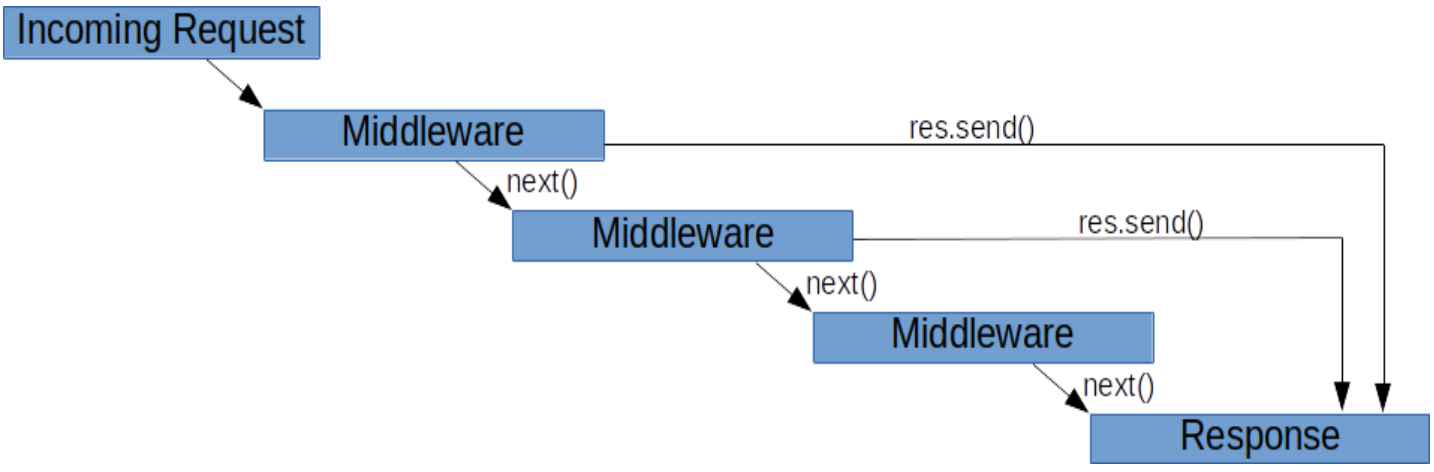
Can make changes to the request and the response objects.

Can end the request-response cycle.

Can call the next middleware function in the stack.

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging, and you will be stuck on the particular webpage and just loading and loading and so on.

When a request is received by Express, each middleware that matches the request is run in the order it is initialized until there is a terminating action (like a response being sent).



So if an error occurs, all middleware that is meant to handle errors will be called in order until one of them does not call the next() function call.

If you are building Node js Express Web Application, then you have already used an express middleware.

```
// For serving static files


app.use(express.static('public'));

// For parses incoming post request data

app.use(bodyParser.urlencoded({extended: true}));

app.use(bodyParser.json());
```

As we have discussed earlier, it has **next()** function is available. If I want to log anything for each incoming request, then I need to write following code.

```
const logger = function (req, res, next) {

  console.log('Middleware is used first time');

  next();

}
```

Similarly, there is one Node package called **Morgan** which is doing the same thing. Log all the request in the console. In actual Node js application, we can use the middleware like following.

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

var app = express()

const logger = function (req, res, next) {

  console.log('caught intrusion');

  next();

}

app.use(logger);

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

  res.send('Express middleware tutorial');

});

app.listen(3000);
```

**Authentication Example**

If we have to check whether the user has logged in or not for the current route or try to access authenticated route, then we can write the following tutorial.

```
router.get('/', isLoggedIn, function(req, res){
        res.render('pages/index');
});
function isLoggedIn(req, res, next){
    if(req.isAuthenticated()){
        next();
    }
    else{
        res.redirect("/users/login");
    }
}
```

In above example is also called "**Route Middleware."** It checks the request to see if the user is authenticated or not. Based on the function response, it will redirect the user accordingly.

**Configurable middleware**

If you need your middleware to be configurable, export a function which accepts an options object or other parameters, which, then returns the middleware implementation based on the input parameters.

```
// First.middleware.js
module.exports = function(options) {
  return function(req, res, next) {
    // Implement the middleware function based on the options object
    next();
}
```

Now, use this middleware in our **server.js** file or name it as you want.

```
// server.js
var Fm = require('./First.middleware.js');
app.use(Fm({ option1: 'App', option2: 'Dividend' }));
```

#### 4.1.1 **Types of Express Middleware**

There are mainly five types of middlewares.

##### 4.1.1.1 **Application-level middleware**

Bind application-level middleware to an instance of the app object by using the **app.use().**

This example shows a middleware function with no mount path. Every time user sends a request, the middleware function always runs.

```
var express = require('express')
var app = express()
const logger = function (req, res, next) {
  console.log('caught intrusion');
  next();
}
app.use(logger);
app.get('/', function (req, res) {
  res.send('Express middleware tutorial');
});
```

**Router-level middleware**

Router-level middleware bound to an instance of **express.router().**

```
const router = express.Router();
```

Load router-level middleware by using the **router.use()** function.

```
const app = express();
const router = express.Router();
router.use(function (req, res, next) {
console.log('Time:', Date.now());
  next();
});
```

**Error-handling middleware**

In order to call an error-handling middleware, you simply pass the error to next(), like this:

Note: Error-handling middleware always takes four arguments. You must provide four arguments to identify it as an error-handling middleware function. Even if you don't need to use the next object, you must specify it to maintain the signature. Otherwise, the next object will be interpreted as regular middleware and will fail to handle errors.

```
app.use((err, req, res, next) => {
  // middleware functionality here
})
```

In order to call an error-handling middleware, you simply pass the error to `next()`, like this:

```
app.get('/my-other-thing', (req, res, next) => {
  next(new Error('I am passing you an error!'));
});
app.use((err, req, res, next) => {
  console.log(err);
  if(!res.headersSent){
    res.status(500).send(err.message);
  }
});
```

In this case, the error handling middleware at the end of the pipeline will handle the error. You might also notice that I checked the `res.headersSent` property. This just checks to see if the response has already sent the headers to the client. If it hasn't it sends a 500 HTTP status and the error message to the client. You can also chain error-handling middleware. This is common to handle different types of errors in different ways. For instance:

```
app.get('/nonexistant', (req, res, next) => {
let err = new Error('I couldn\'t find it.');
err.httpStatusCode = 404;
next(err);
});
app.get('/problematic', (req, res, next) => {
let err = new Error('I\'m sorry, you can\'t do that, Dave.');
err.httpStatusCode = 304;
next(err);
```

```
});
// handles not found errors
app.use((err, req, res, next) => {
if (err.httpStatusCode === 404) {
res.status(400).send('NotFound');
}
next(err);
});
// handles unauthorized errors
app.use((err, req, res, next) => {
if(err.httpStatusCode === 304){
res.status(304).send('Unauthorized');
}
next(err);
})
// catch all
app.use((err, req, res, next) => {
console.log(err);
if (!res.headersSent) {
res.status(err.httpStatusCode || 500).send('UnknownError');
}
});
```

In this case, the middleware checks to see if a 404 (not found) error was thrown. If so, it renders the ‘NotFound’ template page and then passes the error to the next item in the middleware. The next middleware checks to see if a 304 (unauthorized) error was thrown. If it was, it renders the ‘Unauthorized’ page, and passes the error to the next middleware in the pipeline. Finally the “catch all” error handler just logs the error and if no response has been sent, it sends the error’s `httpStatusCode` (or an HTTP 500 status if none is provided) and renders the ‘UnknownError’ template.

**Built-in middleware**

Express has the following built-in middleware functions:

express.static serves static assets such as HTML files, images, and so on.

express.json parses incoming requests with JSON payloads.

express.urlencoded parses incoming requests with URL-encoded payloads.

**Third-party middleware**

Install the Node.js package for the required functionality, then load it in your application at the application level or the router level. We have already seen the body-parser middleware, which is third-party middleware.

The following example illustrates installing and loading the cookie-parsing middleware function cookie-parser.

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

```
var express = require('express')
var app = express()
var cookieParser = require('cookie-parser')

// load the cookie-parsing middleware
app.use(cookieParser())
```

### 4.2 Configuration

When building any kind of server side app, you will eventually have the following thoughts:

I would like to bind my app to a specific port.

I would like to enter a connection string so I can connect to my database.

I would like to use a third-party service through a provided API key or credentials.

Given values to these kinds of parameters will yield a *configuration*. It is a crucial part of the system as our application will have lots of configurable elements, so it must be handled properly throughout the codebase.

We shall use a third party library npm config:

- install the library --- npm install config

- create a folder inside your project and name it --- config

- create files .. default.json, production.json, development.json and custom-environment-variables.json then add your configs accordingly

4.3 Debugging

We use third party module debug to log messages depending on workspace settings. Eg

Lets set two workspaces app:startup and app:db

```
const startupDebug = require('debug')('app:startup')
const dbDebug = require('debug')('app:db')


startupDebug('Morgan enabled.....')
dbDebug('dbconnected.....')
```

then set a variable called DEBUG and give it either of the two workspaces or both

export DEBUG=app:startup …. this allows only startupDebug messages to be displayed

export DEBUG= app:startup,app:db --- will allow both  startupDebug and dbDebug messages to be displayed

export DEBUG=  ---This resets the workspaces

export DEBUG =app:* --- this enables all debug messages for any workspace starting with app: to be displayed

# CHAPTER FIVE: Callbacks, Promises, and Async

## 5.1 Synchronous vs Asynchronous

Synchronous operations in JavaScript entails having each step of an operation waits for the previous step to execute completely. This means no matter how long a previous process takes, subsquent process won't kick off until the former is completed. Asynchronous operations, on the other hand, defers operations. Any process that takes a lot of time to process is usually run alongside other synchronous operation and completes in the future.

This lesson dwells on fundamental concepts that JavaScript relies on to handle asynchronous operations. These concepts include: **Callback** functions, **Promises** and the use of **Async** and **Await** to handle deferred operations in JavaScript.

### Asynchronous Operations

Operations in JavaScript are traditionally synchronous and execute from top to bottom. For instance, a farming operation that logs farming process to the console:

```
console.log("Plant corn");
console.log("Water plant");
console.log("Add fertilizer");
```

If we run the code above, we have the following logged in the console:

```
Plant corn
Water plant
Add fertilizer
```

Now let's tweak that a bit so that watering the farm take longer than planting and fertilizing:

```
console.log("Plant maize");

setTimeout(function() {
  console.log("Water plant")
},3000);
```

```
console.log("Add fertilizer");
```

We get the following in the console:

```
  Plant Maize
  Add fertilizer
  Water plant
```

Why? The setTimeout function makes the operation asynchronous by deferring plant watering to occur after 3 seconds. The whole operation doesn't pause for 3 seconds so it can log "Water plant". Rather, the system goes ahead to apply fertilizers and then water plant after 3 seconds.

### 5.2 Callback Functions

When a function simply accepts another function as an argument, this contained function is known as a callback function. Using callback functions is a core functional programming concept, and you can find them in most JavaScript code; either in simple functions like setInterval, event listening or when making API calls.

Callback functions are syntaxed as:

```
setInterval(function() {
  console.log('hello!');
}, 1000);
```

setInterval accepts a callback function as its first parameter and also a time interval. Another example using .map();

```
const list    = ['man', 'woman', 'child']

// create a new array
// loop over the array and map the data to new content
const newList = list.map(function(val) {
  return val + " kind";
});
```

```
// newList = ['man kind', 'woman kind', 'child kind']
```

In the example above, we used the `.map()` method to iterate through the array `list`, the method accepts a callback function which states how each element of the array will be manipulated. Callback functions can also accept arguments as well.

Multiple functions can be created independently and used as callback functions. This is called multi-level functions. When this function tree created becomes too large, the code becomes incomprehensible sometimes and is not easily refactored. This is known as **callback hell.** Let's see an example:

```
// a bunch of functions are defined up here

// lets use our functions in callback hell
function setInfo(name) {
  address(myAddress) {
    officeAddress(myOfficeAddress) {
      telephoneNumber(myTelephoneNumber) {
        nextOfKin(myNextOfKin) {
          console.log('done'); //let's begin to close each function!
        };
      };
    };
  };
}
```

We are assuming these functions have been previously defined elsewhere. You can see how confusing it is to pass each function as callbacks. This scenario where callbacks form a a chrismas tree like structure is called **callback hell**. Callback functions are useful for short asynchronous operations. When working with large sets, this is not considered best practice. Because of this challenge, **Promises** were introduced to simplify deferred activities.

### 5.3 Promises

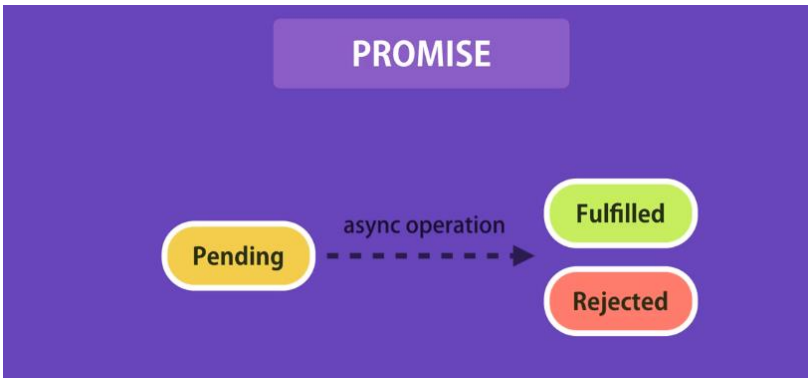
I **promise** to do this **whenever** that is true. If it isn't true, then I won't.

This is a simple illustration of JavaScript Promises. Sounds like an IF statement? We'll soon see a huge difference.

A promise is used to handle the asynchronous result of an operation. JavaScript is designed to not wait for an asynchrnous block of code to completely execute before other synchronous parts of the code can run. For instance, when making API requests to servers, we have no idea if these servers are offline or online, or how long it takes to process the server request.

With Promises, we can defer execution of a code block until an async request is completed. This way, other operations can keep running without interruption.

Promises have three states:




- **Pending**: This is the initial state of the Promise before an operation begins
- **Fulfilled**: This means the specified operation was completed
- **Rejected**: The operation did not complete; an error value is usually thrown

**Creating a Promise**

The Promise object is created using the *new* keyword and contains the promise; this is an executor function which has a **resolve** and a **reject** callback. As the names imply, each of these callbacks returns a value with the **reject** callback returning an error object.

```
const promise = new Promise(function(resolve, reject) {
```

```
  // promise description
})
```

Let’s create a promise:

```
const weather = true
const date    = new Promise(function(resolve, reject) {
  if (weather) {
    const dateDetails = {
      name:     'Cubana Restaurant',
      location: '55th Street',
      table:    5
    };

    resolve(dateDetails)
  } else {
    reject(new Error('Bad weather, so no Date'))
  }
});
```

If `weather` is true, resolve the promise returning the data `dateDetails`, else return an error object with data `Bad weather, so no Date`.

**Using Promises**

Using a promise that has been created is relatively straightforward; we chain `.then()` and `.catch()` to our Promise like so:

```
date
  .then(function(done) {
    // the content from the resolve() is here
  })
  .catch(function(error) {
    // the info from the reject() is here
  });
```

Using the promise we created above, let's take this a step further:

```
const myDate = function() {
  date
    .then(function(done) {
      console.log('We are going on a date!')
      console.log(done)
    })
    .catch(function(error) {
        console.log(error.message)
    })
}

myDate();
```

Since the weather value is true, we call `mydate()` and our console logs read:

```
We are going on a date!
{
  name: 'Cubana Restaurant',
  location: '55th Street'
  table: 5
}
```

`.then()` receives a function with an argument which is the resolve value of our promise. `.catch` returns the reject value of our promise.

Note: Promises are asynchronous. Promises in functions are placed in a micro-task queue and run when other synchronous operations complete.

**Chaining Promises**

Sometimes we may need to execute two or more asynchronous operations based on the result of preceding promises. In this case, promises are chained. Still using our created promise, let’s order an uber if we are going on a date.

So we create another promise:

```
const orderUber = function(dateDetails) {
  return new Promise(function(resolve, reject) {
    const message = `Get me an Uber ASAP to ${dateDetails.location}, we are going on a date!`;

    resolve(message)
  });
}
```

This promise can be shortened to:

```
const orderUber = function(dateDetails) {
  const message = `Get me an Uber ASAP to ${dateDetails.location}, we are going on a date!`;
  return Promise.resolve(message)
}
```

We chain this promise to our earlier date operation like so:

```
const myDate = function() {
  date
    .then(orderUber)
    .then(function(done) {
      console.log(done);
    })
    .catch(function(error) {
      console.log(error.message)
    })
}

myDate();
```

Since our weather is true, the output to our console is:

```
Get me an Uber ASAP to 55th Street, we are going on a date!
```

Once the orderUber promise is chained with .then, subsequent .then utilizes data from the previous one.

## 5.4 Async and Await

An async function is a modification to the syntax used in writing promises. You can call it syntactic sugar over promises. It only makes writing promises easier.

An async function returns a promise -- if the function returns a value, the promise will be resolved with the value, but if the async function throws an error, the promise is rejected with that value. Let's see an async function:

```
async function myRide() {
  return '2017 Dodge Charger';
}
```

and a different function that does the same thing but in promise format:

```
function yourRide() {
  return Promise.resolve('2017 Dodge Charger');
}
```

From the above statements, `myRide()` and `yourRide()` are *equal* and will both resolve to `2017 Dodge Charger`. Also when a promise is rejected, an async function is represented like this:

```
function foo() {
  return Promise.reject(25)
}

// is equal to
async function() {
  throw 25;
}
```

## 5.5 Await

Await is only used with an async function. The await keyword is used in an async function to ensure that all promises returned in the async function are synchronized, ie. they wait for each other. Await eliminates the use of callbacks in `.then()` and `.catch()`. In using async and await, async is prepended when returning a promise,

await is prepended when calling a promise. `try` and `catch` are also used to get the rejection value of an async function. Let's see this with our date example:

```
async function myDate() {
  try {

    let dateDetails = await date;
    let message     = await orderUber(dateDetails);
    console.log(message);

  } catch(error) {
    console.log(error.message);
  }
}
```

Lastly we call our async function:

```
myDate();
```