

NodeJS



Authorized & published by Summitworks Technologies Inc

Agenda

- Web Server
- Introduction to Node JS
- Setup Node.js Development Environment
- Node.js Console
- Node.js Module
- Node Package Manager
- package.json
- Node.js Web Server
- Handle HTTP Request
- Blocking Function
- Non-blocking Function
- Node.js Callback Function
- Callback Hell
- How to fix Callback Hell

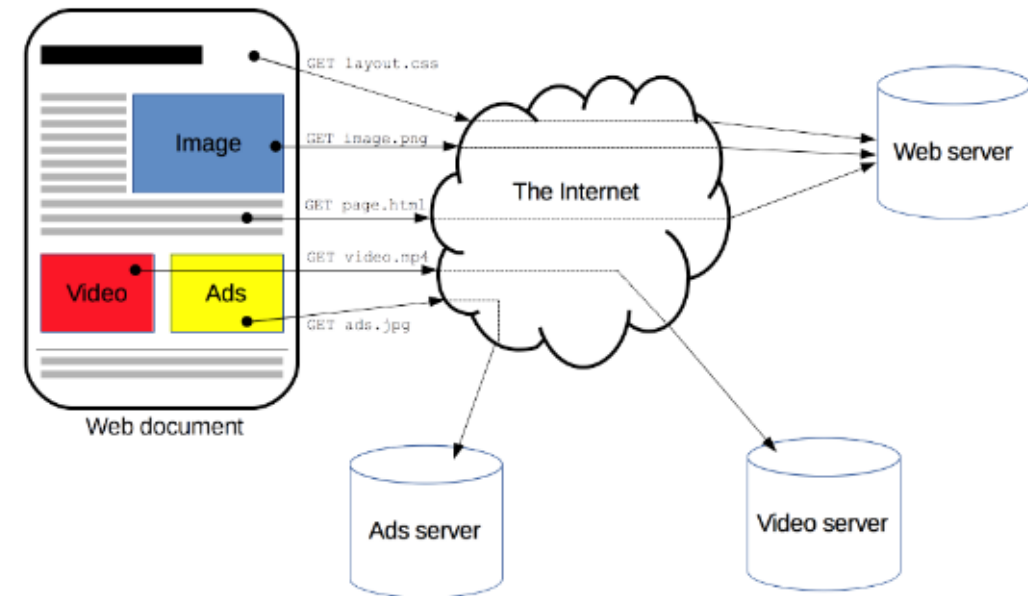
Web Server

What is a Web Server?

- A **web server** is a system that delivers content or services to end users over the internet. A web server consists of physical server, server operating system (OS) and software used to facilitate HTTP communication.
- A server is not necessarily a single machine, but several servers can be hosted on the same machine.

What is a Client?

- A **client** can be a simple application or a whole system that accesses services being provided by a server.
- A client can connect to a server through different means like Internet protocols (HTTP).
- Clients and servers communicate by exchanging individual messages.
- The messages sent by the client, are called **requests** and the messages sent by the server as an answer are called **responses**.



Introduction to NodeJS

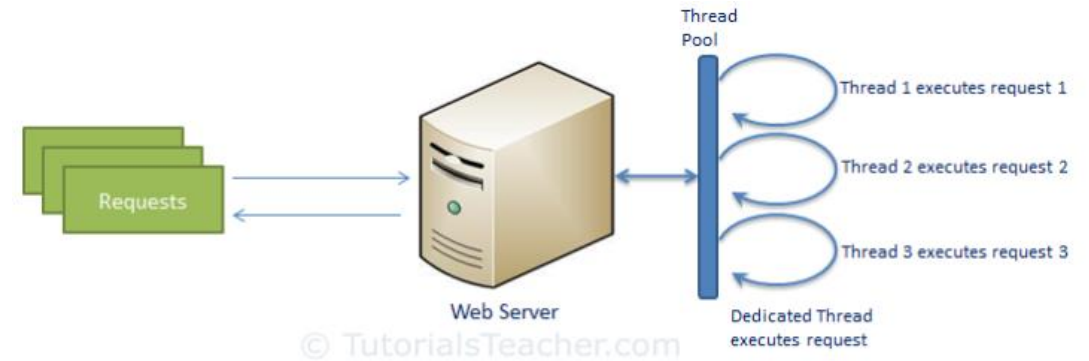
- ***Node.js*** is a server-side platform built on ***Google Chrome's JavaScript Engine (V8 Engine)*** for easily building fast and scalable network applications.
- ***Node.js*** uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.



NodeJS Process Model

Traditional Web Server Model

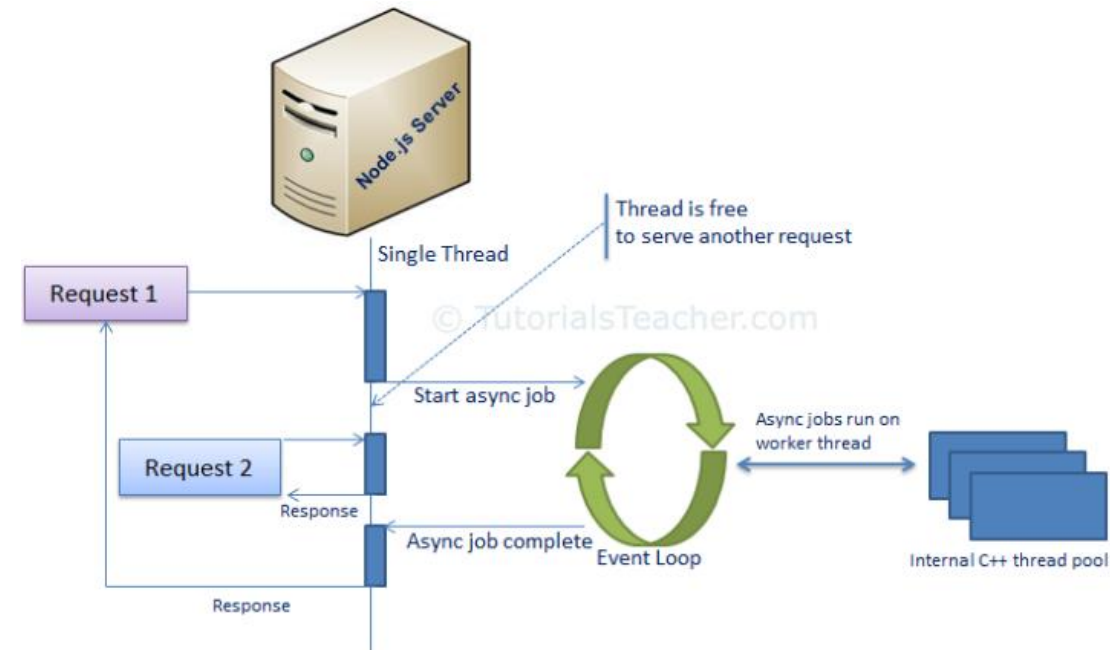
- In the traditional web server model, each request is handled by a dedicated thread from the thread pool.
- If no thread is available in the thread pool at any point of time then the request waits till the next available thread.
- Dedicated thread executes a particular request and does not return to thread pool until it completes the execution and returns a response.



NodeJS Process Model cont.

Node.js Process 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.



Features of NodeJS

- **Extremely fast:** Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
- **I/O is Asynchronous and Event Driven:** All APIs of Node.js library are asynchronous i.e. non-blocking. So a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call. It is also a reason that it is very fast.
- **Single threaded:** Node.js follows a single threaded model with event looping.
- **Highly Scalable:** Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.
- **Open source:** Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications.
- **License:** Node.js is released under the MIT license.

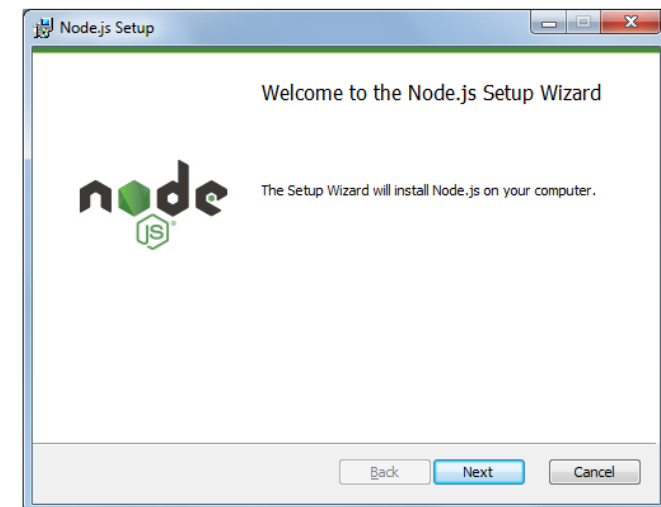
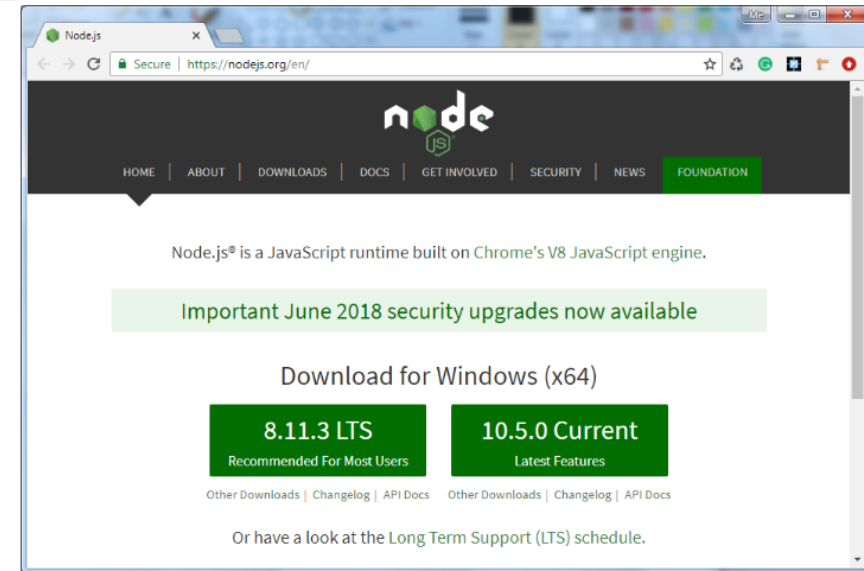
Setup Node.js Development Environment

- Let's 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.
 - Node.js
 - Node Package Manager (NPM)
 - IDE (Integrated Development Environment) or TextEditor
- NPM (Node Package Manager) is included in Node.js installation since Node version 0.6.0., so there is no need to install it separately.

Setup Node.js Development Environment cont.

Install Node.js on Windows

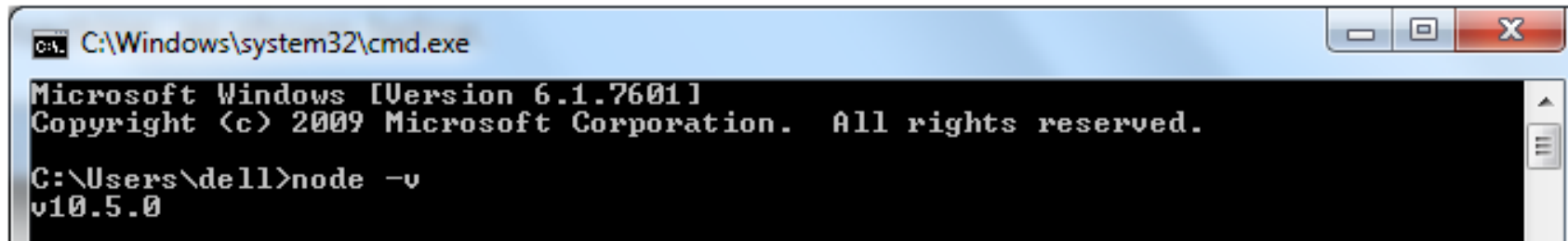
- Visit Node.js official web site <https://nodejs.org/>.
- It will automatically detect OS and display download link as per your Operating System.
- For example, it will display following download link for 64 bit Windows OS.
- Download node MSI for windows by clicking on 8.11.3 LTS or 10.5.0 Current button.
- After you download the MSI, double-click on it to start the installation as shown here.
- 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.



Setup Node.js Development Environment cont.

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 here.



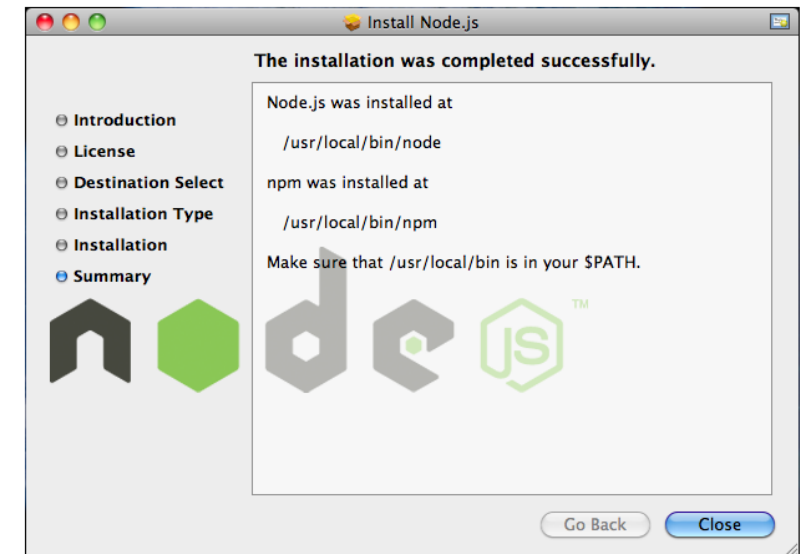
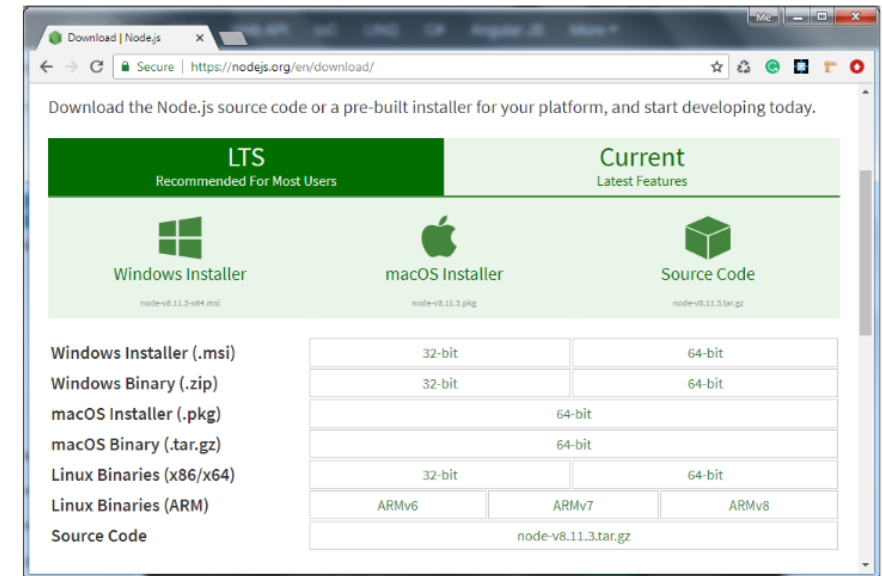
```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\dell>node -v
v10.5.0
```

Setup Node.js Development Environment cont.

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.



Setup Node.js Development Environment cont.

Install Node.js on Mac/Linux

- 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
```

Setup Node.js Development Environment cont.

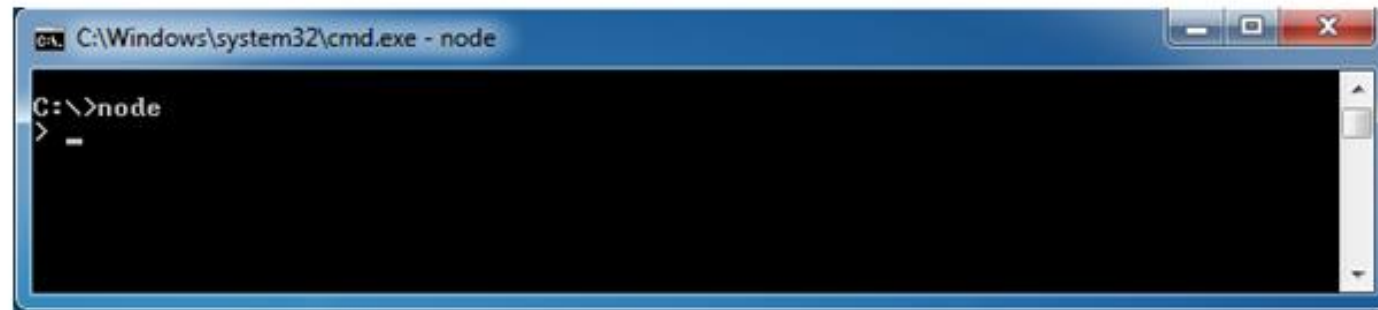
IDE

- 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, Atom 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 in the image.
- It will change the prompt to > in Windows and MAC.



Node.js Console – REPL cont.

- You can now test pretty much any Node.js/JavaScript expression in REPL.
- For example, if you write "10 + 20" then it will display result 30 immediately in new line.
- You can even concatenate strings using + operator as in browser's JavaScript.
- 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.
- Now, you can execute node-example.js from command prompt as shown here.

```
> 10 + 20  
30
```

```
node-example.js  
  
console.log("Hello World");
```



```
C:\Windows\system32\cmd.exe  
C:\>node node-example.js  
Hello World  
C:\>
```


Node.js Console – REPL cont.

- 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.
.clear	Exit from multiline expression.

Node.js Module

- Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.
- It can be a function, can be a class, can be an object or even simple variables.
- Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope.
- Each module can be placed in a separate .js file under a separate folder.
- Node.js includes three types of modules:
 - Core Modules
 - Local Modules
 - Third Party Modules

Node.js Core Modules

- 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 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 the 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.
- Require function is globally available.

Loading Core Modules cont.

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

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

const port = 5000;

const server = http.createServer((req, res) => { // 2 - creating server
  //handle incoming requests here..
});

server.listen(port, () => { //3 - listen for any incoming requests
  console.log(`Server running at port ` + port);
});
```

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.
- In Node.js, module should be placed in a separate JavaScript file.
- Let's write simple logging module which logs the information, warning or error to the console.

Node.js Local Module cont.

- In the example of logging module, we have created an object with a function - sayHello().
- At the end, we have assigned this object to ***module.exports***. The ***module.exports*** in the example exposes a Cat 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 Cat module in our application.

```
cat.js  x
1  var Cat = {
2      legs: 4,
3      head: 1,
4      ears: 2,
5      sayHello: function(){
6          console.log("meow");
7      }
8  };
9
10 module.exports = Cat;
11
```


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 and specify the path of JavaScript file of the module.
- The example demonstrates how to use the Cat module contained in `cat.js`.
- First, it loads the Cat module using `require()` function and specified path (here the path `'./cat'`) where Cat module is stored.
- Notice that we use `./` to locate the module, that means that the module is located in the same folder as the Node.js file.
- The `require()` function returns a `cat` object because Cat module exposes an object in `cat.js` using `module.exports`. So now you can use Cat module as an object and call any of its function using dot notation e.g. `cat.sayHello()`
- Run the above example using command prompt (in Windows) as shown below.

```
var cat = require('./cat');  
  
console.log(cat.legs);  
cat.sayHello();
```

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

Export Module in Node.js

- In the previous section, you learned how to write a local module using `module.exports`.
- Now let's see 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 example exposes simple string message as a module in Message.js.
- Now, import this message module and use it as shown here.
- Run the above example and see the result as shown here.
- 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.

Message.js

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

app.js

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

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

Export Object

- *exports* is an object. So, you can attach properties or methods to it.
- The example exposes an object with a string property in Message.js file.
- In the example, we have attached a property "SimpleMessage" to the exports object. Now, import and use this module as shown below.
- In the 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.

Message.js

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

app.js

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

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

Export Function

- You can attach an anonymous function to exports object as shown here.
- Now, you can use the above module as below.
- 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
```

Log.js

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

app.js

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

Node Package Manager

- Node Package Manager (NPM) is a command line tool that installs, updates or uninstalls Node.js packages in your application.
- It is also an online repository for open-source Node.js packages.
- The node community around the world creates useful modules and publishes them as packages in this repository.
- Official website: <https://www.npmjs.com>
- NPM is included with Node.js installation. After you install Node.js, verify NPM installation by writing the command in terminal or command prompt as shown here.



```
C:\> npm -v  
2.11.3
```

Node Package Manager cont.

- If you have an older version of NPM then you can update it to the latest version using the following command.
- To access NPM help, write **npm help** in the command prompt or terminal window.
- NPM performs the operation in two modes: **global and local**.
- In the global mode, NPM performs operations which affect all the Node.js applications on the computer whereas in the local mode, NPM performs operations for the particular local directory which affects an application in that directory only.

Install Package Locally

- Use the following command to install any third party module in your local Node.js project folder.

```
C:\> npm install npm -g
```

```
C:\> npm help
```

```
C:\> npm install <package name>
```


Node Package Manager cont.

- For example, the following command will install ExpressJS into MyNodeProj folder.

```
C:\MyNodeProj> npm install express
```

- All the modules installed using NPM are installed under **node_modules** folder.
- The above command will create ExpressJS folder under node_modules folder in the root folder of your project and install Express.js there.

Package.json

- The best way to manage locally installed npm packages is to create a package.json file.
- A package.json file:
 - lists the packages that your project depends on.
 - allows you to specify the versions of a package that your project can use using semantic versioning rules.
 - makes your build reproducible, and therefore much easier to share with other developers.



Package.json cont.

Requirements

- A package.json must have:
- "name"
 - all lowercase
 - one word, no spaces
 - hyphens and underscores allowed
- "version"
 - in the form of x.x.x
- "description" : info from the readme, or an empty string "" (optional)
- "main" : always index.js
- "scripts" : by default creates an empty test script
- "keywords" : empty
- "author" : empty
- "license" : ISC

Package.json cont.

Creating a package.json

- There are two basic ways to create a package.json file.
1. Run a CLI questionnaire
 - To create a package.json with values that you supply, run:
 - This will initiate a command line questionnaire that will conclude with the creation of a package.json in the directory in which you initiated the command
 2. Create a default package.json
 - To get a default package.json, run npm init with the --yes or -y flag:
 - This method will generate a default package.json using information extracted from the current directory.

```
> npm init
```

```
> npm init --yes
```

Package.json cont.

Specifying Dependencies

- To specify the packages your project depends on, you need to list the packages you'd like to use in your package.json file.
- There are 2 types of packages you can list
 - "**dependencies**": These packages are required by your application in production.
 - "**devDependencies**": These packages are only needed for development and testing.
- You can add dependencies to a package.json file from the command line or by manually editing the package.json file.
- For example, the project below uses any version of the package my_dep that matches major version 1 in production and requires any version of the package my_test_framework that matches major version 3, but only for development:

```
{
  "name": "my_package",
  "version": "1.0.0",
  "dependencies": {
    "my_dep": "^1.0.0"
  },
  "devDependencies": {
    "my_test_framework": "^3.1.0"
  }
}
```

Package.json cont.

Adding dependencies to a package.json file from the command line

- To add dependencies and devDependencies to a package.json file from the command line, you can install them in the root directory of your package using the **--save-prod** flag for dependencies or the **--save-dev** flag for devDependencies.
- To add an entry to the "dependencies" attribute of a package.json file, on the command line, run the following command:

```
npm install <package-name> [--save-prod]
```

- To add an entry to the "devDependencies" attribute of a package.json file, on the command line, run the following command:

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

Package.json cont.

Install Package Globally

- NPM can also install packages globally so that all the node.js application on that computer can import and use the installed packages.
- NPM installs global packages into /<User>/local/lib/node_modules folder.
- Apply -g in the install command to install package globally.
- For example, the following command will install ExpressJS globally.

```
C:\MyNodeProj> npm install -g express
```


Package.json cont.

Update Package

- To update the package installed locally in your Node.js project, navigate the command prompt or terminal window path to the project folder and write the following update command.

```
C:\MyNodeProj> npm update <package name>
```

- The following command will update the existing ExpressJS module to the latest version.

```
C:\MyNodeProj> npm update express
```

Package.json cont.

Uninstall Packages

- Use the following command to remove a local package from your project.

```
C:\>npm uninstall <package name>
```

- The following command will uninstall ExpressJS from the application.

```
C:\MyNodeProj> npm uninstall express
```

Node.js Web Server

- 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 (Like 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 example shown here is simple Node.js web server contained in server.js file.

```
//Create Node.js Web Server

const http = require('http'); // 1 - Import Node.js core module
const port = 5000;

const server = http.createServer((req, res) => { // 2 - creating server
  //handle incoming requests here..
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
  console.log(req.url)
});

server.listen(port, () => { //3 - listen for any incoming requests
  console.log(`Server running at port ` + port);
});
```

Create Node.js Web Server cont.

- In the above example, we import the *http module* using *require()* function.
- The http module is a core module of Node.js, so don't 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.

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

Create Node.js Web Server cont.

Request Object

- The function that's passed in to `createServer` is called once for every HTTP request that's made against that server, so it's called the request handler.
- `req` is an object containing information about the HTTP request that raised the event.
- The `req` parameter contains the method, url and header information of the request made
 - Method - HTTP verbs (GET,PUT, POST,DELETE)
 - Url- The url is the full URL without the server, protocol or port.
 - Header- HTTP message headers are represented by an object like this

```
{ 'content-length': '123',  
  'content-type': 'text/plain',  
  'connection': 'keep-alive',  
  'host': 'mysite.com',  
  'accept': '*/*' }
```

which contain info like what kind of browser made the request, what sort of responses it can handle, whether or not it's able to understand HTTP compression, etc.
- Body- The data is passed in the body during an HTTP Put/Post request

Create Node.js Web Server cont.

Response Object

- The `res` parameter in the `CreateServer` function is the response sent back to the client.
- `status code` - Contains the status code of the response (e.g., 200 for a Success, 404 for Not Found, 401 for Unauthorized Error).
- `setHeader` - Is used to set the different attributes of Response Header, like Content-Type, Content-Length.
- `writeHead` - Sends a response header to the request. The status code is a 3-digit HTTP status code, like 404. The last argument, `headers`, are the response headers.
- `write` - This sends a chunk of the response body. This method may be called multiple times to provide successive parts of the body.
- `end` - This method signals to the server that all of the response headers and body have been sent; that server should consider this message complete. If data is specified, it is equivalent to calling `response.write()`.

Blocking Function

- In contrast to asynchronous function, a synchronous function blocks the execution until the task on the resource is completed. Therefore synchronous function is also called as blocking function.
- The first console log is waiting to get a result from the API call and hence it is blocked for that operation.
- As soon as data1 is received, it is printed out using console.log.
- Next the second API call is triggered and the cycle repeats.
- Finally the evaluation of the sum will not start until the completion of second console.log statement, however the sum is totally unrelated to the above 2 API calls and hence hindering the execution of sum.
- The output as printed on the screen will be:

```
1 // assume getData to be an API call
2 console.log("Getting Data1");
3 var data1 = getData('123');
4 console.log("Data is:", data1);
5
6 console.log("Getting Data2");
7 var data2 = getData('456');
8 console.log("Data is:", data2);
9
10 var sum = 1 + 2;
11 console.log("sum is:", sum);
```

Getting Data1

Data is: {"id": 123, "name": "sarwar"}

Getting Data2

Data is: {"id": 456, "name": "nirmala"}

sum is:3

Non-Blocking Function

- Let's see how to write the same code but in a non-blocking form with the use of ***non-blocking or callback functions***.
- A callback is a function, passed as an argument to an asynchronous function, that describes what to do after the asynchronous operation has completed.
- The first API call initiates, but since console.log is in callback function, it won't wait for the completion of the call, and the execution shifts to second API call immediately after that.
- The second API call initiates in a similar manner, and the execution shifts to printing the sum without waiting for the result from the call.
- Sum is printed and as soon as we get results from the 2 API calls they are printed on the console respectively.
- The output as printed on the screen will be:

```
1 console.log("Getting Data1");
2 getData('123', function(data1) {
3     console.log("Data is:", data1);
4 });
5
6 console.log("Getting Data2");
7 getData('456', function(data1) {
8     console.log("Data is:", data1);
9 });
10
11 var sum = 1 + 2;
12 console.log("sum is:", sum);
```

Getting Data1

Getting Data2

sum is:3

Data is: {"id": 123, "name": "sarwar"}

Data is: {"id": 456, "name": "nirmala"}

Node Callback Function

- Asynchronous calls are one of the fundamental factor for Node.js to have become popular.
- Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All the APIs of Node are written in such a way that they support callbacks.
- The basic idea of callback function is that, if we have to do something which could take a long time, let's say we're trying to read a large file, we don't want our node.js server waiting around for the file to be read when it could be dealing with other incoming requests. So to deal with this we tell Node.js to do what it has to do in the background and to call a function when it's finished. This way Node.js can carry on dealing with other requests while it's reading the file, making our code non-blocking.

Callback Hell

- To solve this blocking problem, JavaScript heavily relies on callbacks, which are functions that run after a long-running process (IO, timer, etc.) has finished, thus allowing the code execution to proceed past the long-running task.
- While the concept of callbacks is great in theory, it can lead to some really confusing and difficult-to-read code. Just imagine if you need to make callback after callback:
- As you can see, this can really get out of hand. Throw in some if statements, for loops, function calls, or comments and you'll have some very hard-to-read code. Beginners especially fall victim to this, not understanding how to avoid this "pyramid of doom".

```
getData(function(a){  
    getData(a, function(b){  
        getData(b, function(c){  
            getData(c, function(d){  
                getData(d, function(e){  
                    ...  
                });  
            });  
        });  
    });  
});
```

How to fix Callback Hell

Modularize

- Create reusable functions and place them in a module to reduce the cognitive load required to understand your code. Splitting your code into small pieces like this also helps you handle errors, write tests, forces you to create a stable and documented public API for your code, and helps with refactoring.
- In the example here we have a new file called formuploader.js that contains two functions:
- The module.exports bit is an example of the node.js module system which works in node.
- Now that we have formuploader.js, we just need to require it and use it! Here is how our application specific code looks now:

```
module.exports.submit = formSubmit

function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse)
}

function postResponse (err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err
  statusMessage.value = body
}
```

```
var formUploader = require('formuploader')
document.querySelector('form').onsubmit = formUploader.submit
```

How to fix Callback Hell cont.

Give your functions names

- As you can see naming functions is super easy and has some immediate benefits:
 - makes code easier to read.
 - when exceptions happen you will get stack traces that reference actual function names instead of "anonymous".
 - allows you to move the functions and reference them by their names.
- Adding a little extra information (names) to the functions can make a big difference for readability, especially when you're multiple levels deep in callbacks.
- Now here the first function appends some text while the second function notifies the user of the change.

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

var myFile = '/tmp/test';
fs.readFile(myFile, 'utf8', function appendText(err, txt) {
  if (err) return console.log(err);

  txt = txt + '\nAppended something!';
  fs.writeFile(myFile, txt, function notifyUser(err) {
    if(err) return console.log(err);
    console.log('Appended text!');
  });
});
```