What is a Module in Node.js?

Consider modules to be the same as JavaScript libraries.

built-in modules of Node.js version 6.10.3:

|  |  |
| --- | --- |
| **Module** | **Description** |
| Assert | Provides a set of assertion tests |
| Buffer | To handle binary data |
| child\_process | To run a child process |
| cluster | To split a single Node process into multiple processes |
| Crypto | To handle OpenSSL cryptographic functions |
| Dgram | Provides implementation of UDP datagram sockets |
| Dns | To do DNS lookups and name resolution functions |
| domain | Deprecated. To handle unhandled errors |
| Events | To handle events |
| Fs | To handle the file system |
| http | To make Node.js act as an HTTP server |
| https | To make Node.js act as an HTTPS server. |
| Net | To create servers and clients |
| Os | Provides information about the operation system |
| Path | To handle file paths |
| punycode | Deprecated. A character encoding scheme |
| querystring | To handle URL query strings |
| readline | To handle readable streams one line at the time |
| Stream | To handle streaming data |
| string\_decoder | To decode buffer objects into strings |
| Timers | To execute a function after a given number of milliseconds |
| Tls | To implement TLS and SSL protocols |
| Tty | Provides classes used by a text terminal |
| url | To parse URL strings |
| Util | To access utility functions |
| v8 | To access information about V8 (the JavaScript engine) |
| Vm | To compile JavaScript code in a virtual machine |
| Zlib | To compress or decompress files |

# Node.js Assert Module

var assert = require('assert');  
assert(5 > 7);

var assert = require('assert');  
assert(50 > 70, "My message goes here");

var assert = require('assert');

assert(50 > 70);

Definition and Usage

The assert module provides a way of testing expressions. If the expression evaluates to 0, or false, an assertion failure is being caused, and the program is terminated.

This module was built to be used internally by Node.js.

Syntax

The syntax for including the assert module in your application:

var assert = require('assert');

Assert Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| assert() | Checks if a value is true. Same as assert.ok() |
| deepEqual() | Checks if two values are equal |
| deepStrictEqual() | Checks if two values are equal, using the strict equal operator (===) |
| doesNotThrow() |  |
| equal() | Checks if two values are equal, using the equal operator (==) |
| fail() | Throws an Assertion Error |
| ifError() | Throws a specified error if the specified error evaluates to true |
| notDeepEqual() | Checks if two values are not equal |
| notDeepStrictEqual() | Checks if two values are not equal, using the strict not equal operator (!==) |
| notEqual() | Checks if two values are not equal, using the not equal operator (!=) |
| notStrictEqual() | Checks if two values are not equal, using the strict not equal operator (!==) |
| ok() | Checks if a value is true |
| strictEqual() | Checks if two values are equal, using the strict equal operator (===) |
| throws() |  |

var assert = require('assert');  
var x = { a : { n: 0 } };  
var y = { a : { n: 0 } };  
var z = { a : { n: 1 } };  
assert.deepEqual(x, y); //OK  
assert.deepEqual(x, z); /\*AssertionError: { a: { n: 0 } } deepEqual {a: { n: 1 } }\*/

var assert = require('assert');  
var x = { a : { n: 0 } };  
var y = { a : { n: 0 } };  
var z = { a : { n: '0' } };  
assert.deepStrictEqual(x, y); //OK  
assert.deepStrictEqual(x, z); /\*AssertionError: { a: { n: 0 } } deepStrictEqual {a: { n: '0' } }\*/

## What is Callback?

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

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

## Blocking Code Example

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

Notess Section is giving self learning content

to teach the world in simple and easy way!!!!!

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

var fs = require("fs");

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

console.log(data.toString());

console.log("Program Ended");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Notess Section is giving self learning content

to teach the world in simple and easy way!!!!!

Program Ended

## Non-Blocking Code Example

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

Notess Section is giving self learning content

to teach the world in simple and easy way!!!!!

Update main.js to have the following code −

var fs = require("fs");

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

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

console.log(data.toString());

});

console.log("Program Ended");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

Program Ended

Notess Section is giving self learning content

to teach the world in simple and easy way!!!!!

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

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

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

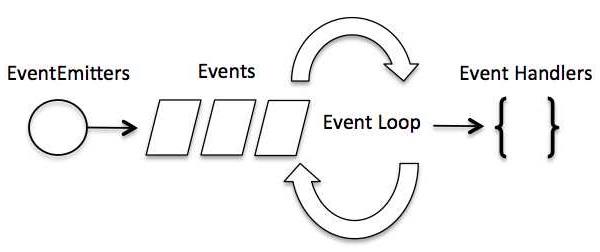
# Node.js - Event Loop

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

## Event-Driven Programming

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

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



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

// Import events module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

Following is the syntax to bind an event handler with an event −

// Bind event and event handler as follows

eventEmitter.on('eventName', eventHandler);

We can fire an event programmatically as follows −

// Fire an event

eventEmitter.emit('eventName');

## Example

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

 Live Demo

// Import events module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

// Create an event handler as follows

var connectHandler = function connected() {

console.log('connection succesful.');

// Fire the data\_received event

eventEmitter.emit('data\_received');

}

// Bind the connection event with the handler

eventEmitter.on('connection', connectHandler);

// Bind the data\_received event with the anonymous function

eventEmitter.on('data\_received', function() {

console.log('data received succesfully.');

});

// Fire the connection event

eventEmitter.emit('connection');

console.log("Program Ended.");

Now let's try to run the above program and check its output −

$ node main.js

IT should produce the following result −

connection successful.

data received successfully.

Program Ended.

## How Node Applications Work?

In Node Application, any async function accepts a callback as the last parameter and a callback function accepts an error as the first parameter. Let's revisit the previous example again. Create a text file named input.txt with the following content.

Notess Section is giving self learning content

to teach the world in simple and easy way!!!!!

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

var fs = require("fs");

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

if (err) {

console.log(err.stack);

return;

}

console.log(data.toString());

});

console.log("Program Ended");

Here fs.readFile() is a async function whose purpose is to read a file. If an error occurs during the read operation, then the **err object** will contain the corresponding error, else data will contain the contents of the file. **readFile**passes err and data to the callback function after the read operation is complete, which finally prints the content.

Program Ended

Notess Section is giving self learning content

to teach the world in simple and easy way!!!!!