**What is Synchronous Programming?**

In most traditional programming practice , most I/O operation occurs synchronously. That generally means that your flow of program will wait until the I/O operation is completed. If we take up an austere situation where you need to read a big file, write that in your database and then send http request to a server to display the already read content. All these operations are expensive and time consuming.

**Asynchronity of Node.**

NODE is Synchronous programming language, well to understand that, let us reconsider the above scenario. To ease up the utilization of the CPU while it is sitting idle, we must consider threads, like to handle the above situation we need to bifurcate our jobs into threads, so they work in parallel and our program don’t have to wait for the I/O to complete.

Node on the other hand is a single thread programming language. It has a mother thread and an event queue. Any request from the server gets stacked up in the queue and executes accordingly. So, when node stops for the I/O ops to complete, unlike our traditional languages like Java, Node fetches the very next job from the queue and starts executing it in the mother thread. Don’t worry if all this makes little sense to you now as we dive deep into the topic, you will have a better clarity by the end.

Ex:-

**Event Driven Programming Language:**

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

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 –

Ex:-

**What the hell are callbacks?**

In a synchronous program, you would write something along the lines of:

function processData () {

var data = fetchData ();

data += 1;

return data;

}

This works just fine and is very typical in other development environments. However, if fetchData takes a long time to load the data (maybe it is streaming it off the drive or the internet), then this causes the whole program to 'block' - otherwise known as sitting still and waiting - until it loads the data. Node.js, being an asynchronous platform, doesn't wait around for things like file I/O to finish - Node.js uses callbacks. A callback is a function called at the completion of a given task; this prevents any blocking, and allows other code to be run in the meantime.

The node.js way to deal with the above would look a bit more like this:

function processData (callback) {

fetchData(function (err, data) {

if (err) {

console.log("An error has occurred. Abort everything!");

return callback(err);

}

data += 1;

callback(data);

});

}

At first glance, it may look unnecessarily complicated, but callbacks are the foundation of Node.js. Callbacks give you an interface with which to say, "and when you're done doing that, do all this." This allows you to have as many IO operations as your OS can handle happening at the same time. For example, in a web server with hundreds or thousands of pending requests with multiple blocking queries, performing the blocking queries asynchronously gives you the ability to be able to continue working and not just sit still and wait until the blocking operations come back. This is a major improvement.

The typical convention with asynchronous functions (which almost all of your functions should be):

function asyncOperation ( a, b, c, callback ) {

// ... lots of hard work ...

if ( /\* an error occurs \*/ ) {

return callback(new Error("An error has occurred"));

}

// ... more work ...

callback(null, d, e, f);

}

asyncOperation ( params.., function ( err, returnValues.. ) {

//This code gets run after the async operation gets run

});

You will almost always want to follow the [error callback convention](https://nodejs.org/en/knowledge/errors/what-are-the-error-conventions), since most Node.js users will expect your project to follow them. The general idea is that the callback is the last parameter. The callback gets called after the function is done with all of its operations. Traditionally, the first parameter of the callback is the error value. If the function hits an error, then they typically call the callback with the first parameter being an Error object. If it cleanly exits, then they will call the callback with the first parameter being null and the rest being the return value(s).

## **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.

Tutorials Point 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());

});

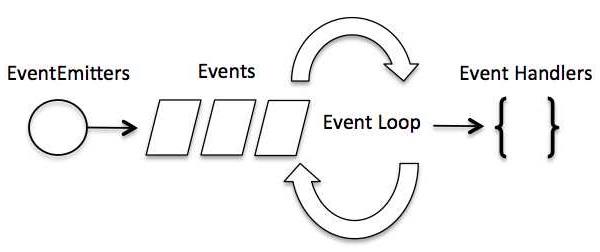
console.log("Program Ended");

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

Tutorials Point is giving self learning content

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

****

**Event Emittor:-**

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

## **EventEmitter Class**

As we have seen in the previous section, EventEmitter class lies in the events module. It is accessible via the following code −

// Import events module

var events = require('events');

// Create an eventEmitter object

var eventEmitter = new events.EventEmitter();

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

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

## **Methods**

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **addListener(event, listener)**  Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained. |
| 2 | **on(event, listener)**  Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be chained. |
| 3 | **once(event, listener)**  Adds a one time listener to the event. This listener is invoked only the next time the event is fired, after which it is removed. Returns emitter, so calls can be chained. |
| 4 | **removeListener(event, listener)**  Removes a listener from the listener array for the specified event. **Caution −** It changes the array indices in the listener array behind the listener. removeListener will remove, at most, one instance of a listener from the listener array. If any single listener has been added multiple times to the listener array for the specified event, then removeListener must be called multiple times to remove each instance. Returns emitter, so calls can be chained. |
| 5 | **removeAllListeners([event])**  Removes all listeners, or those of the specified event. It's not a good idea to remove listeners that were added elsewhere in the code, especially when it's on an emitter that you didn't create (e.g. sockets or file streams). Returns emitter, so calls can be chained. |
| 6 | **setMaxListeners(n)**  By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event. This is a useful default which helps finding memory leaks. Obviously not all Emitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited. |
| 7 | **listeners(event)**  Returns an array of listeners for the specified event. |
| 8 | **emit(event, [arg1], [arg2], [...])**  Execute each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise. |

## **Class Methods**

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

## **Events**

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

## **Example**

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

[Live Demo](http://tpcg.io/4qSKcR)

var events = require('events');

var eventEmitter = new events.EventEmitter();

// listener #1

var listner1 = function listner1() {

console.log('listner1 executed.');

}

// listener #2

var listner2 = function listner2() {

console.log('listner2 executed.');

}

// Bind the connection event with the listner1 function

eventEmitter.addListener('connection', listner1);

// Bind the connection event with the listner2 function

eventEmitter.on('connection', listner2);

var eventListeners = require('events').EventEmitter.listenerCount

(eventEmitter,'connection');

console.log(eventListeners + " Listner(s) listening to connection event");

// Fire the connection event

eventEmitter.emit('connection');

// Remove the binding of listner1 function

eventEmitter.removeListener('connection', listner1);

console.log("Listner1 will not listen now.");

// Fire the connection event

eventEmitter.emit('connection');

eventListeners = require('events').EventEmitter.listenerCount(eventEmitter,'connection');

console.log(eventListeners + " Listner(s) listening to connection event");

console.log("Program Ended.");

Now run the main.js to see the result −

$ node main.js

Verify the Output.

2 Listner(s) listening to connection event

listner1 executed.

listner2 executed.

Listner1 will not listen now.

listner2 executed.

1 Listner(s) listening to connection event

Program Ended.

**What is a Web Server?**

Imagine you are starving, you need something to eat, what you do is you go to a restaurant to order some food and then you wait until the food is ready and served. So your order is a request and serving of food is a response

So the restaurant is serving your request by providing you food. Anything which serves in the internet is a server. Therefore the formal definition for a server simply is

A Web [server](https://whatis.techtarget.com/definition/server) is a program that uses [HTTP](https://searchwindevelopment.techtarget.com/definition/HTTP) (Hypertext Transfer Protocol) to serve the files that form Web pages to users, in response to their requests, which are forwarded by their computers' HTTP clients. Dedicated computers and appliances may be referred to as Web servers as well.

Now there are two kind of servers, multiple threaded and single threaded.

In case of multiple threaded servers, they treat each request in a different thread, it is like having a separate waiter in a restaurant for every customer

In case of single thread, also called Event driven Server, there is an Event Loop and a Queue which holds different requests and execute them simultaneously, it is like having a master chef in a restaurant who takes the order and then assign others to do the same.

Node Js acts mainly as a Middle Ware.

**As a Middle Ware**

**Middleware** functions are functions that have 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.

Middleware functions can perform the following tasks:

* Execute any code.
* Make changes to the request and the response objects.
* End the request-response cycle.
* Call the next middleware 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

**Types of express middleware**

* Application level middleware **app.use**
* Router level middleware **router.use**
* Built-in middleware **express.static,express.json,express.urlencoded**
* Error handling middleware **app.use(err,req,res,next)**
* Thirdparty middleware **bodyparser,cookieparser**

**MVC Framework**

### Model

The Model component corresponds to all the data-related logic that the user works with. This can represent either the data that is being transferred between the View and Controller components or any other business logic-related data. For example, a Customer object will retrieve the customer information from the database, manipulate it and update it data back to the database or use it to render data.

The component ‘Model’ is responsible for managing the data of the application. It responds to the request from the view and it also responds to instructions from the controller to update itself.

Model classes can either be created manually or generated from database entities. We are going to see a lot of examples for manually creating Models in the coming chapters. Thus in this chapter, we will try the other option, i.e. generating from the database so that you have hands-on experience on both the methods.

### View

The View component is used for all the UI logic of the application. For example, the Customer view will include all the UI components such as text boxes, dropdowns, etc. that the final user interacts with.

As seen in the initial introductory chapters, View is the component involved with the application's User Interface. These Views are generally bind from the model data and have extensions such as html, aspx, cshtml, vbhtml, etc. In our First MVC Application, we had used Views with Controller to display data to the final user. For rendering these static and dynamic content to the browser, MVC Framework utilizes View Engines. View Engines are basically markup syntax implementation, which are responsible for rendering the final HTML to the browser

### Controller

Controllers act as an interface between Model and View components to process all the business logic and incoming requests, manipulate data using the Model component and interact with the Views to render the final output. For example, the Customer controller will handle all the interactions and inputs from the Customer View and update the database using the Customer Model. The same controller will be used to view the Customer data.

MVC Controllers are responsible for controlling the flow of the application execution. When you make a request (means request a page) to MVC application, a controller is responsible for returning the response to that request. The controller can perform one or more actions. The controller action can return different types of action results to a particular request.

The Controller is responsible for controlling the application logic and acts as the coordinator between the View and the Model. The Controller receives an input from the users via the View, then processes the user's data with the help of Model and passes the results back to the View.

## Flow Steps

Step 1 − The client browser sends request to the MVC Application.

Step 2 − Global.ascx receives this request and performs routing based on the URL of the incoming request using the RouteTable, RouteData, UrlRoutingModule and MvcRouteHandler objects.

Step 3 − This routing operation calls the appropriate controller and executes it using the IControllerFactory object and MvcHandler object's Execute method.

Step 4 − The Controller processes the data using Model and invokes the appropriate method using ControllerActionInvoker object

Step 5 − The processed Model is then passed to the View, which in turn renders the final output

**Node Modules**

Node Modules are simple or complex functionality organized in single or multiple file JS files which can be reused throughout the Node Js application

There are primarily three types of Node Modules:

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

**Core Modules**

The core modules include bare minimum but important functionalities of Node Js. These modules are compiled into its binary distribution and load automatically when Node process starts, however you need to import the node modules manually in order to use it in your application.

**RESTFUL APIS?**

**What is REST Architecture?**

REST stands for REpresentational State Transfer. REST is web standards based architecture and uses HTTP Protocol. REST is a software architectural style that defines the set of rules to be used for creating web services. Web services which follow the REST architectural style are known as RESTful web services. It revolves around resource where every component is a resource and a resource is accessed by a common interface using HTTP standard methods. REST was first introduced by Roy Fielding in 2000.

A REST Server simply provides access to resources and REST client accesses and modifies the resources using HTTP protocol. Here each resource is identified by URIs/ global IDs. REST uses various representation to represent a resource like text, JSON, XML but JSON is the most popular one.

**HTTP verbs:** Some of the common HTTP methods/verbs are described below:

* **GET:** Retrieves one or more resources identified by the request URI and it can cache the information receive.
* **POST:** Create a resource from the submission of a request and response is not cacheable in this case. This method is unsafe if no security is applied to the endpoint as it would allow anyone to create a random resource by submission.
* **PUT:** Update an existing resource on the server specified by the request URI.
* **DELETE:** Delete an existing resource on the server specified by the request URI. It always return an appropriate HTTP status for every request.
* GET, PUT, DELETE methods are also known as Idempotent methods. Applying an operation once or applying it multiple times has the same effect. Example: Delete any resource from the server and it succeeds with 200 OK and then try again to delete that resource than it will display an error message 410 GONE

## **RESTful Web Services**

A web service is a collection of open protocols and standards used for exchanging data between applications or systems. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer. This interoperability (e.g., communication between Java and Python, or Windows and Linux applications) is due to the use of open standards.

Web services based on REST Architecture are known as RESTful web services. These webservices uses HTTP methods to implement the concept of REST architecture. A RESTful web service usually defines a URI, Uniform Resource Identifier a service, which provides resource representation such as JSON and set of HTTP Methods

**Creating a simple Node Server**

To create a simple web server, we first need to import(require) the module ‘http’ , this module provides us the necessary set of function to run the server. We can use any port which is presently not busy in your machine.

**Const http = require(‘http’)**

**Const port = 3000**

The function createServer from http module takes a callback function which defines our server. So we need to create a callback function which takes Request and Response as two parameters and define them to handle Requests from client and the Responses to client.

**Const requestHandler = (request,response)=> {**

**Console.log(request.name,request.email,request.password)**

**Response.end(“Welcome to NodeJs Server”)**

**}**

**//Create the Server**

**Const server=http.createServer(requestHandler)**

The server variable holds an instance of the http module which will call listen to run the server forever, and our server will be open for requests

**server.listen(port,(err)=>{**

**if(err){**

**return (console.log(err))**

**}**

**Console.log(“Server is up and running in ${port}”)**

**})**

**Open up your CMD or Shell to run the command node index.js to run the server.**

**Introduction to Express Framwork.**

Express.js is a Node js web application server framework, which is specifically designed for building single-page, multi-page, and hybrid web applications.

It has become the standard server framework for node.js. Express is the backend part of something known as the MEAN stack.

. Following are some of the core features of Express framework −

* Allows to set up middlewares to respond to HTTP Requests.
* Defines a routing table which is used to perform different actions based on HTTP Method and URL.
* Allows to dynamically render HTML Pages based on passing arguments to templates.

Installing Express:-

npm install express –save

We will create an web server to unleash the power of express, the web server we created in the previous lesson, will get shorter and smarter.

const express = require('express');

const app = express();

const port=3000

app.get('/', function (request, response) {

res.send('Hello World');

})

var server = app.listen(port, function () {

const host = server.address().address

console.log("Our Server app is listening at http://%s:%s", host, port)

})

**Routing in Node**

We have seen a basic application which serves HTTP request for the homepage. Routing determines how an application will respond to different client request as per the REST protocol. We will be creating a simple GET and POST request through routing.

const express = require('express');

const app = express();

const port=3000

// This responds with "Hello World" on the homepage

app.get('/', function (request, response) {

console.log("Got a GET request for the homepage");

response.send('Hello GET');

})

// This responds a POST request for the homepage

app.post('/', function (request, response) {

console.log("Got a POST request for the homepage");

response.send('Hello POST');

})

const server = app.listen(8081, function () {

const host = server.address().address

console.log("Example app listening at http://%s:%s", host, port)

})

**Serving Static Files**

**How to make API endpoints**

**Postman API testing**

**Connect Mongo DB as a Database**

**Cookie Management**

**Example of a Web Server and School Library System.**

**File Systems**

**Buffers**

**Streams**

**File System**