# JavaScript Closures :

What is Closure?:

A closure is the combination of a function and the lexical environment within which that function was declared.

A closure is an inner function that has access to the outer (enclosing) function’s variables — scope chain. The closure has three scope chains: it has access to its own scope (variables defined between its curly brackets), it has access to the outer function’s variables, and it has access to the global variables.

Let see a closure example below:

function User(name){

var displayName = function(greeting){  
 console.log(greeting+' '+name);  
 }  
return displayName;  
}

var myFunc = User('Raj');

myFunc('Welcome '); //Output: Welcome Raj  
myFunc('Hello '); //output: Hello Raj

**Callbacks**  function definition:

A reference to executable code, or a piece of executable code, that is passed as an argument to other code.

From above definition,callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

1)

setTimeout(function() {

console.log("world");

}, 2000)

console.log("hello"); // output: hello world

2)

function greeting(name) {  
  
 console.log('Hello '+name);  
}

function processUserInput(callback) {  
 //var name = prompt('Please enter your name.');  
 name='lnt';  
 callback(name);  
}

processUserInput(greeting); //output Hello lnt

**Promises** definition:

***The*Promise*object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.***

**Promise** represents the result of asynchronous function. Promises can be used to avoid **chaining of callbacks**. In JavaScript, so whenever JavaScript Code Execute as Asynchronously, need to handle an operation as one of the way would be using promises.

A Promise is in one of these states:

* *pending*: initial state, neither fulfilled nor rejected.
* *fulfilled*: meaning that the operation completed successfully.
* *rejected*: meaning that the operation failed.

var promise1 = new Promise(function(resolve,reject){

isDbOperationCompleted = false;

if(isDbOperationCompleted){  
 resolve('Completed');  
} else{  
 reject('Not completed');  
}

});

promise1.then(function(result){  
 console.log(result); //Output : Completed  
}).catch(function(error){  
 console.log(error); //if isDbOperationCompleted=FALSE   
 //Output : Not Completed  
})

### Async & Await:

[Babel](https://babeljs.io/) now supporting async/await out of the box, and ES2016 (or ES7) just around the corner, **async & await basically just syntactic sugar on top of Promises**, these two keywords alone should make writing asynchronous code in Node much more bearable.

In JavaScript Asynchronous pattern handled in various versions,

***ES5 -> Callback***

***ES6 -> Promise***

***ES7 -> async & await***

However, what a lot of people may have missed is that the entire foundation for async/await is **promises**. In fact every async function you write will return a promise, and every single thing you await will ordinarily be a promise.

async function getUserDetail() {  
 try {  
 let users = await getUsers();  
 return users[0].name;  
 } catch (err) {  
 return {  
 name: 'default user'  
 };  
 }  
}

# Building a Restful CRUD API with Node.js, Express and MongoDB

### What is Node.js?

*Node.js is an open source server framework, completely free, and used by thousands of developers around the world.*

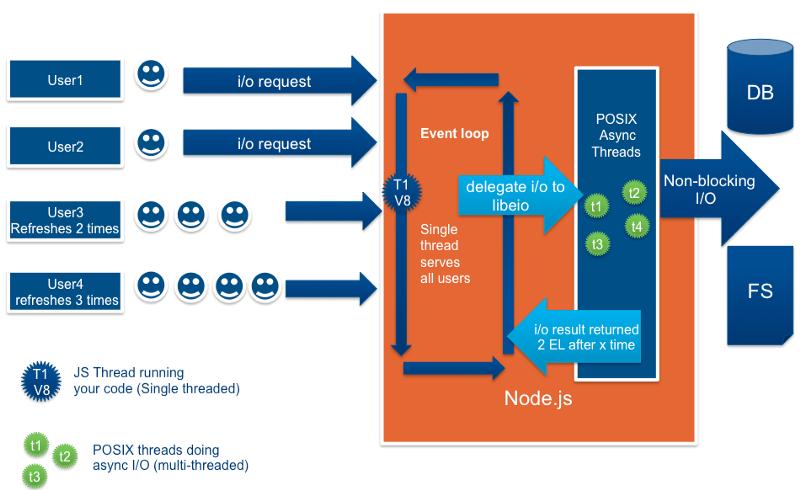
Node.js is an open source runtime environment for server-side and networking applications and is single threaded.

It uses Google JavaScript V8 Engine to execute code.

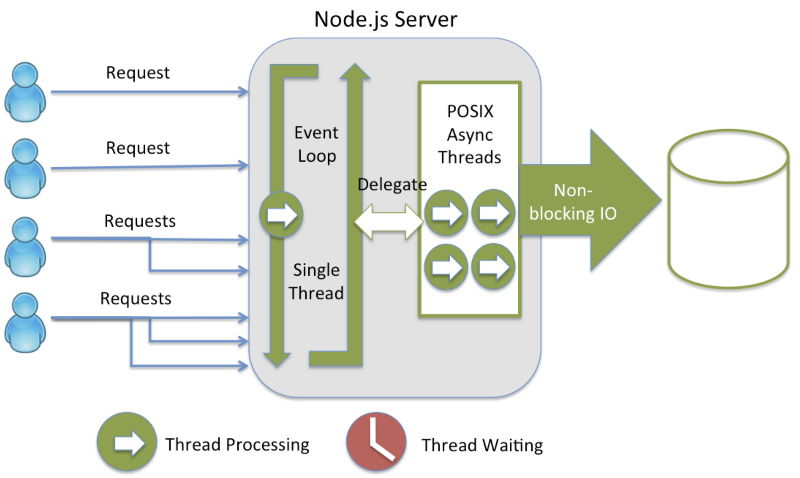
It is a cross platform environment and can run on Microsoft Windows, Linux

# Node.js Architecture:

**How Node.Js Single Thread mechanism Work ? Understanding Event Loop in NodeJs**



In NodeJs as we received request to server from the client Instead of create a new thread for each user request,the server just add the request to the event queue, each consecutive or concurrent request lined up in queue to be processed by the event loop:

In that diagram flow of an NodeJs Handling user request, 

Nodejs handling user request &Operates threads

**Event loop**is an Single thread which serves every thread ,event loop thread is run with infinite while loop there is **no exit** condition, So event loop keep during running entire application runtime ,

event loop pulling thread from queue the order they put in and if the task is just non-blocking and just involved CPU utilisation ,then process it ,if that succeeding task put back into the event queue for further process otherwise return the response to the user,

If there is an blocking task for example File I/O or db operations ,It utilises the one of the thread in thread pool,these child threads to do background tasks, If the blocking task has processing asynchronously and the succeeding the task has put into the event queue and finally the response is send back to the user.

[**Express**](https://expressjs.com/)  is a framework used for **Node** and it is most commonly used as a web application for **node js**. **Express** is just a module framework for Node that you can use for applications that are based on server/s that will "listen" for any input/connection requests from clients.

Its one of the most popular web frameworks for node.js. It is built on top of node.js **http** module, and adds support for **routing**, **middleware**, **view system** etc. It is very simple and minimal, unlike other frameworks that try do way to much, thereby reducing the flexibility for developers to have their own design choices.

[**Mongoose**](http://mongoosejs.com/) is an ODM (Object Document Mapping) tool for Node.js and MongoDB. It helps you convert the objects in your code to documents in the database and vice versa.

[body-parser](https://www.npmjs.com/package/body-parser) is a module that parses the request (of various content types) and creates a req.body object that we can access in our routes.

# Sample Application

Simple web server, configuring the database, Note model and different routes for handling all the CRUD operations

-> [let](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let), [const](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const), [arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions), [promises](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises)

We create an express app, and add two body-parser middlewares using express’s app.use() method. A [middleware](http://expressjs.com/en/guide/writing-middleware.html) is a function that has access to the request and responseobjects. It can execute any code, transform the request object, or return a response.

**Creating the Application**

**1. Fire up your terminal and create a new folder for the application.**

$ mkdir node-easy-notes-app

**2. Initialize the application with a package.json file**

Go to the root folder of your application and type npm init to initialize your app with a package.json file.

$ cd node-easy-notes-app

$ npm init

name: (node-easy-notes-app)

version: (1.0.0)

entry point: (index.js) server.js

test command:

git repository:

keywords: Express RestAPI MongoDB Mongoose Notes

author: callicoder

license: (ISC) MIT

About to write to /Users/rajeevkumarsingh/node-easy-notes-app/package.json:

{

"name": "node-easy-notes-app",

"version": "1.0.0",

"description": "Never miss a thing in Life. Take notes quickly. Organize and keep track of all your notes.",

"main": "server.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [

"Express",

"RestAPI",

"MongoDB",

"Mongoose",

"Notes"

],

"author": "callicoder",

"license": "MIT"

}

Is this ok? (yes) yes

Note that I’ve specified a file named server.js as the entry point of our application. We’ll create server.js file in the next section.

**3. Install dependencies**

We will need express, mongoose and body-parser modules in our application. Let’s install them by typing the following command -

$ npm install express body-parser mongoose --save

I’ve used --save option to save all the dependencies in the package.json file. The final package.json file looks like this -

{

"name": "node-easy-notes-app",

"version": "1.0.0",

"description": "Never miss a thing in Life. Take notes quickly. Organize and keep track of all your notes.",

"main": "server.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [

"Express",

"RestAPI",

"MongoDB",

"Mongoose",

"Notes"

],

"author": "callicoder",

"license": "MIT",

"dependencies": {

"body-parser": "^1.18.3",

"express": "^4.16.3",

"mongoose": "^5.2.8"

}

}

node-easy-notes-app

└── node\_modules/

└── package.json

**Setting up the web server**

Let’s now create the main entry point of our application. Create a new file named **server.js** in the root folder of the application with the following contents -

const express = require('express');

const bodyParser = require('body-parser');

// create express app

const app = express();

// parse requests of content-type - application/x-www-form-urlencoded

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

// parse requests of content-type - application/json

app.use(bodyParser.json())

// define a simple route

app.get('/', (req, res) => {

res.json({"message": "Welcome to node demo application."});

});

// listen for requests

app.listen(3000, () => {

console.log("Server is listening on port 3000");

});

//to start app by typing npm start

"scripts": {

"test": "start",

"start": "node index.js"

},

## Configuring and Connecting to the database

I like to keep all the configurations for the app in a separate folder. Let’s create a new folder config in the root folder of our application for keeping all the configurations -

$ mkdir config

$ cd config

Now, Create a new file database.config.js inside config folder with the following contents -

module.exports = {

url: 'mongodb://localhost:27017/easy-notes'

}

We’ll now import the above database configuration in server.js and connect to the database using mongoose.

Add the following code to the server.js file after app.use(bodyParser.json()) line -

// Configuring the database

const dbConfig = require('./config/database.config.js');

const mongoose = require('mongoose');

mongoose.Promise = global.Promise;

// Connecting to the database

mongoose.connect(dbConfig.url, {

useNewUrlParser: true

}).then(() => {

console.log("Successfully connected to the database");

}).catch(err => {

console.log('Could not connect to the database. Exiting now...', err);

process.exit();

});

Please run the server and make sure that you’re able to connect to the database -

$ node server.js

Server is listening on port 3000

Successfully connected to the database

## Defining the Note model in Mongoose

Next, We will define the Note model. Create a new folder called app inside the root folder of the application, then create another folder called models inside the app folder -

$ mkdir -p app/models

$ cd app/models

Now, create a file called note.model.js inside app/models folder with the following contents -

const mongoose = require('mongoose');

const NoteSchema = mongoose.Schema({

title: String,

content: String

}, {

timestamps: true

});

module.exports = mongoose.model('Note', NoteSchema);

The Note model is very simple. It contains a title and a content field. I have also added a [timestamps](http://mongoosejs.com/docs/guide.html#timestamps) option to the schema.

Mongoose uses this option to automatically add two new fields - createdAt and updatedAt to the schema.

## Defining Routes using Express

Next up is the routes for the Notes APIs. Create a new folder called routes inside the appfolder.

$ mkdir app/routes

$ cd app/routes

Now, create a new file called note.routes.js inside app/routes folder with the following contents -

module.exports = (app) => {

const notes = require('../controllers/note.controller.js');

// Create a new Note

app.post('/notes', notes.create);

// Retrieve all Notes

app.get('/notes', notes.findAll);

// Retrieve a single Note with noteId

app.get('/notes/:noteId', notes.findOne);

// Update a Note with noteId

app.put('/notes/:noteId', notes.update);

// Delete a Note with noteId

app.delete('/notes/:noteId', notes.delete);

}

Note that We have added a require statement for note.controller.js file. We’ll define the controller file in the next section. The controller will contain methods for handling all the CRUD operations.

Before defining the controller, let’s first include the routes in server.js. Add the following require statement before app.listen() line inside server.js file.

// ........

// Require Notes routes

require('./app/routes/note.routes.js')(app);

// ........

If you run the server now, you’ll get the following error -

$ node server.js

module.js:472

throw err;

^

Error: Cannot find module '../controllers/note.controller.js'

This is because we haven’t defined the controller yet. Let’s do that now.

## Writing the Controller functions

Create a new folder called controllers inside the app folder, then create a new file called note.controller.js inside app/controllers folder with the following contents -

const Note = require('../models/note.model.js');

// Create and Save a new Note

exports.create = (req, res) => {

};

// Retrieve and return all notes from the database.

exports.findAll = (req, res) => {

};

// Find a single note with a noteId

exports.findOne = (req, res) => {

};

// Update a note identified by the noteId in the request

exports.update = (req, res) => {

};

// Delete a note with the specified noteId in the request

exports.delete = (req, res) => {

};

Let’s now look at the implementation of the above controller functions one by one -

#### ****//Creating a new Note****

// Create and Save a new Note

exports.create = (req, res) => {

// Validate request

if(!req.body.content) {

return res.status(400).send({

message: "Note content can not be empty"

});

}

// Create a Note

const note = new Note({

title: req.body.title || "Untitled Note",

content: req.body.content

});

// Save Note in the database

note.save()

.then(data => {

res.send(data);

}).catch(err => {

res.status(500).send({

message: err.message || "Some error occurred while creating the Note."

});

});

};

#### ****//Retrieving all Notes****

// Retrieve and return all notes from the database.

exports.findAll = (req, res) => {

Note.find()

.then(notes => {

res.send(notes);

}).catch(err => {

res.status(500).send({

message: err.message || "Some error occurred while retrieving notes."

});

});

};

#### ****//Retrieving a single Note****

// Find a single note with a noteId

exports.findOne = (req, res) => {

Note.findById(req.params.noteId)

.then(note => {

if(!note) {

return res.status(404).send({

message: "Note not found with id " + req.params.noteId

});

}

res.send(note);

}).catch(err => {

if(err.kind === 'ObjectId') {

return res.status(404).send({

message: "Note not found with id " + req.params.noteId

});

}

return res.status(500).send({

message: "Error retrieving note with id " + req.params.noteId

});

});

};

#### ****//Updating a Note****

// Update a note identified by the noteId in the request

exports.update = (req, res) => {

// Validate Request

if(!req.body.content) {

return res.status(400).send({

message: "Note content can not be empty"

});

}

// Find note and update it with the request body

Note.findByIdAndUpdate(req.params.noteId, {

title: req.body.title || "Untitled Note",

content: req.body.content

}, {new: true})

.then(note => {

if(!note) {

return res.status(404).send({

message: "Note not found with id " + req.params.noteId

});

}

res.send(note);

}).catch(err => {

if(err.kind === 'ObjectId') {

return res.status(404).send({

message: "Note not found with id " + req.params.noteId

});

}

return res.status(500).send({

message: "Error updating note with id " + req.params.noteId

});

});

};

The {new: true} option in the [findByIdAndUpdate()](http://mongoosejs.com/docs/api.html#findbyidandupdate_findByIdAndUpdate) method is used to return the modified document to the then() function instead of the original.

#### ****//Deleting a Note****

// Delete a note with the specified noteId in the request

exports.delete = (req, res) => {

Note.findByIdAndRemove(req.params.noteId)

.then(note => {

if(!note) {

return res.status(404).send({

message: "Note not found with id " + req.params.noteId

});

}

res.send({message: "Note deleted successfully!"});

}).catch(err => {

if(err.kind === 'ObjectId' || err.name === 'NotFound') {

return res.status(404).send({

message: "Note not found with id " + req.params.noteId

});

}

return res.status(500).send({

message: "Could not delete note with id " + req.params.noteId

});

});

};

You can check out the documentation of all the methods that we used in the above APIs on Mongoose’s official documentation -

* [Mongoose save()](http://mongoosejs.com/docs/api.html#document_Document-save)
* [Mongoose find()](http://mongoosejs.com/docs/api.html#find_find)
* [Mongoose findById()](http://mongoosejs.com/docs/api.html#findbyid_findById)
* [Mongoose findByIdAndUpdate()](http://mongoosejs.com/docs/api.html#findbyidandupdate_findByIdAndUpdate)
* [Mongoose findByIdAndRemove()](http://mongoosejs.com/docs/api.html#findbyidandremove_findByIdAndRemove)

now test all the APIs one by one using postman.

Thank You... :-)