
node
JS

What is Node.js?

- ✓ JavaScript runtime built on Chrome's V8 JavaScript engine
- ✓ JavaScript running on the server
- ✓ Used to build powerful, fast & scalable web applications
- ✓ Uses an event-driven, non-blocking I/O model

MOTIVATION

- Node.js might be the most exciting single piece of software in the current **JavaScript universe** — used by **LinkedIn**, **Groupon**, **PayPal**, **Walmart**, etc.
- Node.js is one of the **most watched projects** on GitHub; it has more than million
  modules in **npm package manager**.
- Node.js combined with a client-side **MV* framework**, a **NoSQL database** (such as MongoDB or CouchDB) and **JSON** offers a unified JavaScript **development stack**.

Single Threded Vs MultiThreded

- **Node**. js is a **single threaded** language which in background uses **multiple threads** to execute asynchronous code. **Node**. js is non-blocking which means that all functions (callbacks) are delegated to the event loop and they are (or can be) executed by different threads
- In computer programming, **single-threading** is the processing of one command at a time.

Synchronous Vs Asynchronous
Blocking Vs Non Blocking

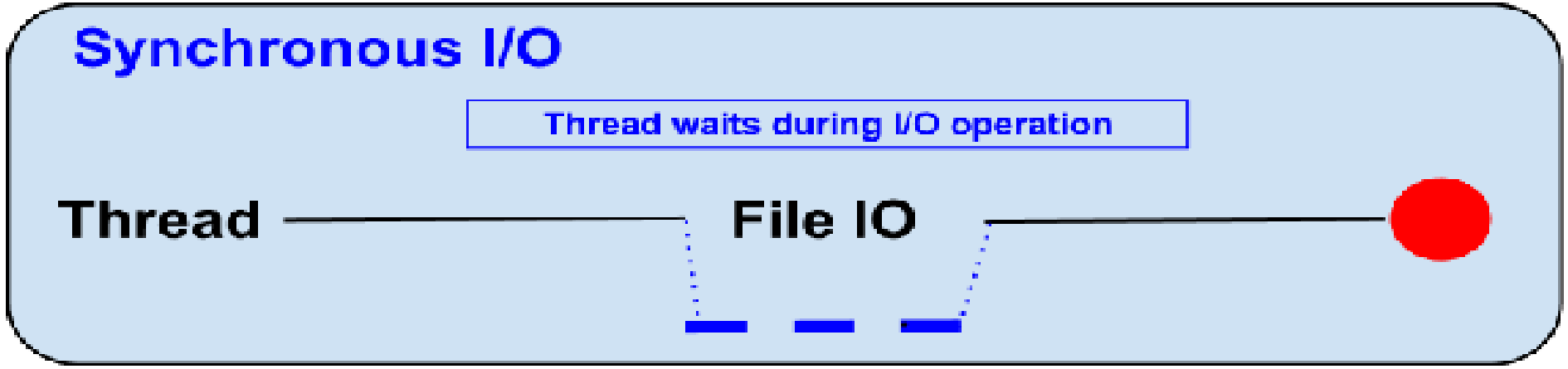
Blocking is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation completes. This happens because the event loop is unable to continue running JavaScript while a **blocking** operation is occurring.

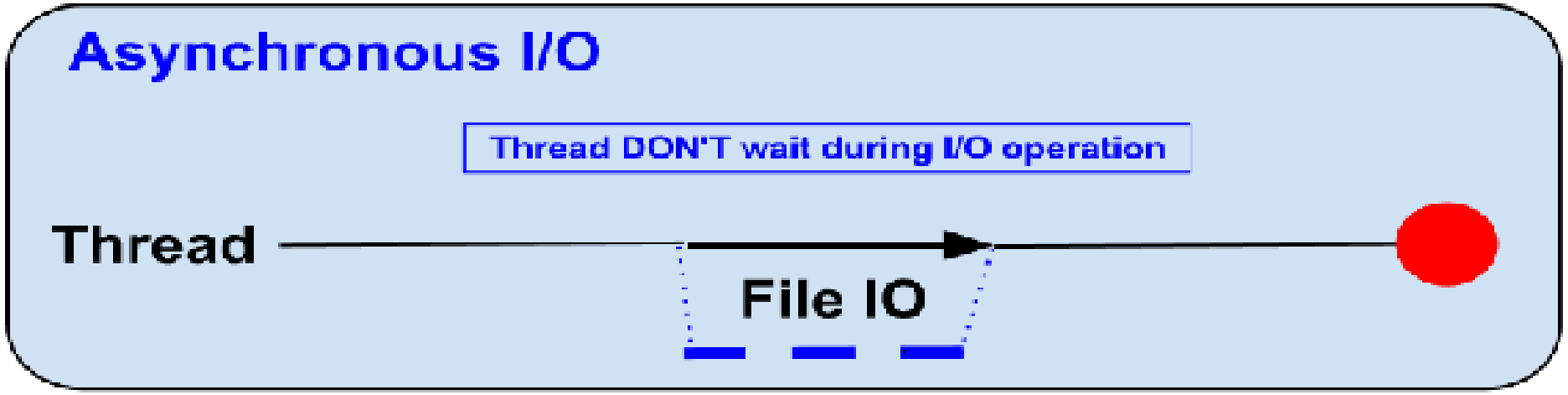
Synchronous methods in the Node.js standard library that use libuv are the most commonly used **blocking** operations.

Blocking methods execute synchronously and non-blocking methods execute asynchronously.

Blocking vs Non-Blocking

- Example :: Read data from file and show data





WHO IS USING NODE.JS IN PRODUCTION?

- **Yahoo!** : iPad App **Livestand** uses Yahoo! Manhattan framework which is based on Node.js.
- **LinkedIn** : LinkedIn uses a combination of Node.js and MongoDB for its mobile platform. iOS and Android apps are based on it.
- **eBay** : Uses Node.js along with ql.io to help application developers in improving eBay's end user experience.
- **Dow Jones** : The WSJ Social front-end is written completely in Node.js, using Express.js, and many other modules.

What Can We Build With Node.js?

- ✓REST APIs & Backend Applications
- ✓Real-Time Services (Chat, Games, etc)
- ✓Blogs, CMS, Social Applications
- ✓Utilities & Tools
- ✓Anything that is not CPU-intensive

NPM

- ✓ Node.js Package Manager
- ✓ Used to install node programs/modules
- ✓ Easy to specify and link dependencies
- ✓ Modules get installed into the **“node_modules”** folder

```
npm install express
```

```
npm install –g express
```

NPM

- **npm** makes it easy for JavaScript developers to share and reuse code in the form of modules.
- **npm** comes preinstalled with Node distributions.
- **npm** runs through the **command line** and allows to retrieve modules from the **public package registry** maintained on http://npmjs.org

NODE Package manager (npm)

- Node.js supports modularity. Each modules can be bundled under a single package.
- NPM is used to install, update, uninstall and configure Node JS Platform modules/packages very easily.

Module management: NPM

Common js modules are installed with npm

- npm install
- npm uninstall
- npm update

Modules can be installed globally (applications), or locally to your applications (dependencies).

NPM COMMANDS

- npm install
- npm install bootstrap , jquery
- npm install –g @angular/cli@latest
- npm install bootstrap@4.0
- npm install fontawesome
- npm install normalize
- npm i whoami // to run use whoami

NPM COMMANDS

- npm update
- npm uninstall
- npm help
- npm init
- `npm install -g npm@latest`
- npm install -g nodemon
- **npm list** to show the installed packages in the current project as a dependency tree.

NPM COMMANDS

- You can get the latest (or any) package info with the view command.

npm view bootstrap

- The docs command works similar to repo but it will do it’s best to find the canonical documentation site for a package.

npm docs bootstrap

NPM COMMANDS

- **Shorthands and Flags**
- Like npm i stands for npm install, there are other phonetic shorthands that exist, including a shorthand for test and combinations of test and install. These quick keystrokes can save you whole seconds over the course of a year—you're welcome.

- npm i – install // create node_modules folder from package.json
- npm t – test // for Unit testing
- npm ci - clean-install
- npm search packagename // npm search nodemon
- npm install -g npm@latest // install latest version of npm
- npm // all information regarding the npm command
- ppm update // will update all the packages

PACKAGE.JSON

- A package is a folder containing a program described by a **package.json file — a package descriptor**.
- **A package descriptor** is used to store all **metadata** about the module, such as name, version, description, author etc.
- This file is a **manifest** of your **Node project** and should be placed at your **project root** to allow:
  - **reinstalling** your dependencies by defining a **dependencies** field;
  - **publishing** your module to npm by defining the **name** and **version** fields,
  - **storing** common scripts related to the package by defining the **scripts** object.

package.json File

- Goes in the root of your package/application
- Tells npm how your package is structured and what to do to install it

```
{
  "name": "mytasklist",
  "version": "1.0.0",
  "description": "Simple task manager",
  "main": "server.js",
  "author": "Brad Traversy",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.15.2",
    "express": "^4.14.0",
    "mongojs": "^2.4.0"
  }
}
```

```
npm init
```

NODEJS HANDS-ON

INSTALL NODEJS

HTTPS://WWW.NODEJS.ORG

Node

GET REPL (NODE SHELL)

REPL (READ, EVAL, PRINT, LOOP) is a computer environment similar to Shell (Unix/Linux) and command prompt. Node comes with the **REPL** environment when it is installed. System interacts with the user through outputs of commands/expressions **used**. It is useful in writing and debugging the codes.

.break Sometimes you get stuck, this gets you out

.clear Alias for .break

.editor Enter editor mode

.exit Exit the repl

.help Print this help message

Use ctrl+l to clear the node prompt

If all is well you should drop into a console where you can type arbitrary JavaScript commands. You can do maths, create functions, assign variables, everything you can do with JavaScript.

Examples

1.

```
D:/nodehandson/> node
➢   for (i=0;i<=10;i++)
➢   {
... console.log(i)
... }
```

2.

```
function greeting(greet){
... console.log(greet)
... }
> greeting ("Hello World")
Hello World
```

EXAMPLE

.editor

1.

```
var x=10;
var y=20;
var z=x+y;
console.log (“Result Is :”+z);
```

Result Is : 30

2.

```
console.log("Hello World")
var x=new Date();
console.log(x);
```

Using .js File

You can also run saved programs using the node command. Node files have a .js (JavaScript) suffix, like this: program.js.

We can execute our JavaScript program using node like this:

node app.js

app.js

```
console.log("Hello World")
var x=new Date();
console.log(x);
```

Use VS Code & run this file using **node app.js**

You can make Node output to the terminal using console.log(). Create a file called app.js and add a line that uses console.log() to tell Node write "Hello World" to the console.

Using .js File

```
// Operators & Logic
var v1=80;
var v2=140;
var v3=100;
if(v1>v2 && v1>v3)
{
   console.log("V1 is greatest")
}
else if(v2>v1 && v2>v3)
{
   console.log("V2 is greatest")
}
else
{
   console.log("V3 is greatest")
}
```

Using .js File

```
// logic with for
var i;
for(i=1; i<=10;i++)
{
   console.log(i)
}
```

Using .js File

```
// Using Switch Case
var location=7;
switch(location)
{
   case 1:  console.log("Location Is North")
   break;
   case  2:  console.log("Location Is South")
   break;
   case 3:  console.log("Location Is East")
   break;
   case  4:  console.log("Location Is West")
   break;
   default :
      console.log("Please enter Another Location")

}
```

Node Task 2 :

Using switch case find out the given number is Odd or Even in nodejs

NodeJS Important Concepts

Callback Function

A callback function is a function (It can be any function Anonymous Function, Arrow Function) passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

```
function show(){
    console.log("I am show Function");
}
function geeky(callback){
    callback();
}
geeky(show);
```

Node Js Callback Function

Read data from file

When read data completed, show data

Do other tasks


```
fs.readFile( “test.txt”, function( err, data ) {
console.log(data);
});
```

NodeJS Important Concepts

Anonymous Functions

Anonymous functions allow the creation of functions which have no specified name.

- Can be stored in a Variable
- Can be Returned in a Function
- Can be pass in a Function

Syntax: -

```
function ( ) {
  body of function;
};
```

NodeJS Important Concepts

```
<!DOCTYPE html>
<html>
    <head><title>Geeky Shows</title>
    </head>
    <body>
        <script>
            // Anonymous Function
            var disp = function(){
                document.write("Geekyshows");
            };
            disp();
        </script>
    </body>
</html>
```

NodeJS Important Concepts

Arrow Function

An arrow function expression (previously, and now incorrectly known as fat arrow function) has a shorter syntax compared to function expressions. Arrow functions are always anonymous.

Syntax: -

```
( ) => { statements};
```

```
var myfun = function show( ) {
   document.write("GeekyShows");
};

var myfun = ( ) =>{document.write("GeekyShows");};
```

NodeJS Important Concepts

```
<html>
    <head><title>Geeky Shows</title>
    </head>
    <body>
        <script>

            // Arrow Function
            var myfun = () => {
                document.write("GeekyShows");
                };
            myfun();
        </script>
    </body>
</html>
```

FILE SYSTEM

FILE SYSTEM

In which we interact with the file system, and serve a web page from a file.

Filing

We can access the file system using the fs module. This gives us methods to read from and write to a file.

The fs module allows us to specify a callback function which will have access to our file. The file will be opened for you, then the callback will be invoked and the file passed in to it. You don't need to worry about opening and closing the file, you just need to write code to talk to the file object which you will receive.

Node is not blocked during IO, because your code is not invoked until the file is ready for writing.

FILE SYSTEM

- **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.**

FILE SYSTEM

First require fs

We will need the fs module, so first require it, like this:

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

Create & Write Content

- **rwfile.js**

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

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

Append File Content

- **appendfile1.js**

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

fs.appendFile('test.txt', ‘ File Appended', function (err) {
                    if (err)
      console.log(err);
                    else
      console.log('Append operation complete.');
});
```

Read File Content

- **readfile.js**

```
var fs = require('fs'),
  path = './test.txt';
fs.readFile(path, function(err, fcontent) {
    console.log('' + fcontent);
});
```

Delete File Content

- **deletefile.js**

```
var fs = require('fs');
fs.unlink('test.txt', function () {
   console.log(‘Delete operation complete.');
});
```

Reading Directories

We can use the fs.readdir method to list all the files and directories within a specified path:

```
const fs = require('fs')

fs.readdir('./', (err, files) => {
        if (err) {
                console.error(err)
                return
        }

        console.log('files: ', files)
})
// output  will give the output: files:  [ 'index.js', 'tmp' ]
```

Creating Directory

Directories can be created and removed with the fs.mkdir and fs.rmdir methods respectively:

Creating a new directory:

```
const fs = require('fs')

fs.mkdir('./newdir', err => {
	if (err) {
		console.error(err)
		return
	}

	console.log('directory created')
})
```

Removing a directory:

```
const fs = require('fs')

fs.rmdir('./newdir', err => {
	if (err) {
		console.error(err)
		return
	}

	console.log('directory deleted')
})
```

FILE SYSTEM

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**

FILE SYSTEM

- 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 shown below.

app.js

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

The msg variable becomes a 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
```

FILE SYSTEM

- **Export Function as a Class**
- **In JavaScript, a function can be treated like a class. The following example exposes a function that can be used like a class.**

FILE SYSTEM

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 the new keyword. Run the above example, as shown below.

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

CREATING CUSTOM MODULE IN NODEJS

A node module is a simple JavaScript file that saves an object in a variable called module.exports

For example, we could create a cat module like this

cat.js

```
var cat = {
  legs: 4,
  head: 2,
  ears: 2,
  sayHello: function() {
    console.log('meow');
  }
};
module.exports = cat;
```

next.js

In another file we can now require('./cat'), like so:

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

NODEJS MODULES

MODULES

- Modules are **reusable software components** that form the building blocks of applications.

- **Modular programming** is a software design technique that emphasizes separating the functionality of a program into **independent**, **interchangeable modules** such that each covers only **one aspect of the desired functionality**.

- Modules should be **FIRST**:
  - **F**ocused.
  - **I**ndependent.
  - **R**eusable.
  - **S**mall.
  - **T**estable.

Node.js Module Types

There are basically three types of modules,

1. File based Modules
2. Core Modules
3. External or Third Party Modules

Node.js **File based Modules (Custom Module)**

This types of modules are created locally in your node.js application. It includes different functionalities of your application in seperate files and folders.

The require function is used to import a module in nodes.js application.

Node.js **File based Modules (Custom Module)**

Example

Create *LocalModule.js* file and insert the follwing code.

```
// create a variable calc that have four function add, subtract, multiply and divide.

const calc = {
   // Add the two number
   add: function (num1, num2) {
      return num1 + num2
   },
   // Subtract the two number
   subtract: function (num1, num2) {
      return num1 - num2;
   },
   // multiply the two number
   multiply: function (num1, num2) {
      return num1 * num2;
   },
   // divide the two number
   divide: function (num1, num2) {
      return num1 / num2
   }
};

// export the modules to consume on different modules.
module.exports.calc = calc;
```

Node.js **File based Modules (Custom Module)**

To use the above functionality in a different module, create **index.js** and write the code

// To use custom module declare the code like this.

```
const myCalculation = require('./LocalModule.js');
// Declare two variable
const num1 = 5;
const num2 = 4;

// consume the existing function from LocalModule.js

console.log(`Addition of ${num1} and ${num2} is:  ${myCalculation.calc.add(num1, num2)}`);
console.log(`Subtraction of ${num1} and ${num2} is: ` + myCalculation.calc.subtract(num1, num2));
console.log(`Multiply of ${num1} and ${num2} is: ` + myCalculation.calc.multiply(num1, num2));
console.log(`Divide of ${num1} and ${num2} is: ` + myCalculation.calc.divide(num1, num2));
```

Node.js command prompt

```
D:\Arvind\solution\nodesolution\nodeExamples>node index
Addition of 5 and 4 is:  9
Subtraction of 5 and 4 is: 1
Multiply of 5 and 4 is: 20
Divide of 5 and 4 is: 1.25

D:\Arvind\solution\nodesolution\nodeExamples>
```

Example - Custom Node Module

calc.js

```
//Creating a custom node module
// And making different functions
exports.add = function (a, b) {
  return a + b; // Adding the numbers
};
exports.sub = function (a, b) {
  return a - b; // Subtracting the numbers
};
exports.mul = function (a, b) {
  return a * b; // Multiplying the numbers
};
exports.div = function (a, b) {
  return a / b; // Dividing the numbers
};
```

Another way to show the Result

index.js

```
// Importing the custom node module with the below statement
var calculator = require('./calc');

var a = 21 , b = 67
console.log("Addition of " + a + " and " + b + " is " + calculator.add(a, b));
console.log("Subtraction of " + a + " and " + b + " is " + calculator.sub(a, b));
console.log("Multiplication of " + a + " and " + b + " is " + calculator.mul(a, b));
console.log("Division of " + a + " and " + b + " is " + calculator.div(a, b));
```

Node.js Core Modules

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.

Node.js **Core Modules**

It is also known as (built in modules).

These core modules are compiled into its binary distribution and load automatically when Node.js process starts.

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

```
1.const variableName = require('module_name');
2.const path = require('path');
```

Node.js **Core Modules**

Example

Write the below code in CoreModule.js file.

```
// declare the core library file.
const path = require('path');

//Use the core library file.
console.log("Directory of index file: " + path.dirname(__filename));
console.log("Extension of index file: " + path.extname(__filename));
```

Node.js command prompt

```
D:\Arvind\solution\nodesolution\nodeExamples>node CoreModule
Directory of index file: D:\Arvind\solution\nodesolution\nodeExamples
Extension of index file: .js

D:\Arvind\solution\nodesolution\nodeExamples>_
```

Node.js **Core Modules**

Nodejs HTTP Module:
It is a built-in module of node.js. It allows node.js applications to transfer data using HyperText Transfer Protocol (HTTP).
This module creates an HTTP server that listens to server ports and also gives responses back to the client.
Properties:

1. http.METHODS: this tells us all the methods available in http module.
Code to check HTTP methods:

```
let http = require('http');
console.log(http.METHODS)
```

Node.js **Core Modules**

File Edit Selection View Go Run Terminal Help — index.js - DataFlair - Visual Studio Code

EXPLORER

OPEN EDITORS
- index.js

DATAFLAIR
- node_modules
- index.js
- package-lock.json
- package.json
- sum.js

OUTLINE
- http

NPM SCRIPTS

index.js > ...

```
1  let http = require('http');
2  console.log(http.METHODS)
3
```

OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS

```
C:\Users\ASUS\Documents\visual studio code\DataFlair>node index.js
[
  'ACL',          'BIND',        'CHECKOUT',
  'CONNECT',      'COPY',        'DELETE',
  'GET',          'HEAD',        'LINK',
  'LOCK',         'M-SEARCH',    'MERGE',
  'MKACTIVITY',   'MKCALENDAR',  'MKCOL',
  'MOVE',         'NOTIFY',      'OPTIONS',
  'PATCH',        'POST',        'PROPFIND',
  'PROPPATCH',    'PURGE',       'PUT',
  'REBIND',       'REPORT',      'SEARCH',
  'SOURCE',       'SUBSCRIBE',   'TRACE',
  'UNBIND',       'UNLINK',      'UNLOCK',
  'UNSUBSCRIBE'
]

C:\Users\ASUS\Documents\visual studio code\DataFlair>
```


Ln 3, Col 1 Spaces: 4 UTF-8 CRLF JavaScript

Installing third party modules

Modules that are available online and are installed using the npm are called third party modules.

Examples of third party modules are express, mongoose, etc.

- NPM Install

npm install <options> <package unique name>

Eg - npm install express --save

Node.js “require”

The modules/packages available can be imported in .js file using “require” function.

Ex – require(“http”);

This helps in loading “http” package in current .js file.

Creating Server

- In NodeJS we create a simple Node server and ship out a website.
- Node is not a web server, but it comes bundled with modules that let you create a web server very easily.
- Node is single threaded and event driven. This is actually a very good architecture for a web server.
- We define callback functions that listen out for events, for example network connections, file system events, API calls, etc. When the event occurs, the callback function is executed.
- We can have many thousands or even millions of open connections with little impact on performance.

Creating Server

```
var http = require('http');
http.createServer(function (request, response) {
  response.writeHead(200);
  response.write('<h1>Hello Node!!!!</h1>\n');
  response.end();
}).listen(3000);
console.log('Server running at http://localhost:3000');  //127.0.0.1
```

// Save the file as server1.js

//Start the server with: node server1.js

Visit http://localhost:3000 to ensure you have achieved success.

Creating Server

1. Require http

First we need to require the http module. This comes built in to Node. We require this module and save the object that the module returns in a variable. We can then use this object to create a server.

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

2. Create the server

Now we need to create a server and set it listening.

The createServer function receives a callback function. This is the function that will be called when a network request comes in.

```
http.createServer(function (request, response) {
  // You have access to the request and response objects here.
}).listen(3000);
```

Handle Multiple URL

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.

Creating Server

3. Write to the response

The callback function receives two objects, which we typically call request and response (or req and res if you prefer)

Whenever someone hits our website, Our callback function will receive request and response objects. We can write whatever we like to the response head and body, and finally return the result:

```
response.writeHead(200); // write a 200 OK header

response.write('<h1>Hello Node!</h1>

'); // write to the body

response.end(); // return the response to the user
```

We call response.end() to return the response. If you forget this your server will never return any content. This is by design as it allows us to create long lasting connections that only return a result when something interesting happens.

Creating Server

HTTP

- To use the HTTP server and client one must require('http')
- The HTTP interfaces in Node.js are designed to support many features of the protocol

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

http.createServer( function (request, response) {

   var pathname = url.parse(request.url).pathname;

   console.log("Request for " + pathname + " received.");

   fs.readFile(pathname.substr(1), function (err, data) {
      if (err) {

         console.log(err);
         response.writeHead(404, {'Content-Type': 'text/html'});

      }else{
         response.writeHead(200, {'Content-Type': 'text/html'});
         response.write(data.toString());
      }
      response.end();
   });
}).listen(3000);

console.log('Server running at localhost:3000');
```

- Import Required Modules
- Creating Server
- Parse the fetched URL to get pathname
- Request file to be read from file system
- Creating Header with content type as text or HTML
- Generating Response
- Listening to port: 3000

Creating Server

TASKS :

- Modify your server to make it return a string of your choosing.
- Modify your server to make it listen to port 5000.
- Now modify your code. Try to write the current URL as a string to the response. If I visit http://localhost:3000/Home I should see a web page containing the word Home Page, ideally wrapped in an h1 tag.
- Exercise - A simple router
- Given that we can gain access to the URL, write a simple 2 page website that responds to http://localhost:3000/nacktschnecke and http://localhost:3000/about and serves content appropriately. Do this using if, else if and else.
- If no route matches, have your server return a 404 file not found status code and page.

Handle Multiple URL

nodeurl.js

- the following example demonstrates handling HTTP request and response in Node.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();

   }
```

Handle Multiple URL

```
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..')
```

Handle Multiple URL

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.

Multiple URL –open file

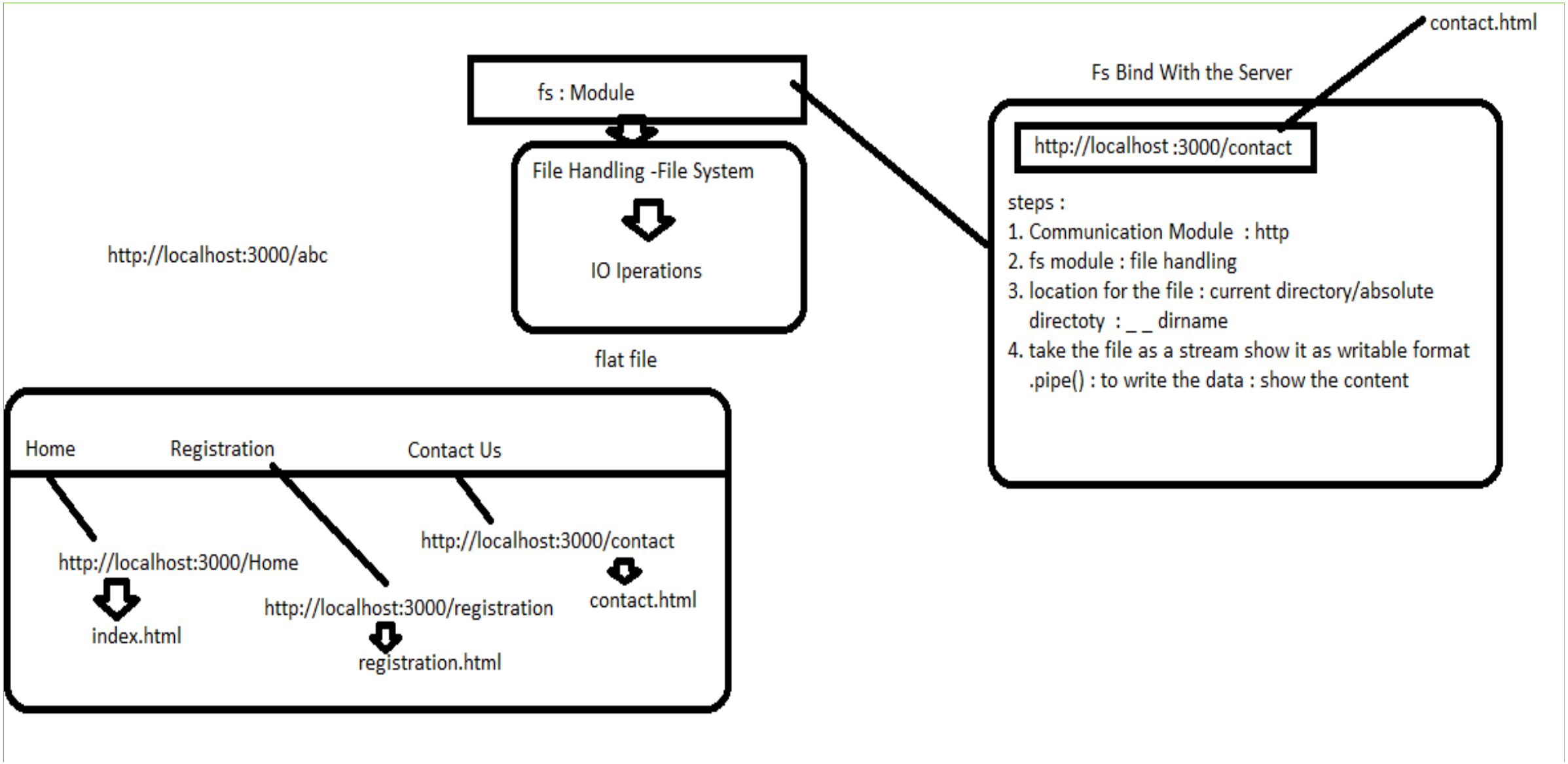
http : communication method : import

fs : file handling

_ _ dirname :

_dirname is an environment variable that tells you the absolute path of the directory containing the currently executing file.

Multiple URL –open file



Multiple URL –open file

```
var http = require("http");
var fs = require("fs");
var server = http.createServer(function (req, res) {
  console.log("A request was made: " + req.url);
  if (req.url === "/home" || req.url === "/") {
    res.writeHead(200, { "Content-Type": "text/html" });
    fs.createReadStream(__dirname + "/index.html").pipe(res);
  } else if (req.url === "/contact") {
    res.writeHead(200, { "Content-Type": "text/html" });
    fs.createReadStream(__dirname + "/contact.html").pipe(res);
  } else if (req.url === "/social") {
    res.writeHead(200, { "Content-Type": "text/html" });
    fs.createReadStream(__dirname + "/social.html").pipe(res);
  } else {
    var ninjas = [
      { name: "abc", age: 30 },
      { name: "def", age: 32 },
    ];
    res.writeHead(200, { "Content-Type": "application/json" });
    res.end(JSON.stringify(ninjas));
  }
});

server.listen(3000, "127.0.0.1");
console.log("Listening to port 3000");
```

Sending JSON Response

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

```
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..')
```

Node.js Request Module

The request module is used to make HTTP calls. It is the simplest way of making HTTP calls in node.js using this request module. It follows redirects by default.

Feature of Request module:

It is easy to get started and easy to use.

It is widely used and popular module for making HTTP calls.

Installation of request module:

You can visit the link Install Request module. You can install this package by using this command.

npm install request

Node.js Request Example

Filename: index.js
//npm install request

```
const request = require('request')
// Request URL
var url = 'https://jsonplaceholder.typicode.com/todos/1';
request(url, (error, response, body)=>{
        // Printing the error if occurred
        if(error) console.log(error)

        // Printing status code
        console.log(response.statusCode);

        // Printing body
        console.log(body);
});
```

Node.js Request Example

Task : Using REQUET module show the data on browser

Node.js Using ***AXIOS*** module

In this approach we will send request to getting a resource using AXIOS library. Axios is a promise base HTTP client for NodeJS. You, can also use it in the browser. Using promise is a great advantage when dealing with asynchronous code like network request.

Installing module:

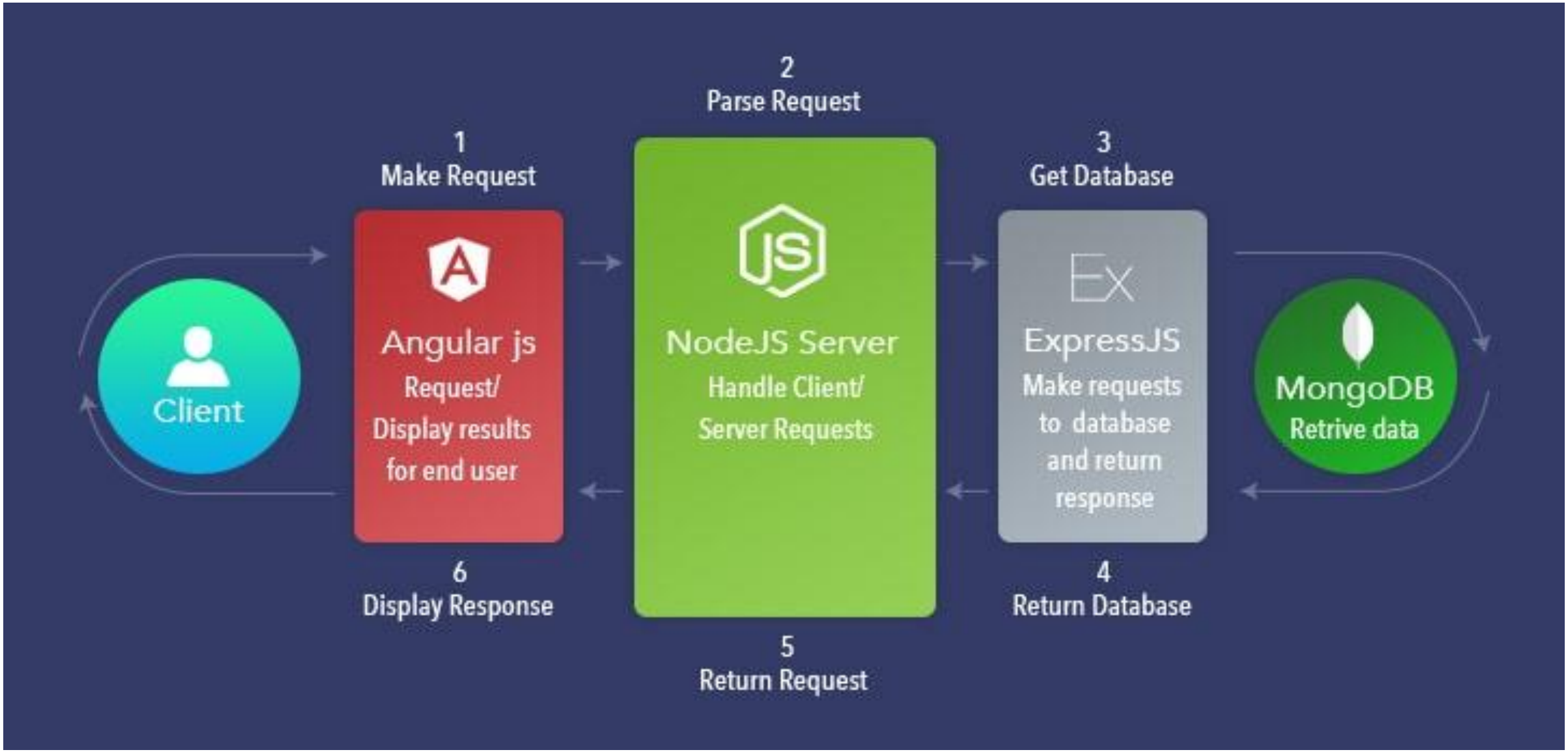
npm i axios

Node.js Using ***AXIOS*** module :Example

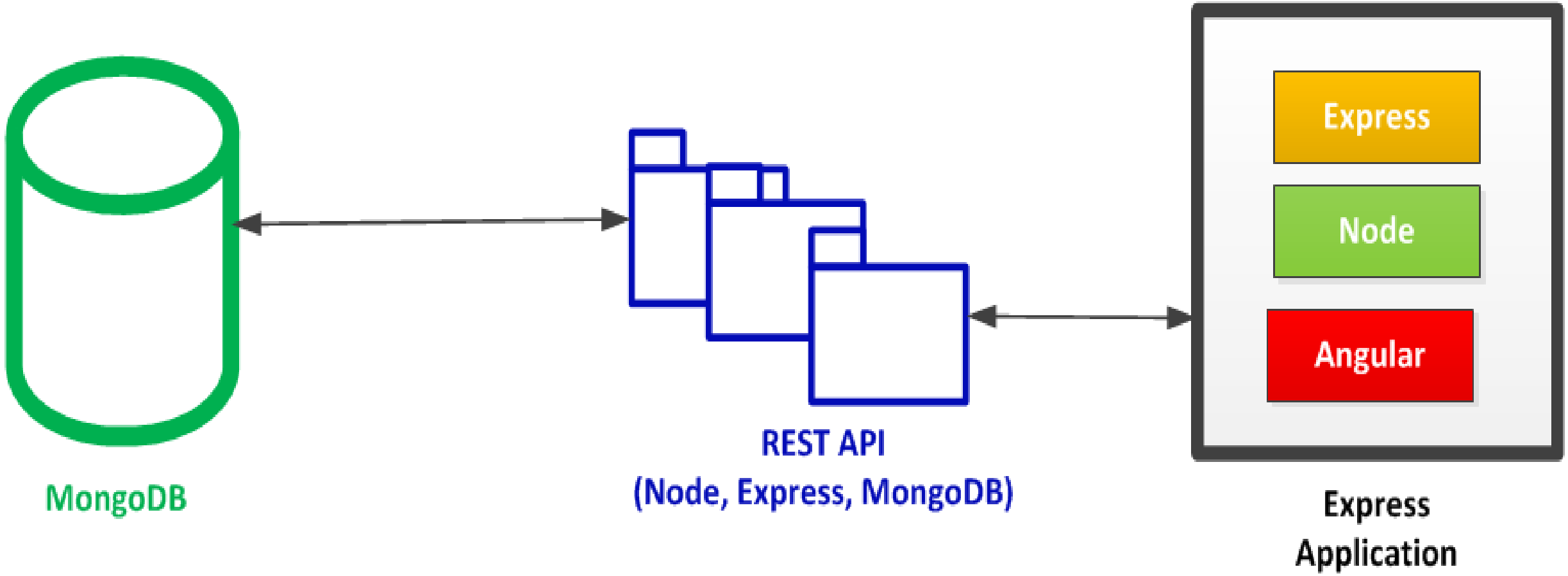
index.js

```
const axios = require('axios')

// Make request
axios.get('https://jsonplaceholder.typicode.com/posts/1')

// Show response data
.then(res => console.log(res.data))
.catch(err => console.log(err))
```

NodeJS with MEAN



NodeJS with MEAN



Node.js with MONGODB

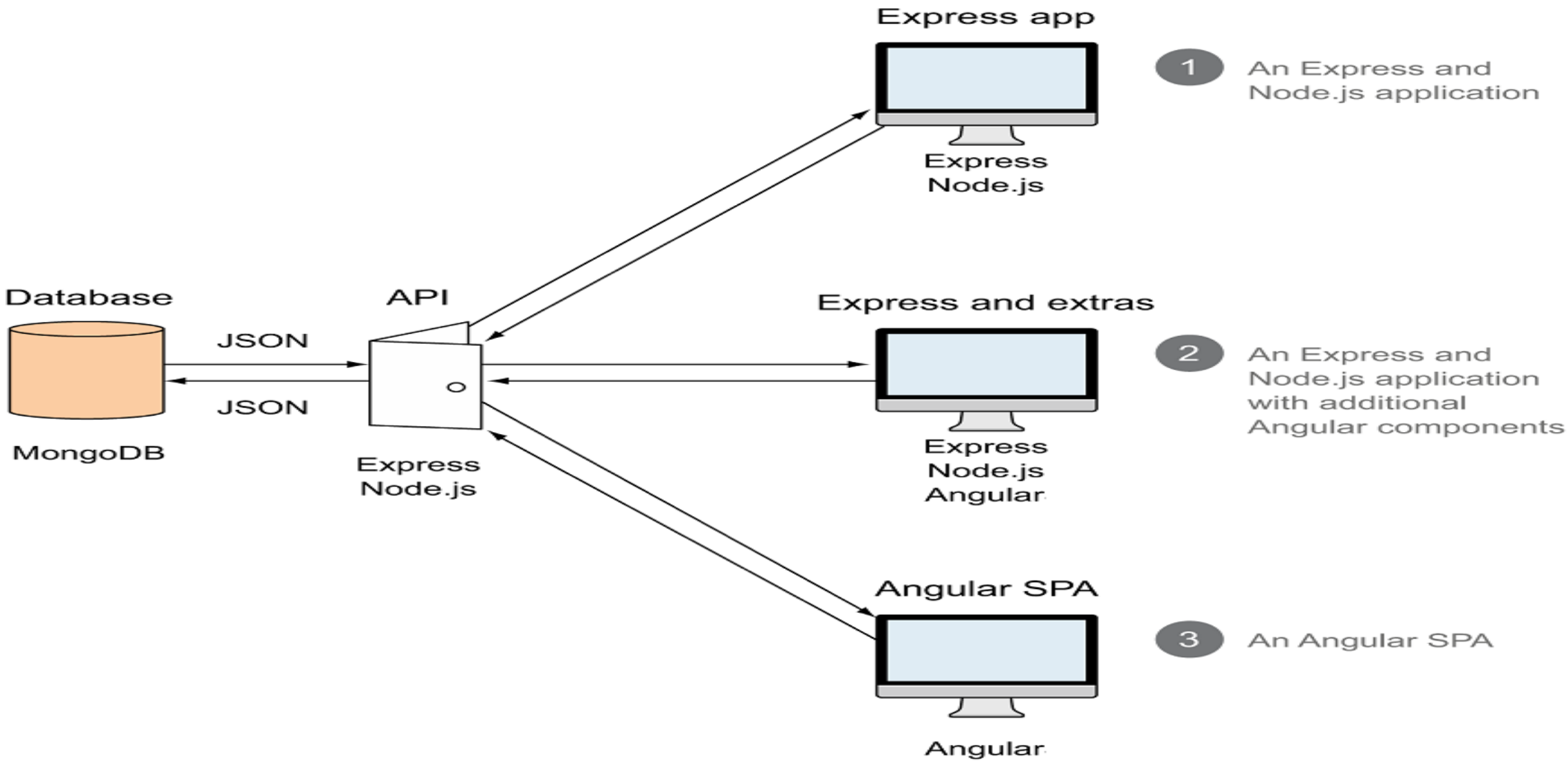
In order to access MongoDB database, we need to install MongoDB drivers. To install native mongodb drivers using NPM, open command prompt and write the following command to install MongoDB driver in your application.

npm install mongodb --save

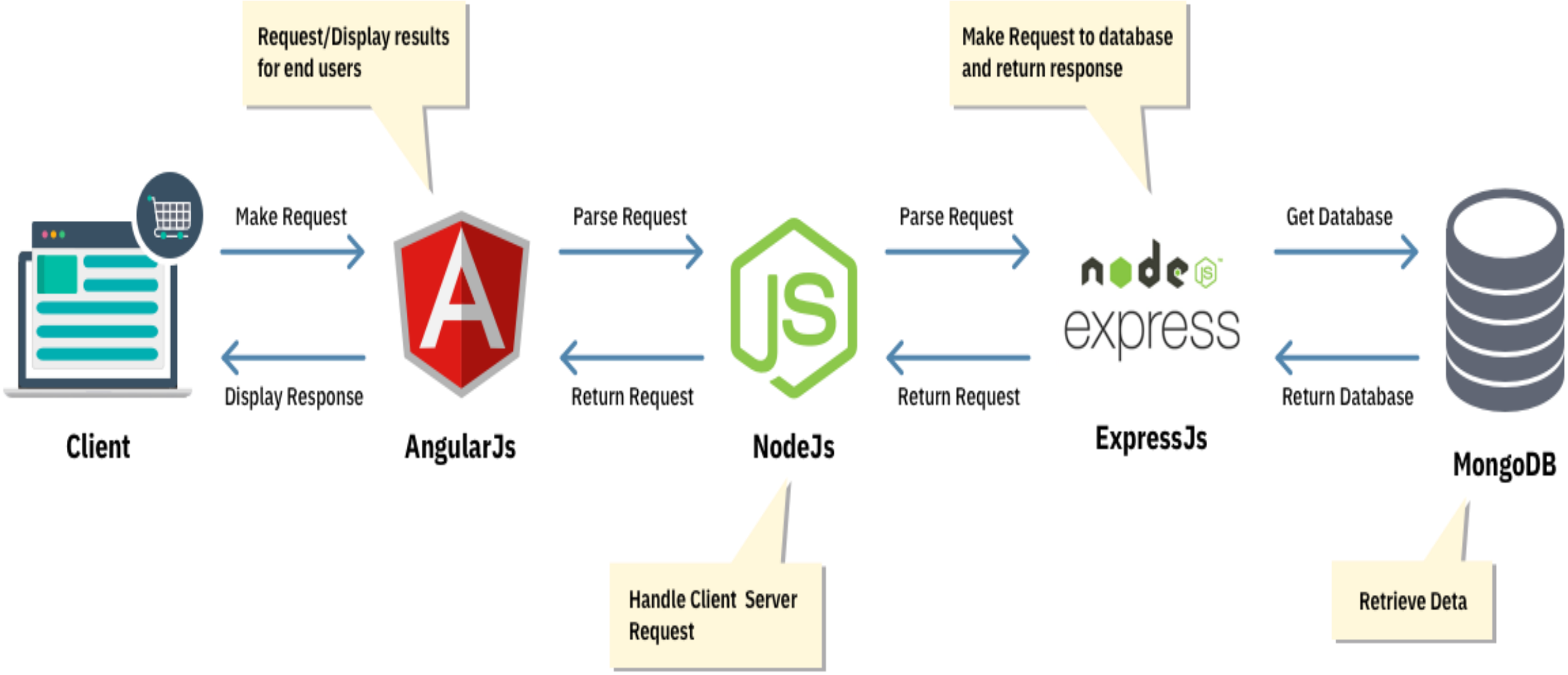
This will include mongodb folder inside node_modules folder. Now, start the MongoDB server using the following command. (Assuming that your MongoDB database is at C:\MyNodeJSConsoleApp\MyMongoDB folder.)

mongod -dbpath C:\MyNodeJSConsoleApp\MyMongoDB

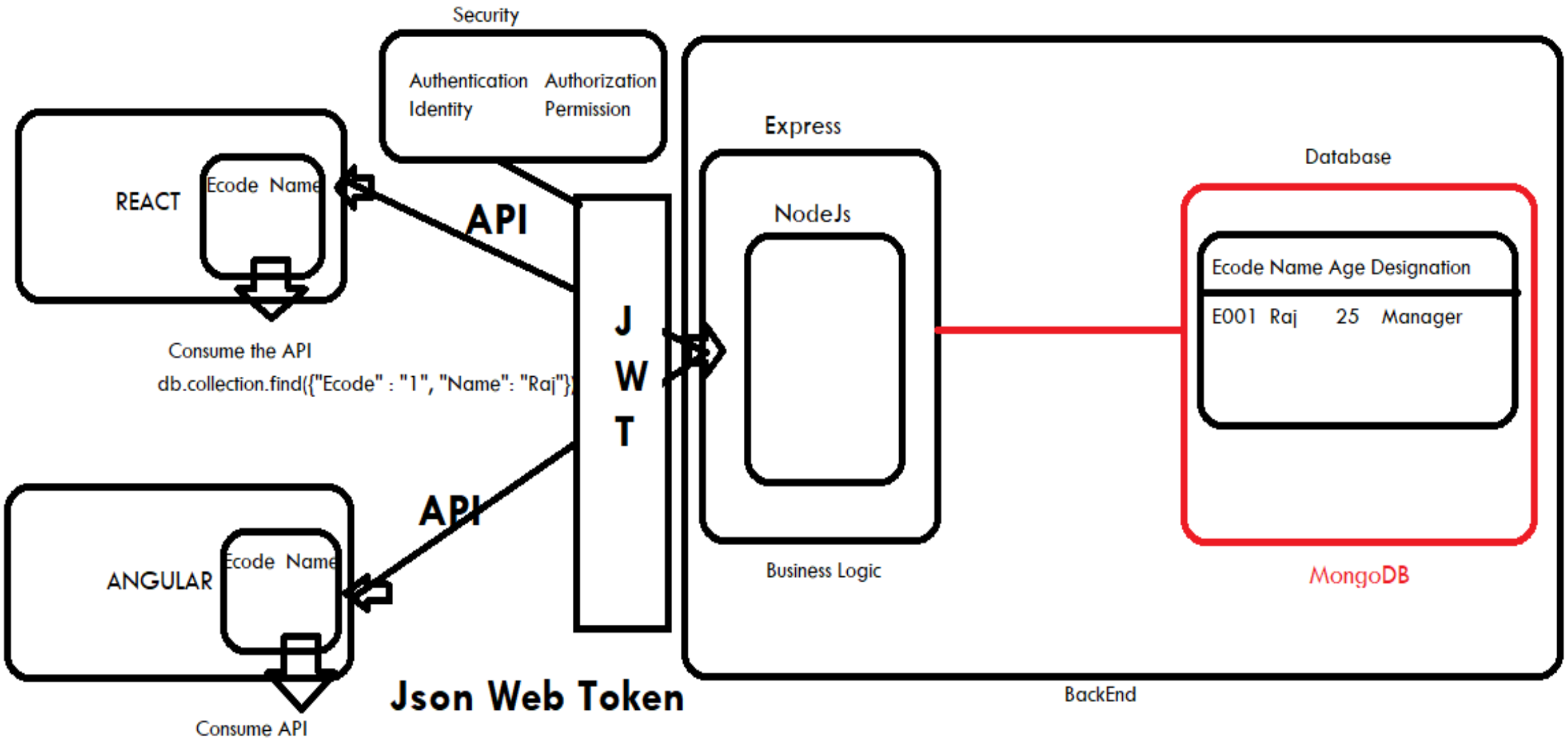
NodeJS with MEAN



NodeJS with MEAN



NodeJS with MEAN



NodeJS with MongoDB

```
var MongoClient = require('mongodb').MongoClient;

var url = 'mongodb://localhost/EmployeeDB';

MongoClient.connect(url, function(err, db) {

    console.log("Connected");

        db.close();
    });
```

1. using the mongodb driver
2. Specify the connection url
3. Connecting to the database
4. Writing to the console log
5. Closing the database connection

NodeJS with MongoDB

Using the find function to create a cursor of records

```
var MongoClient = require('mongodb').MongoClient;
var url = 'mongodb://localhost/EmployeeDB';

MongoClient.connect(url, function(err, db) {

  var cursor =db.collection('Employee').find( );

  cursor.each(function(err, doc) {

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

1

2

3

Printing the results to the console

For each record in the cursor we are calling a function

Node.js with MONGODB

Connecting MongoDB

The following example demonstrates connecting to the local MongoDB database.

app.js

```
var MongoClient = require('mongodb').MongoClient;
// Connect to the db
MongoClient.connect("mongodb://localhost:27017/MyDb", function (err, db) {
    if(err) throw err;
    //Write databse Insert/Update/Query code here..
});
```

Node.js with MONGODB

In the above example, we have imported mongodb module (native drivers) and got the reference of MongoClient object. Then we used MongoClient.connect() method to get the reference of specified MongoDB database. The specified URL "mongodb://localhost:27017/MyDb" points to your local MongoDB database created in MyMongoDB folder. The connect() method returns the database reference if the specified database is already exists, otherwise it creates a new database.

Now you can write insert/update or query the MongoDB database in the callback function of the connect() method using db parameter.

Node.js with MONGODB : Create Database & collection

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";
MongoClient.connect(url, {useNewUrlParser: true, useUnifiedTopology: true}, function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb27");
  dbo.createCollection("customers", function(err, res) {
    if (err) throw err;
    console.log("Collection created!");
    db.close();
  });
});
```

//Note : node app.js : If execution stuck uninstall the MongoDb version 5 npm uninstall mongodb & install the mongodb version 4 npm install mongodb@4

Node.js with MONGODB : Insert data into collection

insertdata.js

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url, {useNewUrlParser: true, useUnifiedTopology: true},
function(err, db) {
  if (err) throw err;
  var dbo = db.db("mydb27");
  var myobj = { name: “Raj Kumar", address: “Bangalore" };
  dbo.collection("customers").insertOne(myobj, function(err, res) {
    if (err) throw err;
    console.log("1 document inserted");
    db.close();
  });
});
```

Insert Documents

The following example demonstrates inserting documents into MongoDB database.

app.js

```
var MongoClient = require('mongodb').MongoClient;
// Connect to the db
MongoClient.connect("mongodb://127.0.0.1:27017/MyDb", function (err, db) {
    db.collection('Persons', function (err, collection) {
        collection.insert({ id: 1, firstName: 'Steve', lastName: 'Jobs' });
        collection.insert({ id: 2, firstName: 'Bill', lastName: 'Gates' });
        collection.insert({ id: 3, firstName: 'James', lastName: 'Bond' });
        db.collection('Persons').count(function (err, count) {
            if (err) throw err;
            console.log('Total Rows: ' + count);
        });
    });
});
```

Insert Documents

In the above example, db.collection() method creates or gets the reference of the specified collection. Collection is similar to table in relational database. We created a collection called Persons in the above example and insert three documents (rows) in it. After that, we display the count of total documents stored in the collection.

Running the above example displays the following result.

```
> node app.js
Total Rows: 3
```

Update/Delete Documents

The following example demonstrates updating or deleting an existing documents(records).

```
var MongoClient = require('mongodb').MongoClient;
// Connect to the db
MongoClient.connect("mongodb://127.0.0.1:27017/MyDb", function (err, db) {
   db.collection('Persons', function (err, collection) {
      collection.update({id: 1}, { $set: { firstName: 'James', lastName: 'Gosling'} }, {w:1},
                                    function(err, result){
                                          if(err) throw err;
                                          console.log('Document Updated Successfully');
                                      });
      collection.remove({id:2}, {w:1}, function(err, result) {
         if(err) throw err;
         console.log('Document Removed Successfully');
      });
   });
});
```

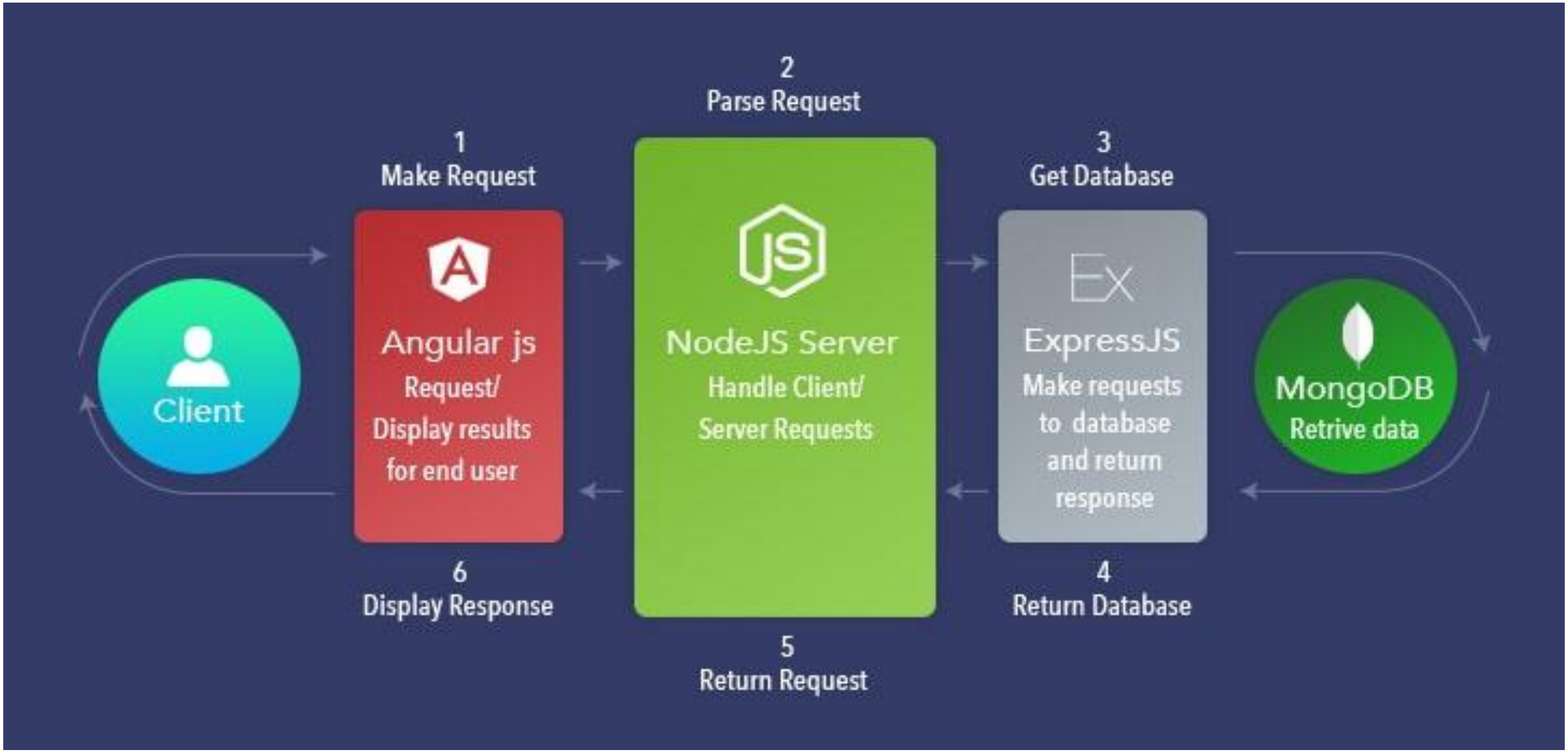
Query Database

The following example demonstrates executing a query in the MongoDB database.

app.js

```
var MongoClient = require('mongodb').MongoClient;
// Connect to the db
MongoClient.connect("mongodb://127.0.0.1:27017/MyDb", function (err, db) {
   db.collection('Persons', function (err, collection) {
       collection.find().toArray(function(err, items) {
         if(err) throw err;
         console.log(items);
      });
   });
});
```

NodeJS with MEAN



CONNECT Node with MySQL

CONNECT Node with MySQL

Connecting to MySQL
Before you do anything, you need to install the right NPM package.

$ **npm install mysql**
mysql is a great module which makes working with MySQL very easy and it provides all the capabilities you might need.

Once you have mysql installed, all you have to do to connect to your database is

CONNECT Node with MySQL

```
// Import module   npm install mysql
var mysql = require('mysql')

var connection = mysql.createConnection({
        host:"localhost",
        user:"root",
        password:"",
        database : “test"
})

// Connecting to database
connection.connect(function(err) {
        if(err){
        console.log("Error in the connection")
        console.log(err)
        }
        else{
        console.log(`Database Connected`)
        connection.query(`SHOW DATABASES`,
        function (err, result) {
                if(err)
                console.log(`Error executing the query - ${err}`)
                else
                console.log("Result: ",result)
        })
        }
})
```

Connect With MySQL: Insert Data

```
//npm install mysql
var mysql = require('mysql');

var con = mysql.createConnection({
   host: "localhost",
   user: "root",
   password: "",
   database: "test"
});
con.connect(function(err) {
  if (err) throw err;
  console.log("Connected!");
  var sql = "INSERT INTO testtable VALUES (103, 'Ritu', 'ritu@gmail.com','Developer')";
  con.query(sql, function (err, result) {
    if (err) throw err;
    console.log("1 record inserted");
  });
});
```

Connect With MySQL: Fetch Data

```
var mysql = require('mysql');
var con = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "",
  database: "test"
});

con.connect(function(err) {
  if (err) throw err;
  //Select all customers and return the result object:
  con.query("SELECT * FROM testtable", function (err, result, fields) {
    if (err) throw err;
    console.log(result);
  });
});
```

Connect With MySQL: Improvement

Steps:

1. Create config.js
2. Create stored procedure
3. Import/include config.js
4. Call stored Procedure

config.js

```
let config = {
   host    : 'localhost',
   user    : 'root',
   password: '',
   database: 'test'
 };
 module.exports = config;
```

Stored Procedure using Mysql

```
DELIMITER //
 CREATE PROCEDURE GetAllProducts()
  BEGIN
  SELECT *  FROM testtable;
  END //
 DELIMITER ;
```

storedproc.js

```
let mysql = require('mysql');  //npm  I mysql
let config = require('./config.js');

let connection = mysql.createConnection(config);

let sql = 'CALL GetAllProducts()';

connection.query(sql, true, (error, results, fields) => {
  if (error) {
    return console.error(error.message);
  }
  console.log(results[0]);
});

connection.end();
```

//Now run storedproc.js on command prompt : node storedproc.js

CONNECT PostgreSQL with NodeJs

CONNECT PostgreSQL with NodeJs

Steps :

1. Create package.json

   npm init // package.json will create

2. Inastall pg package

   npm install pg //node_modules folder will create automatically

3. Create app.js & put the logic

CONNECT PostgreSQL with NodeJs

```
const pgtools = require("pgtools");
const config = {
  user: "postgres",
  host: "localhost",
  password: "Test",
  port: 5432
};

pgtools.createdb(config, "myFirstDb", function(err, res) {
  if (err) {
    console.error(err);
   // process.exit(-1);
  }
  console.log(res);
});
```

Note :

The process.exit() method is used to end the process which is running at the same

Code: It can be either 0 or 1. 0 means end the process without any kind of failure

and 1 means end the process with some failure.

CONNECT PostgreSQL with NodeJs

app.js

```
const {Client}=require('pg');
const client =new Client(
  {
    host:"localhost",
    port: 5432,
    user:"postgres",
    password:"Test",
    database:"mydatabase10“
  });
  client.connect();
  client.query('select * from employee',(err,result)=>{
    if(!err)
    {
      console.log(result.rows);
    }
    client.end();
  })
```

CONNECT PostgreSQL with NodeJs using config

Task :

1. Create a seperate config.js for the database configuration & use it inside the show.js
2. create stored procedure showproc for getting all the data from employee table & call the stored procedure inside the show.js to fetch all the data of the employee table.

PostgreSQL with NodeJs with separate config.js

config.js

```
let config = {
   host    : "localhost",
   user    : "postgres",
   password: "Test",
   database: "mydatabase10"
 };
 module.exports = config;
```

show.js

```
let config = require("./config.js");
const {Client}=require('pg');
const client =new Client(config);
   client.connect();
   client.query('select * from employee',(err,result)=>{
      if(!err)
      {
         console.log(result.rows);
      }
      client.end();
   })

// run show.js : node show.js
```

PostgreSQL with NodeJs with separate config.js

config.js

```
let config = {
   host    : "localhost",
   user    : "postgres",
   password: "Test",
   database: "mydatabase10"
 };
 module.exports = config;
```

```
CREATE OR REPLACE FUNCTION ShowProc()
RETURNS SETOF employee
LANGUAGE SQL
AS $$
SELECT * FROM employee;
$$;
```

show.js

```
let config = require("./config.js");
const {Client}=require('pg');
const client =new Client(config);
   client.connect();
  // client.query('select * from employee',(err,result)=>{
 client.query('select ShowProc()',(err,result)=>{
       if(!err)
       {
          console.log(result.rows);
       }
       client.end();
   })

// run show.js : node show.js
```

CONNECT PostgreSQL with NodeJs & Express

Steps :

1. Create package.json

   npm init // package.json will create

2. Inastall pg package

   npm install pg //node_modules folder will create automatically

3. Install express package // connect your output with browser

4. Create app.js & put the logic

PostgreSQL with NodeJs & Express & Execute the output on browser

App.js

```
const express = require('express');
const { Client } = require('pg');
const connectionString = 'postgres://postgres:Test@localhost:5432/mydatabase10';
//"postgres://YourUserName:YourPassword@localhost:5432/YourDatabase";
const client = new Client({
   connectionString: connectionString
});
client.connect();
var app = express();
app.set('port', process.env.PORT || 4000);
```

PostgreSQL with NodeJs & Express & Execute the output on browser

```
app.get('/', function (req, res, next) {
   //client.query('SELECT * FROM Employee where id = $1', [1], function (err, result) {
client.query('SELECT * FROM Employee', function (err, result) {
       if (err) {
           console.log(err);
           res.status(400).send(err);
       }
       res.status(200).send(result.rows);
   });
});
app.listen(4000, function () {
   console.log('Server is running.. on Port 4000');
});
```

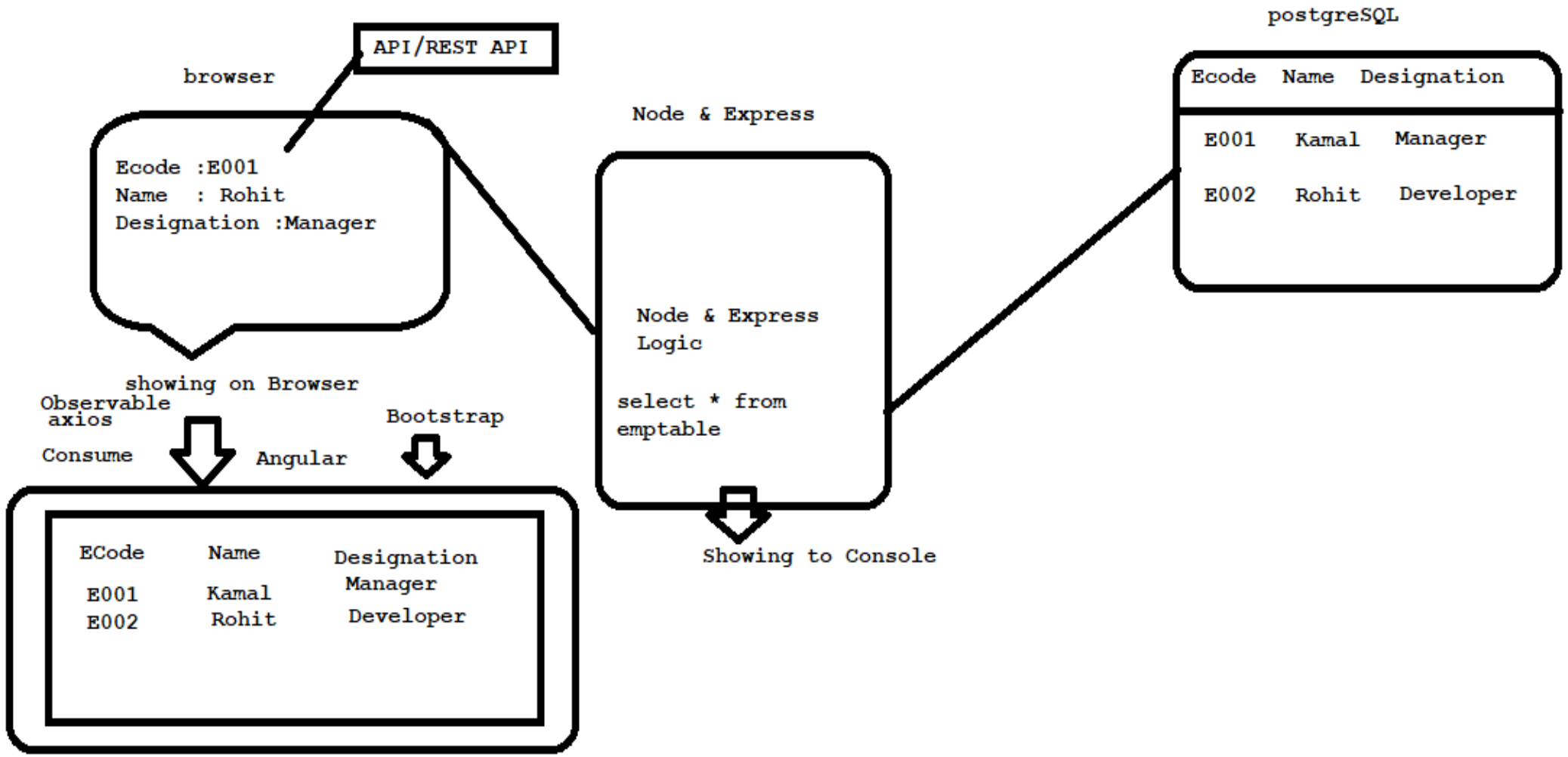
Using AXIOS to get the data from PostgreSQL as REST API created using NodeJs & Express

```
const axios = require('axios')

// Make request
axios.get('http://localhost:4000')  //API created using node & express

// Show response data
.then(res => console.log(res.data))
.catch(err => console.log(err))
```

request module to get the data from PostgreSQL as REST API created using NodeJs & Express

```
// npm install request
const request = require('request')

// Request URL

var url = 'http://localhost:4000/';  // API created by Node & Express

request(url, (error, response, body)=>{

   // Printing the error if occurred

   if(error) console.log(error)


   // Printing status code

   console.log(response.statusCode);


   // Printing body

   console.log(body);

});
```

CASE STUDY


postgreSQL
API/REST API
browser
Ecode :E001
Name : Rohit
Designation :Manager
Node & Express
Ecode Name Designation
E001 Kamal Manager
E002 Rohit Developer
Node & Express
Logic
select * from
emptable
showing on Browser
Observable
axios
Bootstrap
Consume
Angular
Showing to Console
ECode Name Designation
E001 Kamal Manager
E002 Rohit Developer

CASE STUDY : Complete step by step

STEPS TO CONSUME THE REST API CREATED UISNG NODE & EXPRESS

1. You will install the Angular CLI

 npm install –g @angular/cli

2. creat The Angular Project

   ng new my-project

3. npm start

5. create a service

6. put the logic to consume the service

8. Decorate the output inside the app.component.html

9. Create app.js using node & express . Put the logic to show it on browser & Bind the CORS as middleware

10. Create the database using PostgreSQL & put the data inside the employee table

CREATING API WITH NODE, EXPRESS & POSTGRESQL

STEPS TO ENABLE THE CORS -Mandatory

1. install cors as module - npm i cors

2. Bind it with app.js as middleware

app.js

```
const express = require('express');
var cors = require('cors'); //npm i cors
const { Client } = require('pg');
const connectionString = 'postgres://postgres:Test@localhost:5432/mydatabase10';
//"postgres://YourUserName:YourPassword@localhost:5432/YourDatabase";
const client = new Client({
   connectionString: connectionString
});
client.connect();
var app = express();
app.use(cors());
app.set('port', process.env.PORT || 5000);
app.get('/', function (req, res, next) {
```

CREATING API WITH NODE, EXPRESS & POSTGRESQL

```
//client.query('SELECT * FROM Employee where id = $1', [1], function (err, result) {
client.query('SELECT * FROM Employee', function (err, result) {
        if (err) {
            console.log(err);
            res.status(400).send(err);
        }
        res.status(200).send(result.rows);
    });
});
app.listen(5000, function () {
    console.log('Server is running.. on Port 5000');
});
```

Creating API with Node , Express & MySQL

STEPS TO ENABLE THE CORS -Mandatory

1. install cors as module - npm i cors
2. Bind it with app.js as middleware
3. npm install mysql
4. npm install express
5. npm I nodemon

Creating API with NODE, Express & MySQL

app.js

```
const express = require("express");
var cors = require("cors"); //npm i cors
const mysql = require("mysql");
const connection = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "",
  database: "mydatabase10"
});
connection.connect();
var app = express();
app.use(cors());
```

Creating API with NODE, Express & MySQL

```
app.set("port", process.env.PORT || 5000);
app.get("/", function (req, res, next) {
  connection.query("SELECT * FROM Employee", function (err, result) {
    if (err) {
      console.log(err);
      res.status(400).send(err);
    }
    res.status(200).send(result);
  });
});
app.listen(5000, function () {
  console.log("Server is running.. on Port 5000");
});
```

Creating API with Node , Express & MongoDB

STEPS TO ENABLE THE CORS -Mandatory

1. install cors as module - npm i cors
2. Bind it with app.js as middleware
3. npm install mongodb
4. npm install express
5. npm I nodemon

Creating API with Node , Express & MongoDB

App.js

```
const express = require('express');
const MongoClient = require('mongodb').MongoClient;
const app = express();
const port = 4000;
const url = 'mongodb://localhost:27017/’;
const dbName = 'mydatabase10';
const client = new MongoClient(url, { useUnifiedTopology: true });
app.get('/employees', async (req, res) => {
  try {
    await client.connect();
    const db = client.db(dbName);
    const collection = db.collection('employee');
```

Creating API with Node , Express & MongoDB

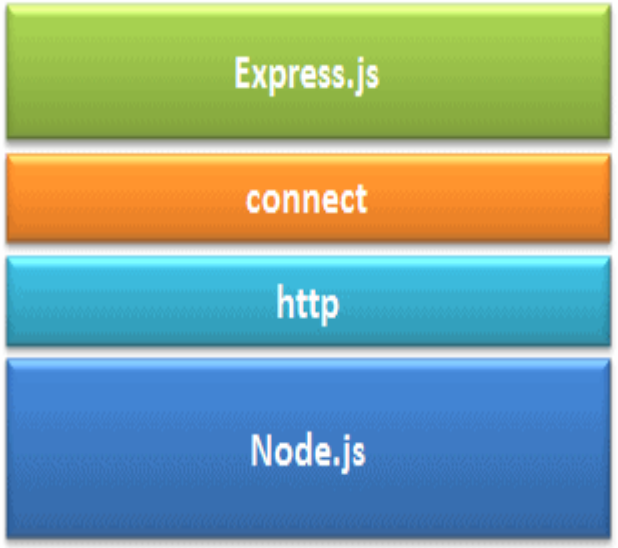
```
// Find all documents in the collection
    const employees = await collection.find({}).toArray();
    // Respond with the list of employee documents in JSON format
    res.json(employees);
  } catch (e) {
    res.status(500).send('Error connecting to the database');
  } finally {
    // Ensures that the client will close when you finish/error
    await client.close();
  }
});
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

EXPRESS

- Web application framework for Node.js
- Light-weight and minimalist
- Provides boilerplate structure & organization for your web-apps

EXPRESS

- 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.



Advantages of Express.js

- Makes Node.js web application development fast and easy.
- Easy to configure and customize.
- Allows you to define routes of your application based on HTTP methods and URLs.
- Includes various middleware modules which you can use to perform additional tasks on request and response.
- Easy to integrate with different template engines like Jade, Vash, EJS etc.
- Allows you to define an error handling middleware.
- Easy to serve static files and resources of your application.
- Allows you to create REST API server.
- Easy to connect with databases such as MongoDB, Redis, MySQL

EXPRESS

Database Integration

Database systems supported by Express:

- Cassandra
- Couchbase
- CouchDB
- LevelDB
- MySQL
- MongoDB
- Neo4j
- PostgreSQL
- Redis
- SQL Server
- SQLite
- ElasticSearch

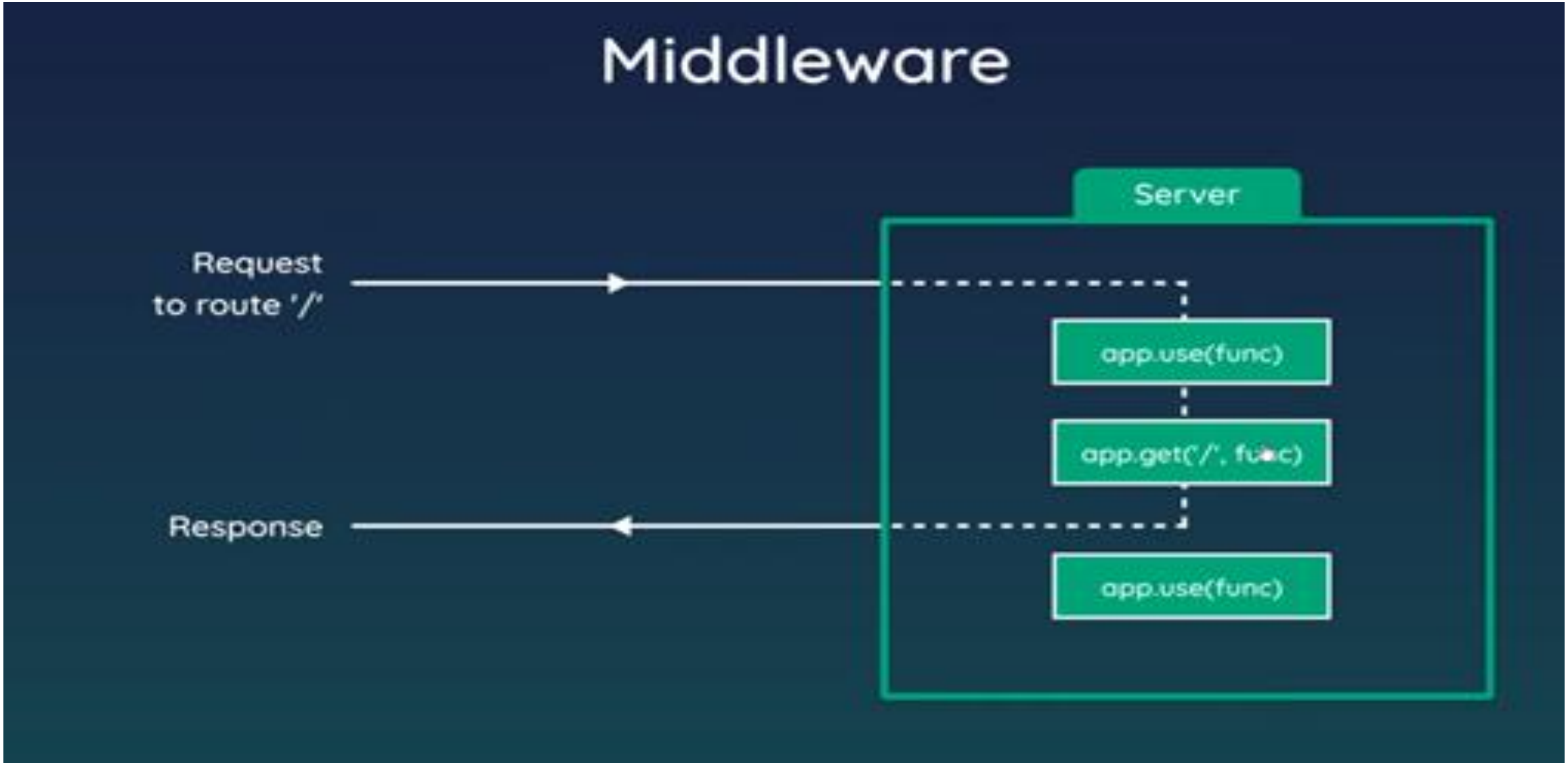
Middleware with Express

- ❑ Express is a Node module that gives us middleware. Execution flows through middleware, and a response drops out the bottom.
- ❑ Express is a node module that which provides us with middleware. Middleware is like a pipeline. Requests are passed through each middleware function in turn top to bottom. Each middleware function receives the request and response objects, and can modify them.

Middleware functions can do things like:

- Log a request
- Check authorization
- Serve a static file
- Inspect the URL, parse out URL parameters, and save them in the request
- Compile a SASS file and serve the result as CSS
- Serve a web page
- Serve an error or 404

NodeJS Important Concepts



NodeJS Important Concepts

Middleware Examples

- Logger middleware to log details of every request
- Authentication check middleware for protected routes
- Middleware to parse JSON data from requests
- Return 404 pages

EXPRESS

Middleware

```
1   var express = require('express')
2   var app = express()
3
4   var requestTime = function (req, res, next) {
5     req.requestTime = Date.now()
6     next()
7   }
8
9   app.use(requestTime)
10
11  app.get('/', function (req, res) {
12    var responseText = 'Hello World!<br>'
13    responseText += '<small>Requested at: ' + req.requestTime + '</small>'
14    res.send(responseText)
15  })
16
17  app.listen(3000);
```

Uses the *requestTime* middleware function

EXPRESS

Router-level middleware

- Router-level middleware binds to an instance of express.Router()
- Works in the same way as application-level middleware

```
var app = express()
var router = express.Router()

// a middleware function with no mount path. This code is executed for every request to the router
router.use(function (req, res, next) {
  console.log('Time:', Date.now())
  next()
})

// a middleware sub-stack shows request info for any type of HTTP request to the /user/:id path
router.use('/user/:id', function (req, res, next) {
  console.log('Request URL:', req.originalUrl)
  next()
}, function (req, res, next) {
  console.log('Request Type:', req.method)
  next()
})
```

Router middleware i.e. executed for every router request

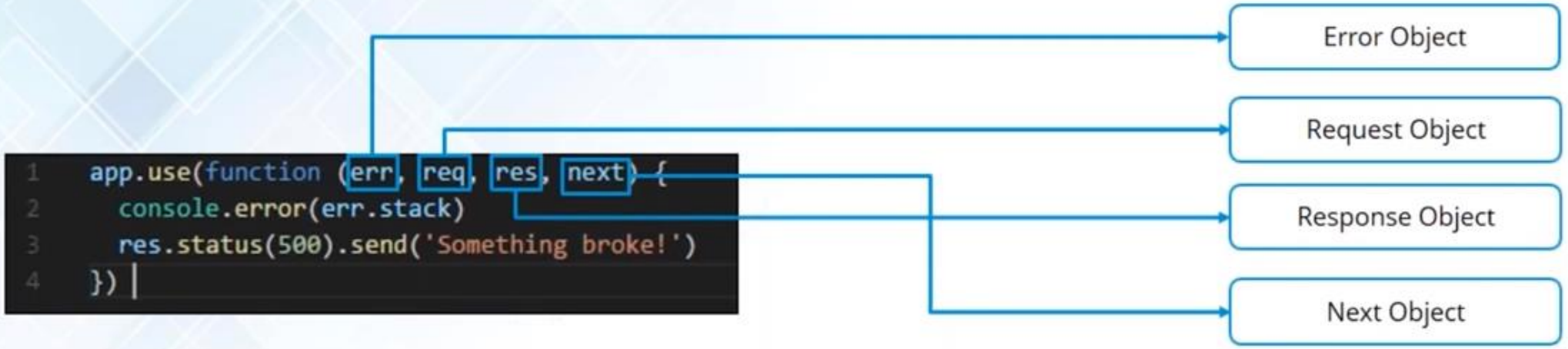
Router middleware i.e. executed for given path

EXPRESS

Error-handling middleware

- Error-handling middleware always takes ***four*** arguments: *(err, req, res, next)*
- next object will be interpreted as regular middleware and will fail to handle errors
- Define error-handling middleware functions in the same way as other middleware functions

```
1  app.use(function (err, req, res, next) {
2    console.error(err.stack)
3    res.status(500).send('Something broke!')
4  })
```



Middleware with Express

Install express locally using NPM

Create a directory to hold your express app. We'll use the NPM generator to make a package file. This will define our dependencies. Run the following:

npm init

Now install the express dependency:

npm install express --save

Check out your directory. You have gained an node_modules directory that contains express and all it's dependencies. Now check out your package.json file. The express dependency has been saved in it.

You can now run:

npm install

Hello world example

```
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening at http://localhost:${port}`)
})
```

Hello world example

- This app starts a server and listens on port 3000 for connections. The app responds with “Hello World!” for requests to the root URL (/) or route. For every other path, it will respond with a 404 Not Found.

- The example above is actually a working server: Go ahead and click on the URL shown. You’ll get a response, with real-time logs on the page, and any changes you make will be reflected in real time.
- This is powered by RunKit, which provides an interactive JavaScript playground connected to a complete Node environment that runs in your web browser. Below are instructions for running the same app on your local machine.

Create Server in Express

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

Web Server

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

app.js: Express.js Web Server Copy

```
var express = require('express');
var app = express();
// define routes here..

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

Create Server in Express

- 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.

Creating Server In Express

We create a simple server in Express by wiring together middleware functions.

```
var express = require('express');
var http = require('http');
var app = express();
// create a route
app.get('/', function(req, res){
  res.writeHead(200);
  res.write('Hello Express!!!');
  res.end();
});
// Create a server
http.createServer(app).listen(3000);
```

EXPRESS

Routes

Routing refers to the definition of application end points (URIs) and how they respond to client requests

A ***route method*** is derived from one of the HTTP methods, and is attached to an instance of the express class

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

var app = express();

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

    res.send('GET request to the homepage');

});

//POST method route
app.post('/', function(req, res){

    res.send('POST request to the homepage');

});
```

Importing Express Module

Creating Express Instance

Callback function

Path (route)

Route Methods

HTTP request

HTTP response

EXPRESS

Route Handlers

Provide multiple callback functions which behave like middleware to handle a request

Exception -> Callbacks might invoke next('route') to bypass the remaining route callbacks

```
app.get('/example/a', function (req, res) {
  res.send('Hello from A!')
})
```

A single callback function can handle a route

```
app.get('/example/b', function (req, res, next) {
  console.log('the response will be sent by the next function ...')
  next()
}, function (req, res) {
  res.send('Hello from B!')
})
```

Next to bypass the remaining route callbacks

Used to impose pre-conditions on a route, then pass control to subsequent routes (if no reason to proceed with the current route)

EXPRESS

Routes (app.route)

- Create chainable route handlers for a route path by using *app.route()*
- Path is specified at a single location
- Creating modular routes is helpful, as it reduces redundancy and typos

Creating chainable route

GET Method

POST Method

PUT Method

```
1  app.route('/book')
2    .get(function (req, res) {
3      res.send('Get a random book')
4    })
5    .post(function (req, res) {
6      res.send('Add a book')
7    })
8    .put(function (req, res) {
9      res.send('Update the book')
10   })
```

EXPRESS

Routes (express.Router)

```
var express = require('express')
var router = express.Router()

// middleware that is specific to this router
router.use(function timeLog (req, res, next) {
  console.log('Time: ', Date.now())
  next()
})
// define the home page route
router.get('/', function (req, res) {
  res.send('Birds home page')
})
// define the about route
router.get('/about', function (req, res) {
  res.send('About birds')
})

module.exports = router
```

Creates a router as a module

Loads a middleware function

Defines Home Route

Define About Route

- Router class to create modular, mountable route handlers
- A Router instance is a complete middleware and routing system

EXPRESS

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.

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.

Handle POST Request

```
<!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>
```

Handle POST Request

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
```

Handle POST Request

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

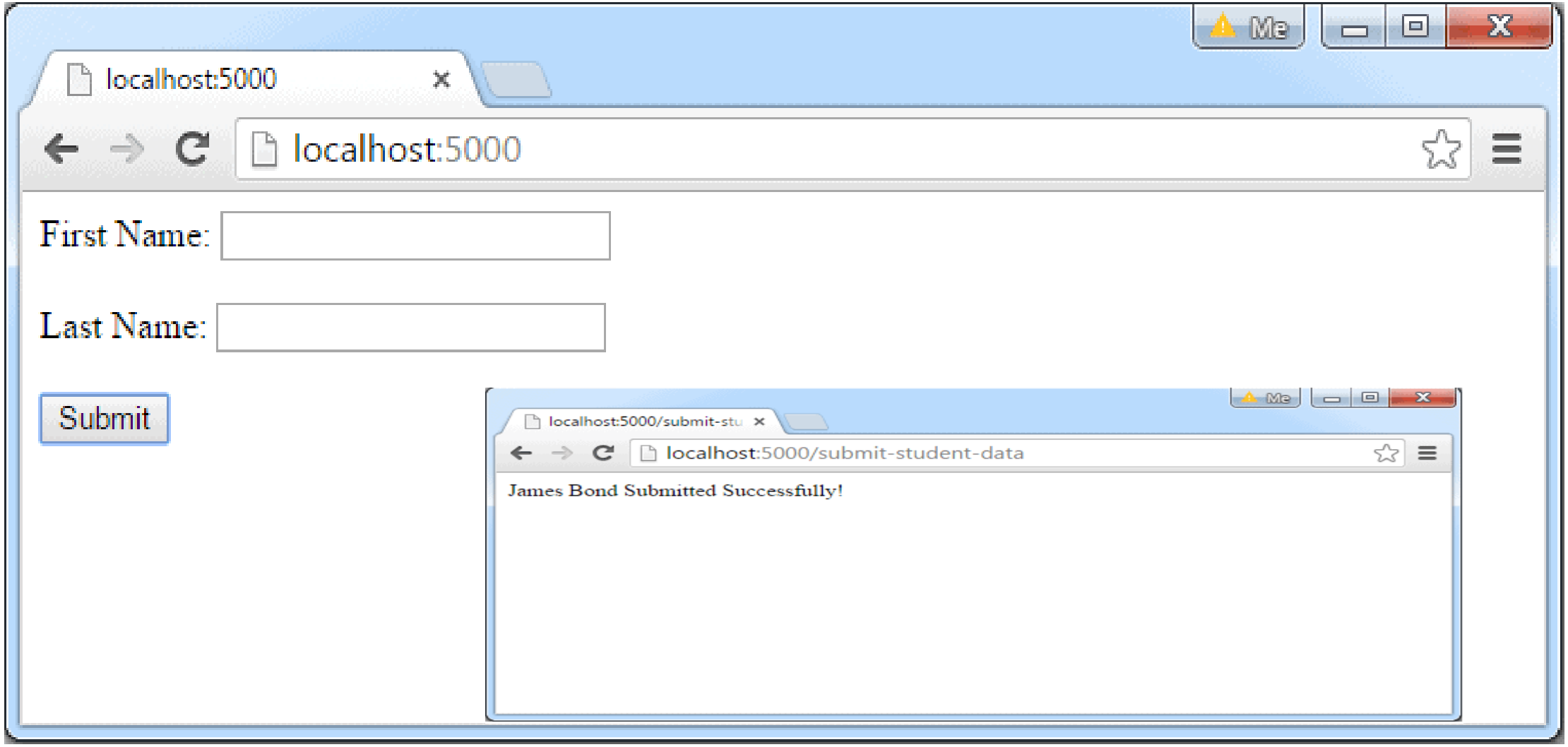
```
var express = require('express');
var app = express();
var bodyParser = require("body-parser");
app.use(bodyParser.urlencoded({ extended: false }));
app.get('/', function (req, res) {
   res.sendFile('index.html');
});
app.post('/submit-student-data', function (req, res) {
   var name = req.body.firstName + ' ' + req.body.lastName;

   res.send(name + ' Submitted Successfully!');
});
var server = app.listen(5000, function () {
   console.log('Node server is running..');
});
```

Handle POST Request

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.

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

Handle POST Request



ROUTE EXAMPLE

```
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..');
});
```

ROUTE EXAMPLE

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.