Node.js Modules and HTML Form Handling

What is a Module

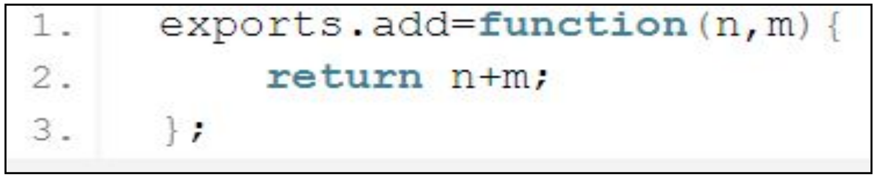
- The <u>module</u> is a small self-contained piece of code. You can <u>include</u> module using the “require” statement.
- A module encapsulates related code into a single unit of code. When creating a module, this can be interpreted as moving all related functions into a file.
- There are three types of modules:
  1) Core Modules
  2) Local Modules
  3) Third-party Modules
- You can use file and folder both as the module.

Types of Modules in Nodejs

- **Core modules** Built-in modules of node.js that are part of nodejs and come with the Node.js installation process. To load/include this module in our program, we use he **require** function.

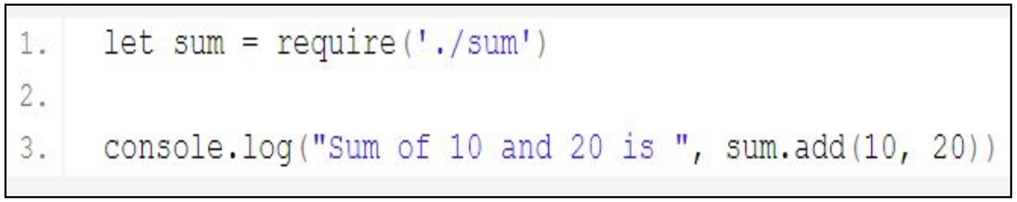
  let module = require('module_name')

- **Local modules** are created by us locally in our Node.js application. These modules are included in our program in the same way as we include the built in module. **Exports** keyword is used to make properties and methods available outside the file.



```
    exports.add=function(n,m){
        return n+m;
    };
```

Creating local modules and exporting (sum.js)



```
let sum = require('./sum')

console.log("Sum of 10 and 20 is ", sum.add(10, 20))
```

Importing local modules (calsum.js)
You can run the module using the **node calsum.js**

- **Third party modules** are available online and are installed using the **npm**. Example: express, mongoose, etc.
- You can load the modules from the "node_modules" folder.

exports vs module.exports

- exports and module.exports reference the same object.
- But methods will be available as public when we use module.exports.
- If a developer unintentionally or intentionally re-assigns module.exports to a different object or different data structure, then any properties added to the original module.exports object will be unaccessible.

Example

```
exports.name = function() {
    console.log('My name is Lemmy Kilmister');
};
```

rocker.js

```
var rocker = require('./rocker.js');
rocker.name(); // 'My name is Lemmy Kilmister'
```

main.js

- Your module returns module.exports to the caller ultimately, not exports. All exports does is collect properties and attach them to module.exports if module.exports doesn't have something on it already. If there's something attached to module.exports already, everything on exports is ignored. Put the following in rocker.js

```
module.exports = 'ROCK IT!';
exports.name = function() {
    console.log('My name is Lemmy Kilmister');
};
```

Both of them are references to the same (empty) object at the beggining. (But only module.exports will be returned!)

And this in another file, and run it:

```
var rocker = require('./rocker.js');
rocker.name(); // TypeError: Object ROCK IT! has no method 'name'
```

Example-1

- misc.js

```
var x = 5;
var addX = function(value) {
  return value + x;
};
module.exports.x = x;
module.exports.addX = addX;
```

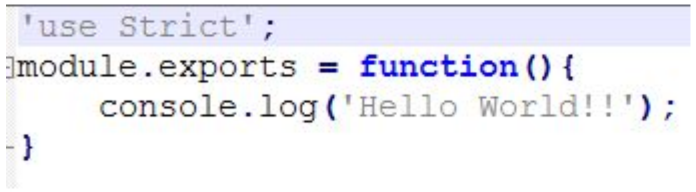
- misc_main.js

```
var misc = require('./misc');
console.log("Adding %d to 10 gives us %d", misc.x, misc.addX(10));
```

Example-2

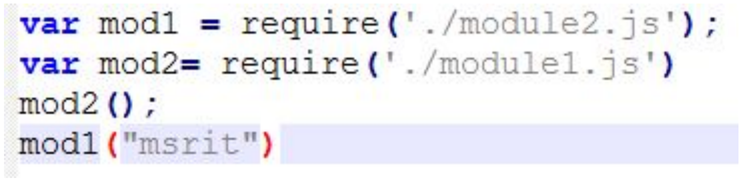
- Create a folder **testnodejs**
- **Create three files module1.js,module2.js and index.js**

module1.js



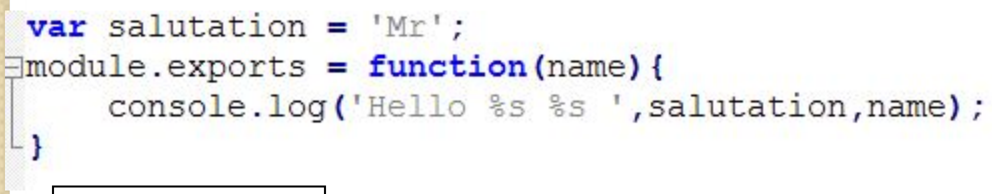
```
'use Strict';
module.exports = function(){
    console.log('Hello World!!');
}
```

index.js



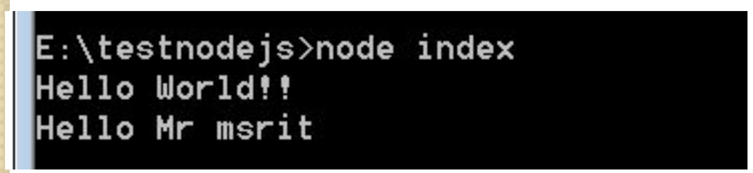
```
var mod1 = require('./module2.js');
var mod2= require('./module1.js')
mod2();
mod1("msrit")
```

module2.js



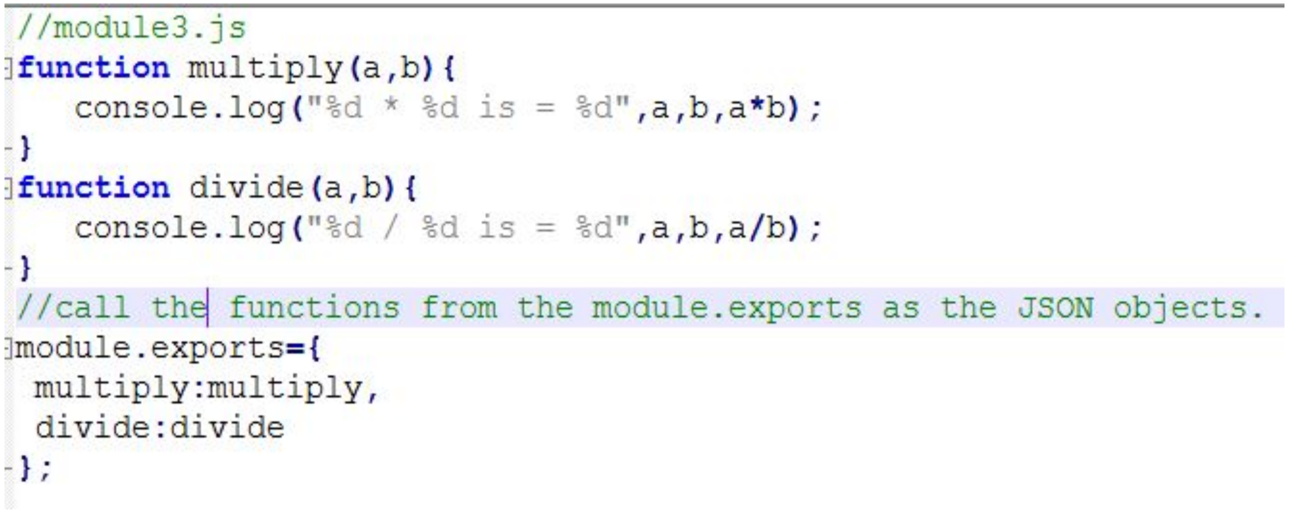
```
var salutation = 'Mr';
module.exports = function(name){
    console.log('Hello %s %s ',salutation,name);
}
```

Output is



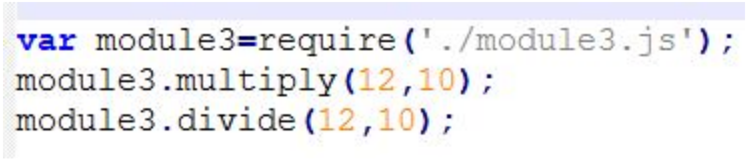
```
E:\testnodejs>node index
Hello World!!
Hello Mr msrit
```

Another module to export Example-3



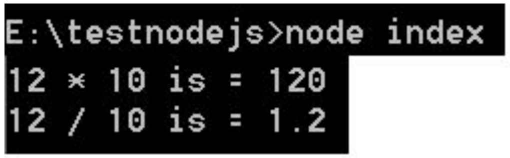
```
//module3.js
function multiply(a,b){
   console.log("%d * %d is = %d",a,b,a*b);
}
function divide(a,b){
   console.log("%d / %d is = %d",a,b,a/b);
}
//call the functions from the module.exports as the JSON objects.
module.exports={
 multiply:multiply,
 divide:divide
};
```

module3.js



```
var module3=require('./module3.js');
module3.multiply(12,10);
module3.divide(12,10);
```

index.js



```
E:\testnodejs>node index
12 * 10 is = 120
12 / 10 is = 1.2
```

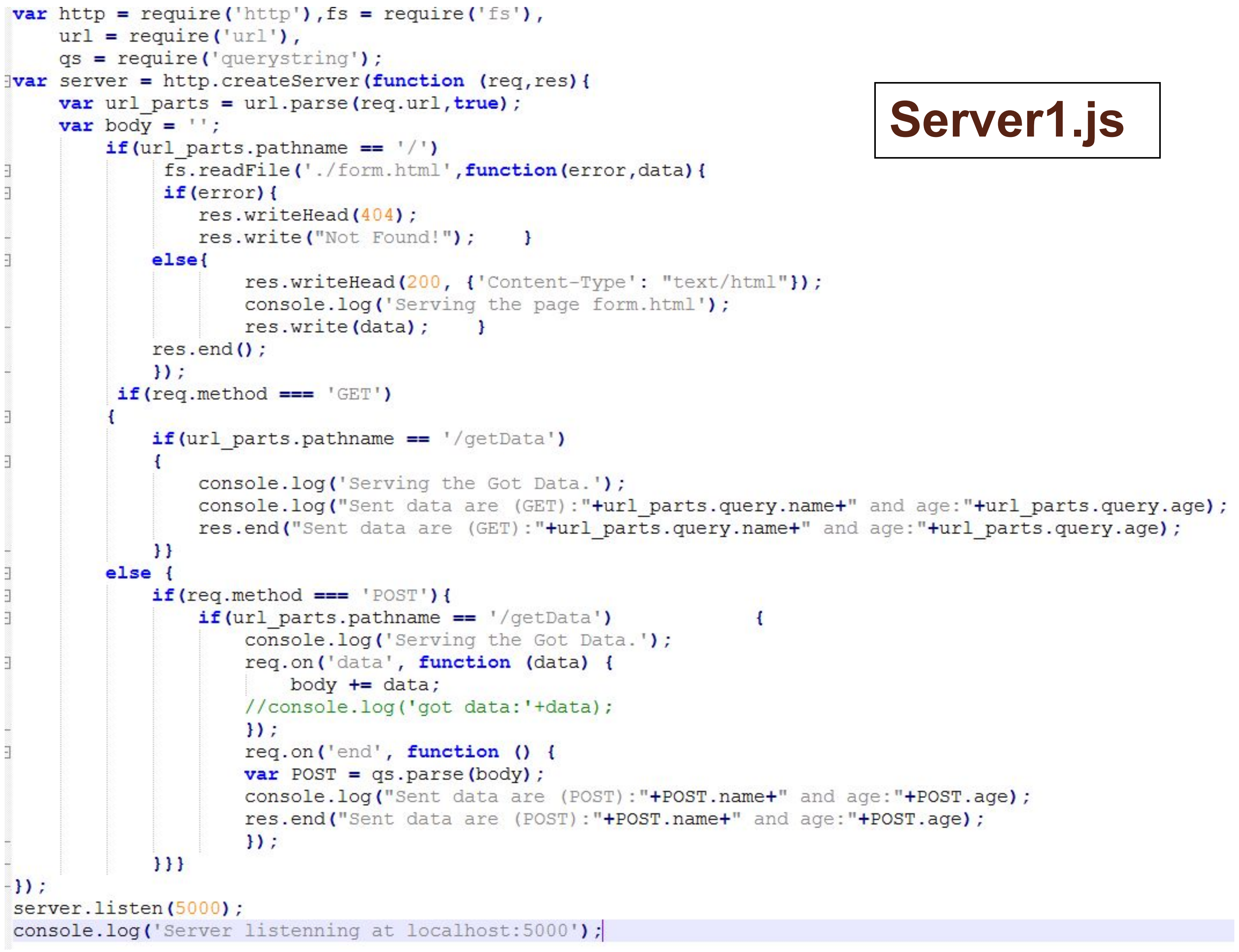
Output

Creating web application using Node.js

- Creating a web application (to be more precise, a web server) in Node.js, you write a single JavaScript function for your entire application. This function listens to a web browser's requests, or the requests from a mobile application consuming your API, or any other client talking to your server. When a request comes in, this function will look at the request and determine how to respond. If you visit the homepage in a web browser, for example, this function could determine that you want the homepage and it will send back some HTML. If you send a message to an API endpoint, this function could determine what you want and respond with JSON.

Creating an HTTP Server With a Form in Node.js

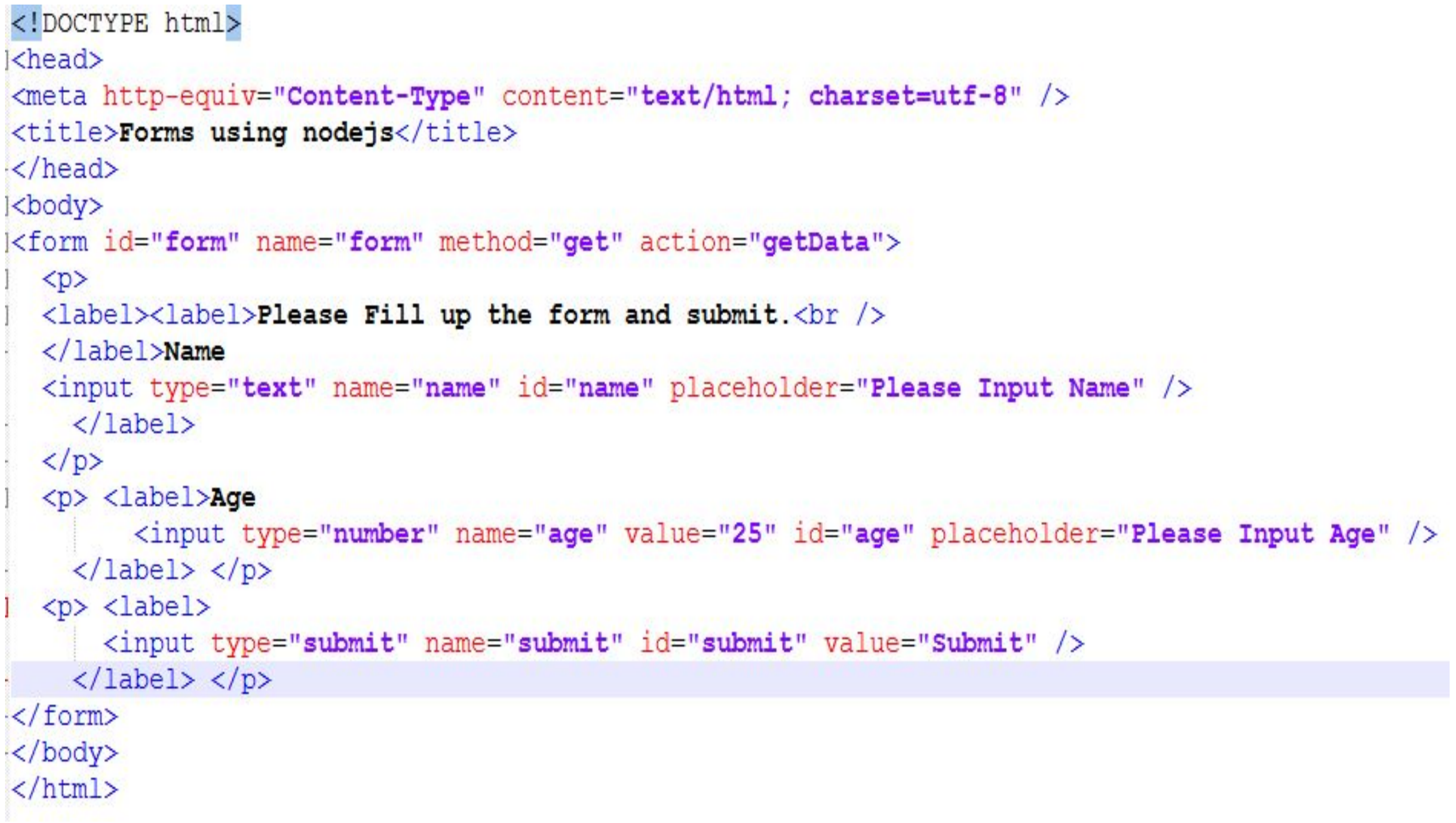
Server1.js



```
var http = require('http'),fs = require('fs'),
    url = require('url'),
    qs = require('querystring');
var server = http.createServer(function (req,res){
    var url_parts = url.parse(req.url,true);
    var body = '';
        if(url_parts.pathname == '/')
            fs.readFile('./form.html',function(error,data){
            if(error){
               res.writeHead(404);
               res.write("Not Found!");    }
           else{
                   res.writeHead(200, {'Content-Type': "text/html"});
                   console.log('Serving the page form.html');
                   res.write(data);    }
           res.end();
           });
         if(req.method === 'GET')
        {
            if(url_parts.pathname == '/getData')
            {
                console.log('Serving the Got Data.');
                console.log("Sent data are (GET):"+url_parts.query.name+" and age:"+url_parts.query.age);
                res.end("Sent data are (GET):"+url_parts.query.name+" and age:"+url_parts.query.age);
            }}
        else {
            if(req.method === 'POST'){
                if(url_parts.pathname == '/getData')            {
                    console.log('Serving the Got Data.');
                    req.on('data', function (data) {
                        body += data;
                    //console.log('got data:'+data);
                    });
                    req.on('end', function () {
                    var POST = qs.parse(body);
                    console.log("Sent data are (POST):"+POST.name+" and age:"+POST.age);
                    res.end("Sent data are (POST):"+POST.name+" and age:"+POST.age);
                    });
            }}}
});
server.listen(5000);
console.log('Server listenning at localhost:5000');
```

form.html

- We'll also need to create a form.html file which will contain the HTML code to render the form as shown below:



```
<!DOCTYPE html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Forms using nodejs</title>
</head>
<body>
<form id="form" name="form" method="get" action="getData">
  <p>
  <label><label>Please Fill up the form and submit.<br />
  </label>Name
  <input type="text" name="name" id="name" placeholder="Please Input Name" />
    </label>
  </p>
  <p> <label>Age
        <input type="number" name="age" value="25" id="age" placeholder="Please Input Age" />
    </label> </p>
  <p> <label>
      <input type="submit" name="submit" id="submit" value="Submit" />
    </label> </p>
</form>
</body>
</html>
```

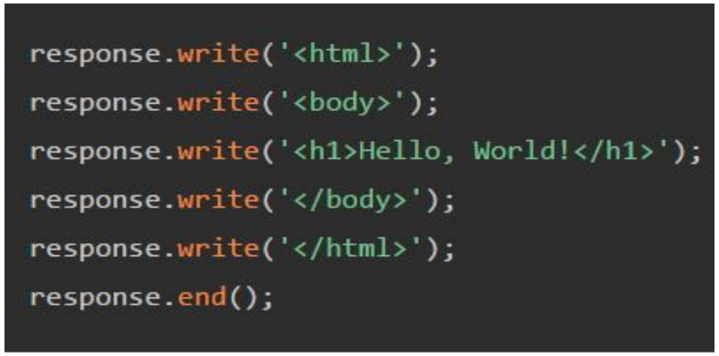
Anatomy of an HTTP Transaction

- Any node web server application will at some point have to create a web server object. This is done by using createServerAny node web server application will at some point have to create a web server object. This is done by using createServer. The function that's passed in to createServer is called once for every HTTP request that's made against that server, so it's called the request handler.
- When an HTTP request hits the server, node calls the request handler function with a few handy objects for dealing with the transaction, request and response.
- In order to actually serve requests, the listen method needs to be called on the serverobject. In most cases, all you'll need to pass to listen is the port number you want the server to listen on.
- Method, URL and Headers
  - The method here will always be a normal HTTP method/verb. The url is the full URL without the server, protocol or port. For a typical URL, this means everything after and including the third forward slash. All headers are represented in lower-case only, regardless of how the client actually sent them. This simplifies the task of parsing headers for whatever purpose.

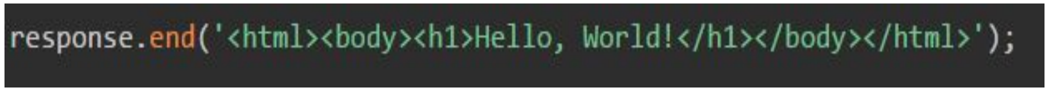
Request Body

- When receiving a POST or PUT request, the request body might be important to your application. Getting at the body data is a little more involved than accessing request headers. The request object that's passed in to a handler implements the ReadableStream interface. This stream can be listened to or piped elsewhere just like any other stream. We can grab the data right out of the stream by listening to the stream's 'data' and 'end' events.
- The chunk emitted in each 'data' event is a Buffer. If you know it's going to be string data, the best thing to do is collect the data in an array, then at the 'end', concatenate and stringify it.

- **Setting Response Headers:** you can *explicitly* write the headers to the response stream. To do this, there's a method called writeHead, which writes the status code and the headers to the stream. Once you've set the headers, you're ready to start sending response data.
- **Sending Response Body:** Since the response object is a WritableStream, writing a response body out to the client is just a matter of using the usual stream methods.
- The end function on streams can also take in some optional data to send as the last bit of data on the stream
- It's important to set the status and headers *before* you start writing chunks of data to the body, since headers come before the body in HTTP responses.



```
response.write('<html>');
response.write('<body>');
response.write('<h1>Hello, World!</h1>');
response.write('</body>');
response.write('</html>');
response.end();
```



```
response.end('<html><body><h1>Hello, World!</h1></body></html>');
```

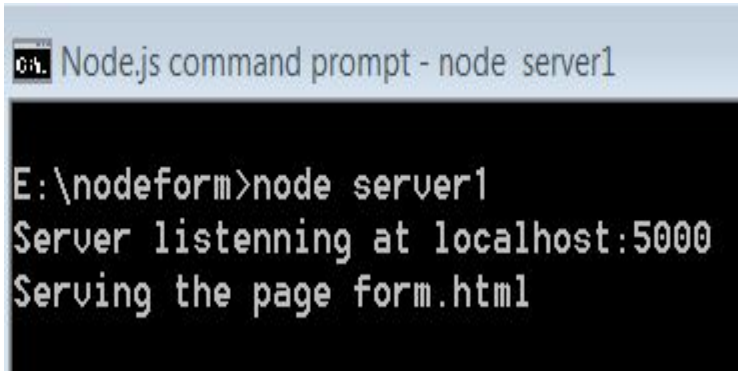
About Errors

- An error in the request stream presents itself by emitting an 'error' event on the stream. **If you don't have a listener for that event, the error will be *thrown*, which could crash your Node.js program.** You should therefore add an 'error' listener on your request streams, even if you just log it and continue on your way.
- The response stream can also emit 'error' events.
- The HTTP status code on a response will always be 200.
- response.statusCode = 404; // Tell the client that the resource wasn't found.

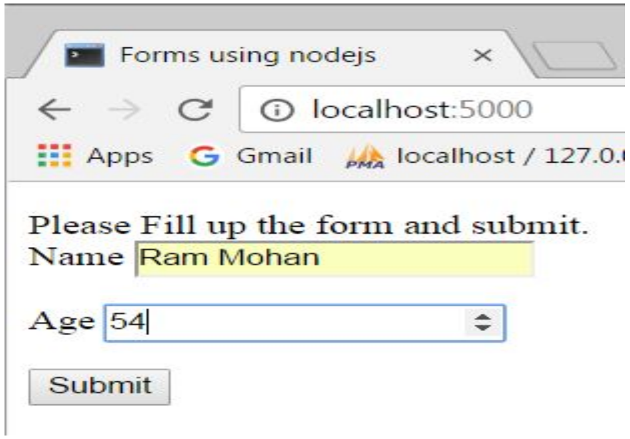
- **fs** is a built-in module in the platform for accessing file system. Using fs, we can perform CRUD operations over files and directories. It contains both synchronous and asynchronous APIs for talking to the file system. Use of asynchronous API is preferred as it will not block the event loop till the operation is completed.
- In the above code we've created a HTTP server. readFile() is used to read the form.html file. The form.html file contains the HTML code to build the fields of a form which is meant to be displayed to the user. If we click on the "Submit" button, it'll do a get/post request with the submitted form information to the same URL.
- Here we first grab the requested url parts and then check the pathname for '/' and the page with form will be sent to client side. Now user can see the page with the form. When user fills up and submits the form then server will get the request 'http://localhost:8080/getData' and the grabbed data will be sent back to client side with a confirmation message.

- We know Node.js is a event driven IO platform. So here we implemented the data event handler, the body variable taking all the data and finally the qs.parse() method pasres the data into different parts as the form fields. To use the qs.parse() we need to require the module named 'querystring'. Finally the data found from the POST request is sent back to the client as a confirmation.
- We used the module 'querystring' to parse the data into the POST. In the POST processing code see the req.on is waiting for the data event and when the data event happens we grab the data and finally at the data event 'end' we parse the data and send back to the client side.
- We can test the system for GET and POST by changing the form method to get and post.

- So this form will be sent to client side on client request. After filling up and successfully submission the form, the form data is grabbed and processed. The form will submitted to the getData function. Finally this will process the data and send a message.
- Then, we're starting the server at port 5000. After starting the server, if we go to localhost:5000 we should see the HTML form displayed as shown below.



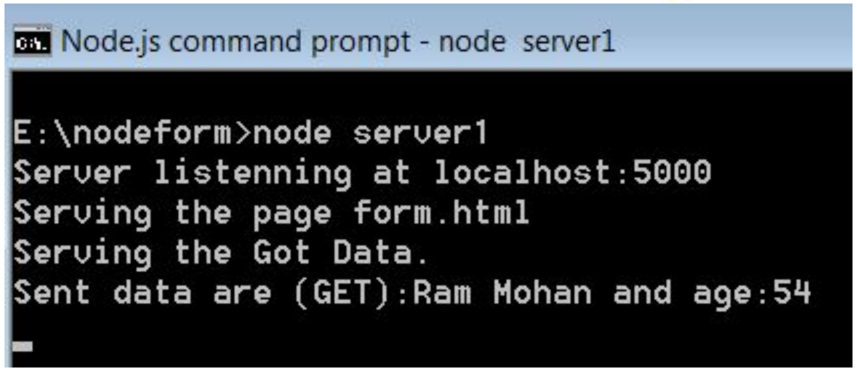
```
E:\nodeform>node server1
Server listenning at localhost:5000
Serving the page form.html
```



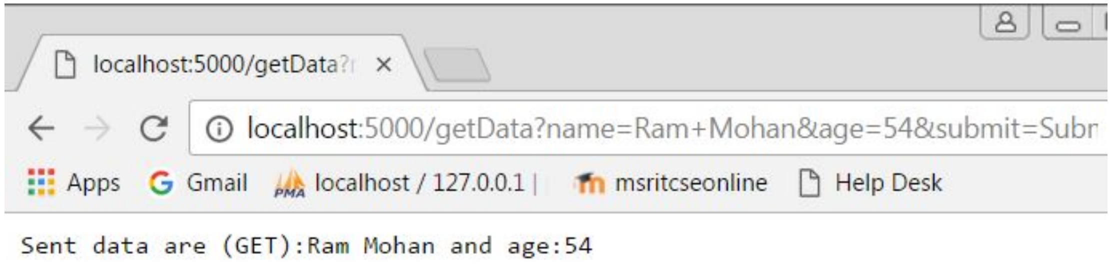
Please Fill up the form and submit.
Name Ram Mohan

Age 54

```
<form id="form" name="form" method="get" action="getData">
```



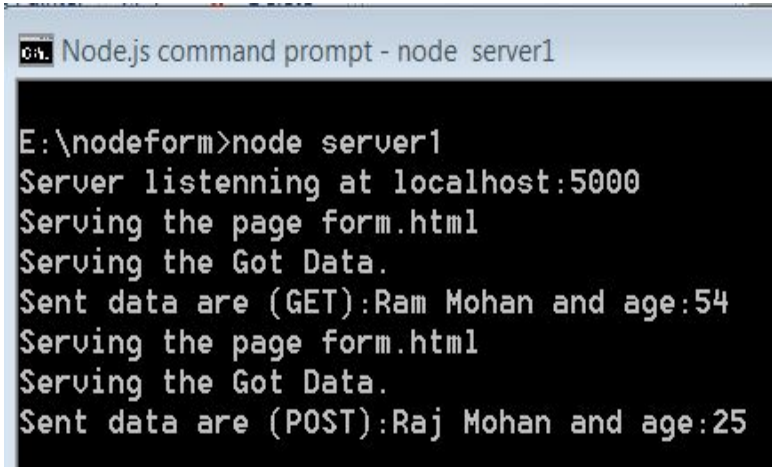
```
E:\nodeform>node server1
Server listenning at localhost:5000
Serving the page form.html
Serving the Got Data.
Sent data are (GET):Ram Mohan and age:54
```



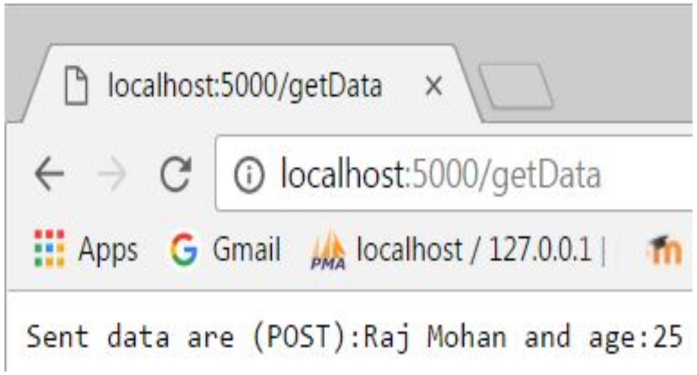
```
localhost:5000/getData?name=Ram+Mohan&age=54&submit=Subr
```

```
Sent data are (GET):Ram Mohan and age:54
```

```
<form id="form" name="form" method="post" action="getData">
```



```
E:\nodeform>node server1
Server listenning at localhost:5000
Serving the page form.html
Serving the Got Data.
Sent data are (GET):Ram Mohan and age:54
Serving the page form.html
Serving the Got Data.
Sent data are (POST):Raj Mohan and age:25
```



```
localhost:5000/getData
```

```
Sent data are (POST):Raj Mohan and age:25
```