Export Module in Node.js

The `module.exports` is a special object which is included in every JavaScript file in the Node.js application by default. The `module` is a variable that represents the current module, and `exports` is an object that will be exposed as a module. So, whatever you assign to `module.exports` will be exposed as a module.

Let's see how to expose different types as a module using `module.exports`.

Export Literals

As mentioned above, `exports` is an object. So it exposes whatever you assigned to it as a module. For example, if you assign a string literal then it will expose that string literal as a module.

The following example exposes simple string message as a module in Message.js.

Message.js

```
module.exports = 'Hello world';
```

Now, import this message module and use it as shown below.

app.js

```
var msg = require('./Message.js');

console.log(msg);
```

Note:
You must specify `./` as a path of root folder to import a local module. However, you do not need to specify the path to import Node.js core modules or NPM modules in the `require()` function.

Export Object

The `exports` is an object. So, you can attach properties or methods to it. The following example exposes an object with a string property in `Message.js` file.

Message.js

```
exports.SimpleMessage = 'Hello world';

//or

module.exports.SimpleMessage = 'Hello world';
```

In the above example, we have attached a property `SimpleMessage` to the exports object. Now, import and use this module, as shown below.

app.js

```
var msg = require('./Messages.js');

console.log(msg.SimpleMessage);
```

In the above example, the `require()` function will return an object `{ SimpleMessage : 'Hello World'}` and assign it to the msg variable. So, now you can use `msg.SimpleMessage`.

Run the above example by writing `node app.js` in the command prompt and see the output as shown below.

In the same way as above, you can expose an object with function. The following example exposes an object with the `log` function as a module.

Log.js

```
module.exports.log = function (msg) {
    console.log(msg);
};
```

The above module will expose an object- `{ log : function(msg){ console.log(msg); } }` . Use the above module as shown below.

app.js

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

msg.log('Hello World');
```

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);
```

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 `()`.

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.

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.

In this way, you can export and import a local module created in a separate file under root folder.

Node.js also allows you to create modules in sub folders. Let's see how to load module from sub folders.

Load Module from the Separate Folder

Use the full path of a module file where you have exported it using `module.exports`. For example, if the log module in the log.js is stored under the `utility` folder under the root folder of your application, then import it, as shown below.

app.js

```
var log = require('./utility/log.js');
```

In the above example, . is for the root folder, and then specify the exact path of your module file. Node.js also allows us to specify the path to the folder without specifying the file name. For example, you can specify only the utility folder without specifying `log.js`, as shown below.

app.js

```
var log = require('./utility');
```

In the above example, Node.js will search for a package definition file called `package.json` inside the utility folder. This is because Node assumes that this folder is a package and will try to look for a package definition. The `package.json` file should be in a module directory. The `package.json` under utility folder specifies the file name using the `main` key, as shown below.

./utility/package.json
Copy

```
{
    "name" : "log",
    "main" : "./log.js"
}
```

Now, Node.js will find the `log.js` file using the `main` entry in `package.json` and import it.

Demonstrates the “http” module and “url” module

The Built-in HTTP Module(creation of nodejs server)

Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

To include the HTTP module, use the `require()` method:

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

Node.js as a Web Server

The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

Use the `createServer()` method to create an HTTP server:

Example(Get your own Node.js Server)

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

//create a server object:
http.createServer(function (req, res) {
  res.write('Hello World!'); //write a response to the client
  res.end(); //end the response
}).listen(8080); //the server object listens on port 8080
```

The function passed into the `http.createServer()` method, will be executed when someone tries to access the computer on port 8080.

Save the code above in a file called "demo_http.js", and initiate the file:

Initiate demo_http.js:

```
C:\Users\Your Name>node demo_http.js
```

If you have followed the same steps on your computer, you will see the same result as the example: http://localhost:8080

Add an HTTP Header

If the response from the HTTP server is supposed to be displayed as HTML, you should include an HTTP header with the correct content type:

Example

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('Hello World!');
  res.end();
}).listen(8080);
```

The first argument of the `res.writeHead()` method is the status code, 200 means that all is OK, the second argument is an object containing the response headers.

Read the Query String

The function passed into the `http.createServer()` has a `req` argument that represents the request from the client, as an object (http.IncomingMessage object).

This object has a property called "url" which holds the part of the url that comes after the domain name:

demo_http_url.js

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write(req.url);
  res.end();
}).listen(8080);
```

The Built-in URL Module

The URL module splits up a web address into readable parts.

To include the URL module, use the `require()` method:

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

Parse an address with the `url.parse()` method, and it will return a URL object with each part of the address as properties:

Example

Split a web address into readable parts:

```
var url = require('url');
var adr = 'http://localhost:8080/default.htm?year=2017&month=february';
var q = url.parse(adr, true);

console.log(q.host); //returns 'localhost:8080'
console.log(q.pathname); //returns '/default.htm'
console.log(q.search); //returns '?year=2017&month=february'

var qdata = q.query; //returns an object: { year: 2017, month: 'february' }
console.log(qdata.month); //returns 'february'
```

Demonstrates the Node.js File System

Node.js as a File Server

The Node.js file system module allows you to work with the file system on your computer.

To include the File System module, use the `require()` method:

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

Common use for the File System module:

- Read files
- Create files
- Update files
- Delete files
- Rename files

Read Files

The `fs.readFile()` method is used to read files on your computer.

Assume we have the following HTML file (located in the same folder as Node.js):

demofile1.html

```
<html>
<body>
<h1>My Header</h1>
<p>My paragraph.</p>
</body>
</html>
```

Create a Node.js file that reads the HTML file, and return the content:

Example

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  fs.readFile('demofile1.html', function(err, data) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    return res.end();
  });
}).listen(8080);
```

Save the code above in a file called "demo_readfile.js", and initiate the file:

Initiate demo_readfile.js:

```
C:\Users\Your Name>node demo_readfile.js
```

If you have followed the same steps on your computer, you will see the same result as the example: http://localhost:8080

Create Files

The File System module has methods for creating new files:

- `fs.appendFile()`
- `fs.open()`
- `fs.writeFile()`

The `fs.appendFile()` method appends specified content to a file. If the file does not exist, the file will be created:

Example

Create a new file using the appendFile() method:

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

fs.appendFile('mynewfile1.txt', 'Hello content!', function (err) {
  if (err) throw err;
  console.log('Saved!');
});
```

The `fs.open()` method takes a "flag" as the second argument, if the flag is "w" for "writing", the specified file is opened for writing. If the file does not exist, an empty file is created:

Example

Create a new, empty file using the open() method:

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

fs.open('mynewfile2.txt', 'w', function (err, file) {
  if (err) throw err;
  console.log('Saved!');
});
```

The `fs.writeFile()` method replaces the specified file and content if it exists. If the file does not exist, a new file, containing the specified content, will be created:

Example

Create a new file using the writeFile() method:

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

fs.writeFile('mynewfile3.txt', 'Hello content!', function (err) {
  if (err) throw err;
  console.log('Saved!');
});
```

Update Files

The File System module has methods for updating files:

- `fs.appendFile()`
- `fs.writeFile()`

The `fs.appendFile()` method appends the specified content at the end of the specified file:

Example

Append "This is my text." to the end of the file "mynewfile1.txt":

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

fs.appendFile('mynewfile1.txt', ' This is my text.', function (err) {
  if (err) throw err;
  console.log('Updated!');
});
```

The `fs.writeFile()` method replaces the specified file and content:

Example

Replace the content of the file "mynewfile3.txt":

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

fs.writeFile('mynewfile3.txt', 'This is my text', function (err) {
  if (err) throw err;
  console.log('Replaced!');
});
```

Delete Files

To delete a file with the File System module, use the `fs.unlink()` method.

The `fs.unlink()` method deletes the specified file:

Example

Delete "mynewfile2.txt":

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

fs.unlink('mynewfile2.txt', function (err) {
  if (err) throw err;
  console.log('File deleted!');
});
```

Rename Files

To rename a file with the File System module, use the `fs.rename()` method.

The `fs.rename()` method renames the specified file:

Example

Rename "mynewfile1.txt" to "myrenamedfile.txt":

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

fs.rename('mynewfile1.txt', 'myrenamedfile.txt', function (err) {
  if (err) throw err;
  console.log('File Renamed!');
});
```

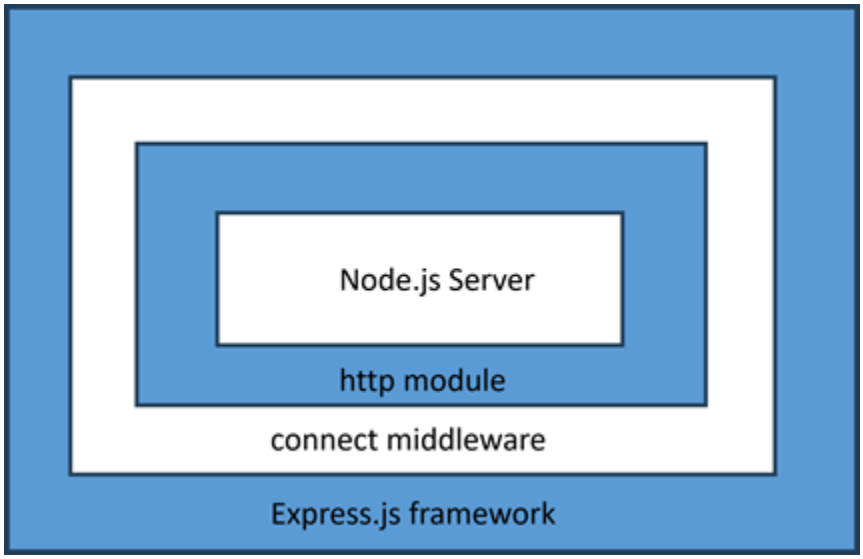
What is express.js .how to create Express. Js server with code

Express.js is a minimal and flexible web application framework that provides a robust set of features to develop Node.js based web and mobile applications. Express.js is one of the most popular web frameworks in the Node.js ecosystem. Express.js provides all the features of a modern web framework, such as templating, static file handling, connectivity with SQL and NoSQL databases.

Following are some of the core features of Express framework –

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

The Express.js is built on top of the connect middleware, which in turn is based on http, one of the core modules of Node.js API.



Installing Express

The Express.js package is available on npm package repository. Let us install express package locally in an application folder named ExpressApp.

```
D:\expressApp> npm init
D:\expressApp> npm install express --save
```

The above command saves the installation locally in the node_modules directory and creates a directory express inside node_modules.

Hello world Example

Following is a very basic Express app which starts a server and listens on port 5000 for connection. This app responds with Hello World! for requests to the homepage. For every other path, it will respond with a 404 Not Found.

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

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(5000, function () {
   console.log("Express App running at http://127.0.0.1:5000/");
})
```

Save the above code as index.js and run it from the command-line.

```
D:\expressApp> node index.js
Express App running at http://127.0.0.1:5000/
```

Visit http://localhost:5000/ in a browser window. It displays the Hello World message.

Application object

An object of the top level express class denotes the application object. It is instantiated by the following statement –

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

The Application object handles important tasks such as handling HTTP requests, rendering HTML views, and configuring middleware etc.

The app.listen() method creates the Node.js web server at the specified host and port. It encapsulates the createServer() method in http module of Node.js API.

```
app.listen(port, callback);
```

Basic Routing

The app object handles HTTP requests GET, POST, PUT and DELETE with app.get(), app.post(), app.put() and app.delete() method respectively. The HTTP request and HTTP response objects are provided as arguments to these methods by the NodeJS server. The first parameter to these methods is a string that represents the endpoint of the URL. These methods are asynchronous, and invoke a callback by passing the request and response objects.

GET method

In the above example, we have provided a method that handles the GET request when the client visits '/' endpoint.

```
app.get('/', function (req, res) {
  res.send('Hello World');
})
```

- Request Object – The request object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, and so on.
- Response Object – The response object represents the HTTP response that an Express app sends when it gets an HTTP request. The send() method of the response object formulates the server's response to the client.

You can print request and response objects which provide a lot of information related to HTTP request and response including cookies, sessions, URL, etc.

The response object also has a sendFile() method that sends the contents of a given file as the response.

```
res.sendFile(path)
```

Save the following HTML script as index.html in the root folder of the express app.

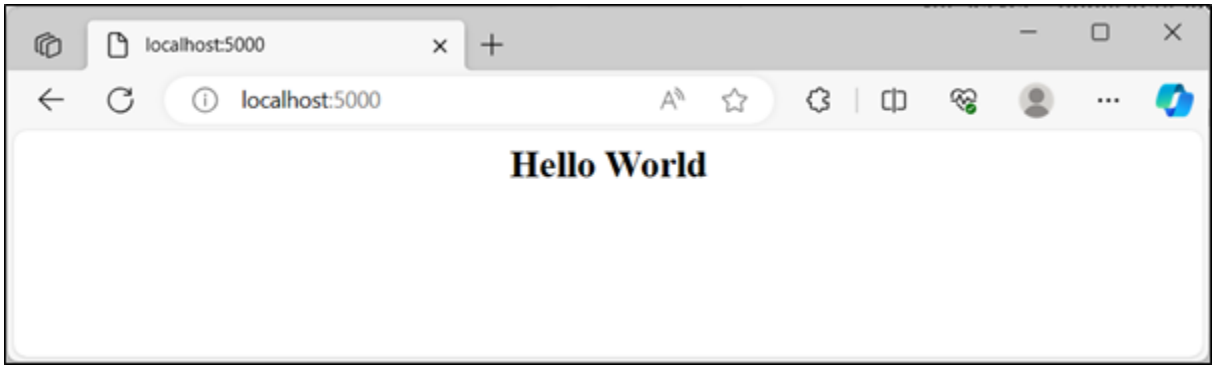
```
<html>
<body>
<h2 style="text-align: center;">Hello World</h2>
</body>
</html>
```

Change the index.js file to the code below –

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

app.get('/', function (req, res) {
  res.sendFile(path.join(__dirname,"index.html"));
})

var server = app.listen(5000, function () {
```

```
  console.log("Express App running at http://127.0.0.1:5000/");
})
```

Run the above program and visit http://localhost:5000/, the browser shows Hello World message as below:



Let us use sendFile() method to display a HTML form in the index.html file.

```
<html>
  <body>

    <form action = "/process_get" method = "GET">
      First Name: <input type = "text" name = "first_name">  <br>
      Last Name: <input type = "text" name = "last_name">  <br>
      <input type = "submit" value = "Submit">
    </form>

  </body>
</html>
```

The above form submits the data to /process_get endpoint, with GET method. Hence we need to provide a app.get() method that gets called when the form is submitted.

```
app.get('/process_get', function (req, res) {
  // Prepare output in JSON format
  response = {
    first_name:req.query.first_name,
    last_name:req.query.last_name
  };
  console.log(response);
```

```
   res.end(JSON.stringify(response));
})
```

The form data is included in the request object. This method retrieves the data from request.query array, and sends it as a response to the client.

The complete code for index.js is as follows −

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

app.use(express.static('public'));

app.get('/', function (req, res) {
   res.sendFile(path.join(__dirname,"index.html"));
})

app.get('/process_get', function (req, res) {
   // Prepare output in JSON format
   response = {
      first_name:req.query.first_name,
      last_name:req.query.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})

var server = app.listen(5000, function () {
   console.log("Express App running at http://127.0.0.1:5000/");
})
```

Visit http://localhost:5000/.

Now you can enter the First and Last Name and then click submit button to see the result and it should return the following result –

```
{"first_name":"John","last_name":"Paul"}
```

POST method

The HTML form is normally used to submit the data to the server, with its method parameter set to POST, especially when some binary data such as images is to be submitted. So, let us change the method parameter in index.html to POST, and action parameter to "process_POST".

```
<html>
  <body>

    <form action = "/process_POST" method = "POST">
      First Name: <input type = "text" name = "first_name">  <br>
      Last Name: <input type = "text" name = "last_name">  <br>
      <input type = "submit" value = "Submit">
    </form>

  </body>
</html>
```

To handle the POST data, we need to install the body-parser package from npm. Use the following command.

```
npm install body-parser –save
```

This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data.

This package is included in the JavaScript code with the following require statement.

```
var bodyParser = require('body-parser');
```

The urlencoded() function creates application/x-www-form-urlencoded parser

```
// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })
```

Add the following app.post() method in the express application code to handle POST data.

```
app.post('/process_post', urlencodedParser, function (req, res) {
   // Prepare output in JSON format
   response = {
      first_name:req.body.first_name,
      last_name:req.body.last_name
   };
   console.log(response);
   res.end(JSON.stringify(response));
})
```

Here is the complete code for index.js file

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

var bodyParser = require('body-parser');
// Create application/x-www-form-urlencoded parser
var urlencodedParser = bodyParser.urlencoded({ extended: false })

app.use(express.static('public'));

app.get('/', function (req, res) {
   res.sendFile(path.join(__dirname,"index.html"));
})

app.get('/process_get', function (req, res) {
   // Prepare output in JSON format
   response = {
      first_name:req.query.first_name,
      last_name:req.query.last_name
   };
   console.log(response);
```

```
   res.end(JSON.stringify(response));
})
app.post("/process_post", )
var server = app.listen(5000, function () {
   console.log("Express App running at http://127.0.0.1:5000/");
})
```

Run index.js from command prompt and visit http://localhost:5000/.

Now you can enter the First and Last Name and then click the submit button to see the following result –

```
{"first_name":"John","last_name":"Paul"}
```

Serving Static Files

Express provides a built-in middleware express.static to serve static files, such as images, CSS, JavaScript, etc.

You simply need to pass the name of the directory where you keep your static assets, to the express.static middleware to start serving the files directly. For example, if you keep your images, CSS, and JavaScript files in a directory named public, you can do this –

```
app.use(express.static('public'));
```

We will keep a few images in public/images sub-directory as follows –

```
node_modules
index.js
public/
public/images
public/images/logo.png
```

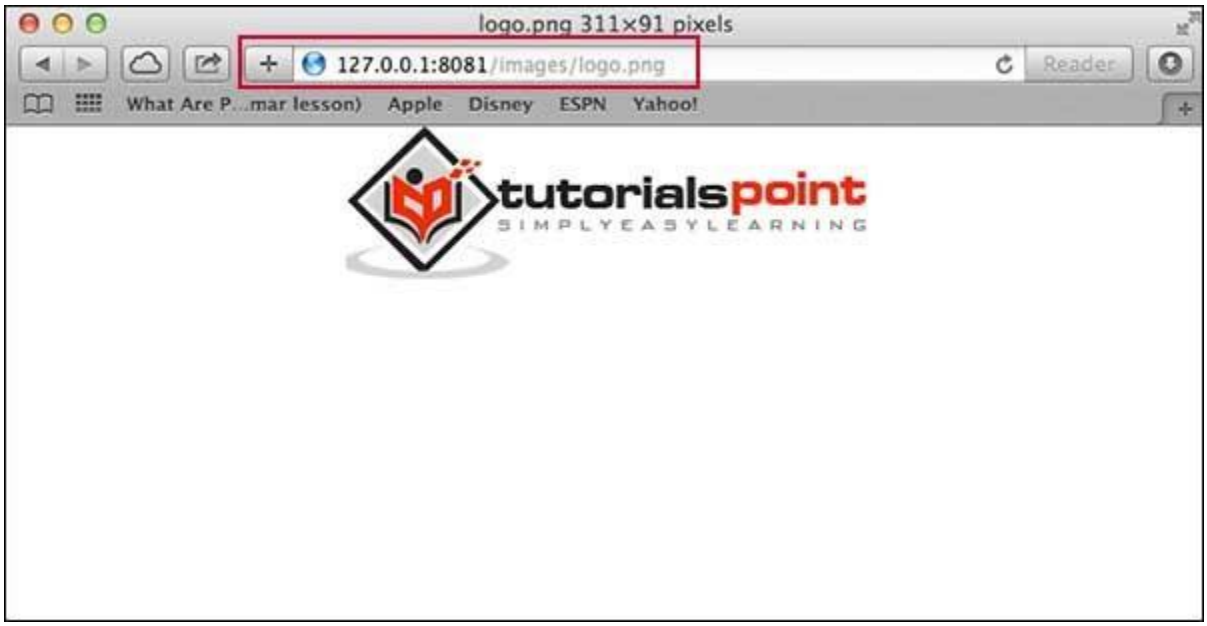
Let's modify "Hello Word" app to add the functionality to handle static files.

```
var express = require('express');
var app = express();
app.use(express.static('public'));

app.get('/', function (req, res) {
   res.send('Hello World');
})

var server = app.listen(5000, function () {
   console.log("Express App running at http://127.0.0.1:5000/");
})
```

Save the above code in a file named index.js and run it with the following command.

```
D:\expressApp> node index.js
```

Now open http://127.0.0.1:5000/images/logo.png in any browser and see observe following result.



To learn Express.js in details, visit our ExpressJS Tutorial (ExpressJS)