Node.js

Lecture (1)

PHP vs Node.js

Here is how PHP or ASP handles a file request:

1. Sends the task to the computer's file system.
2. Waits while the file system opens and reads the file.
3. Returns the content to the client.
4. Ready to handle the next request.

Here is how Node.js handles a file request:

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

- Node.js eliminates the waiting, and simply continues with the next request.
- Node.js runs single-threaded, non-blocking, asynchronous programming, which is very memory efficient.

What Can Node.js Do?

- Node.js can generate dynamic page content
- Node.js can create, open, read, write, delete, and close files on the server
- Node.js can collect form data
- Node.js can add, delete, modify data in your database

What is a Node.js File?

- Node.js files contain tasks that will be executed on certain events
- A typical event is someone trying to access a port on the server
- Node.js files must be initiated on the server before having any effect
- Node.js files have extension ".js"

Node.js at a glance - agenda

- Async Read/Write
- Sync Read/Write
- Node.js Network call (“net”)
- Node.js modules
- Web – Node.js (“http”)
- Web – Node.js (“express”) (html form with get/post)
- Web – Node.js (“express”) (routing)
- Web – Node.js (“express”) (middleware)
- Web – Node.js (“express”) (router)

Node.js Modules

- Consider modules to be the same as JavaScript libraries.
- A set of functions you want to include in your application.
- Node.js has a set of built-in modules which you can use without any further installation.

Node.js – fs (Async Read)

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

fs.readFile('./fn_input.txt', function (err, data) {
   if (err) return console.error(err);
   in_data = data;
   console.log('Async input file content: ' + in_data);
});

console.log('Program Ended.');
```

Node.js – fs (Async Write)

```
var fs = require('fs');
var out_data = 'Output line 1.\r\nOutput line 2.\r\nOutput last line.';

fs.writeFile('./async_output.txt', out_data, function (err) {
        if (err) console.error(err);
        console.log('Async output file content: ' + out_data);
});

console.log('Program Ended.');
```

Node.js – fs (Sync Read)

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

var in_data = fs.readFileSync('./fn_input.txt');

console.log('Sync input file content: ' + in_data);

console.log('Program Ended.');
```

Node.js – fs (Sync Write)

```
var fs = require('fs');
var out_data = 'Output line 1.\r\nOutput line 2.\r\nOutput last line.';

fs.writeFileSync('./sync_output.txt', out_data);
console.log('Sync output file content: ' + out_data);

console.log('Program Ended.');
```

Node.js – net (Network Server)

```
var net = require("net");
var server = net.createServer(function(connection) {
    console.log('Client connected.');
    connection.on('end', function() {
            console.log('Client disconnected.');
    });
    connection.write('Hello World!\n');
    connection.pipe(connection);
// send data back to connection object which is client
});
server.listen(8080, function() {
    console.log('Server is listening.');
});
console.log('Server Program Ended.');
```

Node.js – net (Network Client)

```
var client = net.connect({port: 8080}, 'localhost', function() {
    console.log('Connected to Server.');
});

client.on('data', function (data) {
    console.log(data.toString());
    client.end();
});

client.on('end', function () {
    console.log('Disconnected from server.');
});
console.log('Client Program Ended.');
```

Node.js (Modules)

Module name	Description
buffer	buffer module can be used to create Buffer class.
console	console module is used to print information on stdout and stderr.
dns	dns module is used to do actual DNS lookup and underlying o/s name resolution functionalities.
domain	domain module provides way to handle multiple different I/O operations as a single group.
fs	fs module is used for File I/O.
net	net module provides servers and clients as streams. Acts as a network wrapper.
os	os module provides basic o/s related utility functions.
path	path module provides utilities for handling and transforming file paths.
process	process module is used to get information on current process.

Node.js – web module

- Web server is a software application which processes request using **HTTP protocol** and returns web pages as response to the clients.
- Web server usually delivers html documents along with images, style sheets and scripts.
- Most web server also support server-side scripts using scripting language or redirect to application server which perform the specific task of getting data from database, perform complex logic etc. Web server then returns the output of the application server to client.
- Apache web server is one of the most common web server being used. It is an open-source project.

Web server and Path

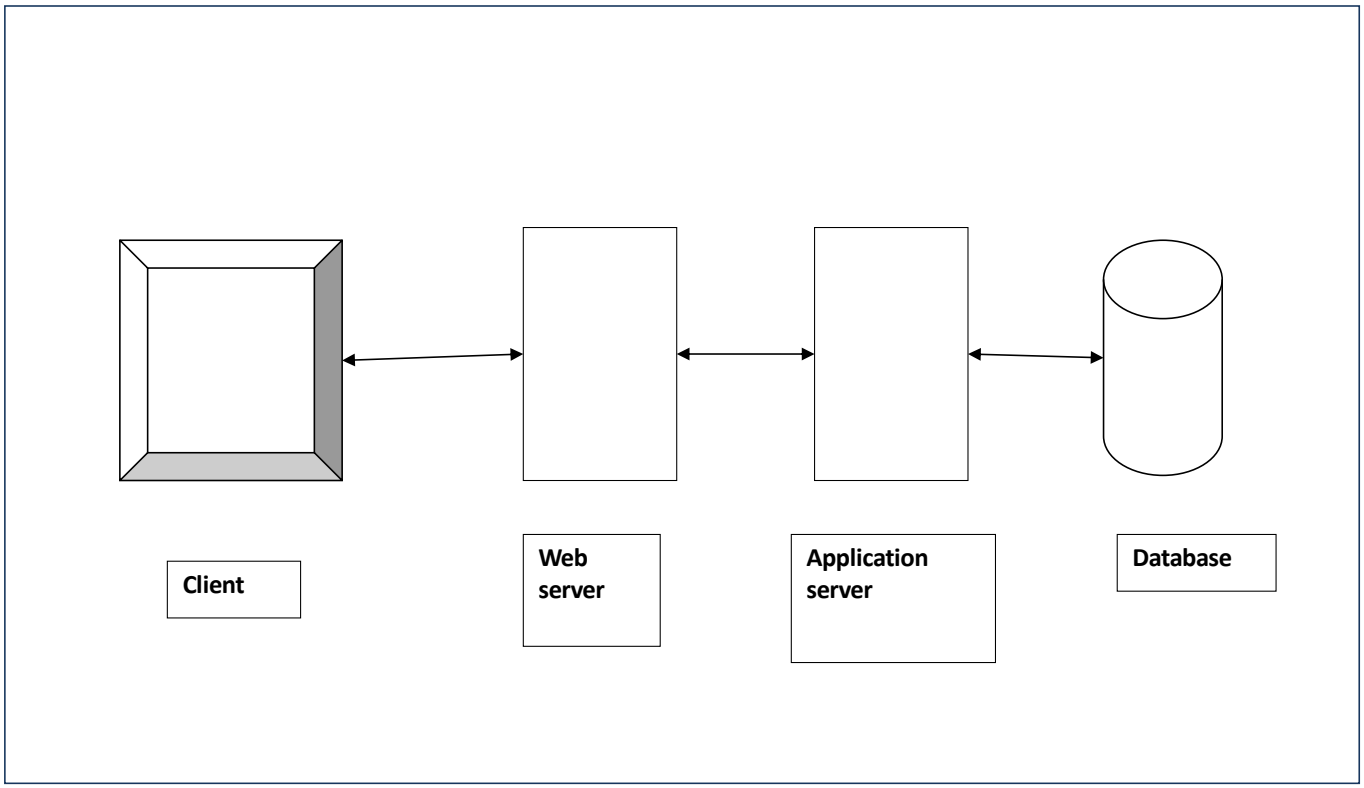
- Web server maps the path of a file using URL, Uniform Resource Locator. It can be a local file system or an external/internal program.
- A client makes a request using browser, URL: http://www.abc.com/dir/index.html
- Browser will make a request as:

```
GET /dir/index.html  HTTP/1.1
HOST www.abc.com
```

Web Architecture

- Web application are normally divided into four layers:
  - Client: this layer consists of web browsers, mobile browsers or applications which can make HTTP request to server.
  - Server: this layer consists of Web Server which can intercepts the request made by clients and pass them the response.
  - Business: this layer consists of application server which is utilized by web server to do dynamic tasks. This layer interacts with data layer via database or some external programs.
  - Data: this layer consists of databases or any source of data.

Web Architecture



Creating Web Server using Node (http module)

- Create an HTTP server using http.createServer method. Pass it a function with parameters request and response.
- Write the sample implementation to return a requested page.
- Pass a port 8081 to listen method.
- http_server.js
- test.html
- http_client.js

http_server.js

```
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.stack);
                        response.writeHead(404, {'Content-Type' : 'text/html'});       // HTTP status: 404 : NOT FOUND
                } else {   response.writeHead(200, {'Content-Type' : 'text/html'});         // HTTP status: 200 : OK
                        response.write(data.toString());
                }
                response.end();                         // send the response body
    });
}).listen(8081);
console.log('Server running at http://127.0.0.1:8081/test.html');
console.log('Server Program Ended.');
```

test.html

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8">
<title>wk4_01_test</title>
</head>
<body>
Hello World!
</body>
</html>
```

http_client.js

```
var http = require('http');
var options = {
    host: 'localhost', port: '8081', path: '/test.html'
};
var callback = function (response) {            // callback function is used to deal with response
    var body = '';
    response.on('data', function (data) {
                body += data;
    });
    response.on('end', function () {
                console.log(body);
    });
    response.on('error', (error) => {
                console.error(error);
    });
};
var req = http.request(options, callback);
req.end();
```

Express JS

Lecture (2)

Node.js – Express

- Express js is a very popular web application framework built to create Node.js Web based applications.
- It provides an integrated environment to facilitate rapid development of Node based Web applications.
- Express framework is based on Connect middleware engine and used Jade html template framework for HTML templating.
- 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 action based on HTTP method and URL.
  - Allows to dynamically render HTML Pages based on passing arguments to templates.

Installing Express

- From the nodejs_workspace (working directory), un-install express if you install express locally from previous chapter.

  npm uninstall express

- Create a directory myapp which is under the nodejs_workspace directory. (mkdir myapp, then cd myapp). [http://expressjs.com/en/starter/installing.html]
- Use the npm init command under the new myapp folder to create a package.json file for your application – myapp.

  npm init

- NOTE: you can hit RETURN to accept the defaults for most of them, except entry point is app.js:

  entry point: app.js

- Install express locally in the nodejs_workspace directory, and save it in the dependencies list:

  npm install express --save

- Install the Express framework globally using npm if you want to create web application using node terminal.

  npm install express -g --save

Node.js – Web Application with express

- http://expressjs.com/en/starter/hello-world.html
- In myapp directory, create a file named app.js and add the following codes:

```
var express = require('express');
var app = express();
app.get('/', (req, res) =>{
        res.send('Hello World!');
});
app.listen(3000, function () {
        console.log('app.js listening to http://localhost:3000/');
});
```

- Run the app.js in the server:

  node app.js

- Then, load http://localhost:3000/ in a browser to see the output.

Web application – get/post form (server.js)

```
var express = require('express');
var app = express();
app.get('/', (req, res) =>{                    // To display index.html
   res.sendFile(__dirname + "/index.html");
});
app.get('/process_get', (req, res) =>{         // To process get method
   var response = { fname: req.query.fname,
            lname: req.query.lname };  // preparing the output in JSON format
   console.log(response);
   res.json(response);
});
```

Basic Routing

- ***Routing*** refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).
- Each route can have one or more handler functions, which are executed when the route is matched.
- Route definition takes the following structure:

```
app.METHOD(PATH, HANDLER);
```

- Where:
  - app is an instance of express.
  - METHOD is an HTTP request method, in lower case.
  - PATH is a path on the server
  - HANDLER is the function executed when the route is matched.

Route methods

- A route method is derived from one of the HTTP methods, and is attached to an instance of the express class.
- The following code is an example of routes that are defined for the GET and POST methods to the root of the app.
- GET method route:

```
app.get('/', (req, res)=> {
        res.send('Get request to the homepage');
});
```

- POST method route:

```
app.post('/', (req, res) =>{
        res.send('Post request to the hmoepage');
});
```

- There is a special routing method, app.all(), which is not derived from any HTTP method. This method is used for loading middleware functions at a path for all request methods. In the following example, the handler will be executed for requests to "/secret" whether you are using GET, POST, PUT, DELETE, or any other HTTP request method that is supported in the http module.

```
app.all('/secret', (req, res, next) => {
        console.log('Accessing the secret section ...');
        next(); // pass control to the next handler
});
```

Node.js - routing (app_route.js)

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

// This route path will match requests to the root route, /
app.get('/', (req, res)=> {
   res.send('Hello World! get method');
});

// This route path will match requests to /user
app.all('/user', (req, res)=> {
   console.log('Accessing /user...');
   res.send('Hello World! /user request...');
});
```

Node.js - routing (app_route.js)

```
// This route path will match requests to /random.text
app.get('/random.txt', (req, res)=> {
    res.send('Hello World! get /random.text');
});
// To define routes with route parameters, simply specify the route parameters in the path of the route as
    shown below.
app.get('/users/:userId/books/:bookId', (req, res)=> {
    res.send(req.params);
});
app.get('/floats/:digit.:decimal', function(req, res) {
    res.send(req.params);
});
app.listen(3000)
```

Route paths

- Route paths, in combination with a request method, define the endpoints at which requests can be made. Route paths can be strings, string patterns, or regular expressions.
- The characters ?, +, *, and () are subsets of their regular expression counterparts. The hyphen (-) and the dot (.) are interpreted literally by string-based paths.
- NOTE: Express uses path-to-regexp for matching the route paths; see the path-to-regexp documentation for all the possibilities in defining route paths. Express Route Tester is a handy tool for testing basic Express routes, although it does not support pattern matching.
- NOTE: Query strings are not part of the route path.

Route paths based on strings

- This route path will match requests to the root route, /.

```
app.get('/', (req, res)=> {
        res.send('root'); });
```

- This route path will match requests to /about.

```
app.get('/about', (req, res)=> {
        res.send('about'); });
```

- This route path will match requests to /random.text.

```
app.get('/random.text', (req, res) =>{
res.send('random.text'); });
```

Route Parameters

- Route parameters are named URL segments that are used to capture the values specified at their position in the URL. The captured values are populated in the req.params object, with the name of the route parameter specified in the path as their respective keys.

  Route path: /users/:userId/books/:bookId

  Request URL: http://localhost:3000/users/34/books/8989
  req.params: { "userId": "34", "bookId": "8989" }

- To define routes with route parameters, simply specify the route parameters in the path of the route as shown below.

```
app.get('/users/:userId/books/:bookId', (req, res) =>{
res.send(req.params); });
```

Route parameters (continue…)

- Since the hyphen (-) and the dot (.) are interpreted literally, they can be used along with route parameters for useful purposes.

  Route path: /flights/:from-:to

  Request URL: http://localhost:3000/flights/LAX-SFO

  req.params: { "from": "LAX", "to": "SFO" }

  Route path: /floats/:digit.:decimal

  Request URL: http://localhost:3000/floats/123.45

  req.params: { "digit": "123", "decimal": "45" }

- NOTE: The name of route parameters must be made up of “word characters” ([A-Za-z0-9_]).

Route handlers

- You can provide multiple callback functions that behave like middleware to handle a request. The only exception is that these callbacks might invoke next('route') to bypass the remaining route callbacks. You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there's no reason to proceed with the current route.
- Route handlers can be in the form of a function, an array of functions, or combinations of both.

Route handlers samples

- A single callback function can handle a route. For example:

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

- More than one callback function can handle a route (make sure you specify the next object). For example:

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

Route handlers samples (continue…)

- An array of callback functions can handle a route. For example:

```
var cb0 = function (req, res, next) { console.log('CB0'); next(); }

var cb1 = function (req, res, next) { console.log('CB1'); next(); }

var cb2 = function (req, res) { res.send('Hello from C!'); }

app.get('/example/c', [cb0, cb1, cb2]);
```

- A combination of independent functions and arrays of functions can handle a route. For example:

```
var cb0 = function (req, res, next) { console.log('CB0'); next(); }

var cb1 = function (req, res, next) { console.log('CB1'); next(); }
app.get('/example/d', [cb0, cb1], (req, res, next) =>{

        console.log('the response will be sent by the next function ...');
        next();

}, function (req, res) { res.send('Hello from D!'); });
```

Response Methods

- The methods on the response object (res) in the following table can send a response to the client, and terminate the request-response cycle. If none of these methods are called from a route handler, **the client request will be left hanging.**

Response Methods (continue…)

Method	Description
res.download()	Prompt a file to be downloaded.
res.end()	End the response process.
res.json()	Send a JSON response.
res.jsonp()	Send a JSON response with JSONP support.
res.redirect()	Redirect a request.
res.render()	Render a review template.
res.send()	Send a response of various types.
res.sendFile()	Send a file as an octet stream.
res.sendStatus()	Set the response status code and send its string representation as the response body.

Express Middleware

- ***Middleware*** functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named next.
- Middleware functions can perform the following tasks:
  - Execute any code.
  - Make changes to the request and the response objects.
  - End the request-response cycle.
  - Call the next middleware in the stack.
- If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

Express Middleware (continue…)

- The following shows the elements of a middleware function call:

```
var express = require('express');
var app = express();
app.get('/', (req, res, next) =>{
        next();
});
app.listen(3000);
```

- Where get: HTTP method for which the middleware function applies.
- '/': Path (route) for which the middleware function applies.
- function (): The middleware function.
- req: HTTP request argument to the middleware function.
- res: HTTP response argument to the middleware function.
- next: Callback argument to the middleware function.

app.use() method

- **app.use([path,] function [, function...])**
- Mounts the specified middleware function or functions at the specified path. If path is not specified, it defaults to '/'.
- NOTE: A route will match any path that follows its path immediately with a "/". For example, app.use('/apple', …) will match "/apple", "/apple/images", "/apple/images/news", and so on.

```
app.use('/admin', (req, res, next)=> {
// GET 'http://www.example.com/admin/new'
console.log(req.originalUrl); // '/admin/new'
console.log(req.baseUrl); // '/admin'
console.log(req.path); // '/new'
next(); });
```

- Mounting a middleware function at a path will cause the middleware function to be executed whenever the base of the requested path matches the path.

app.use() (continue…)

- Since path defaults to "/", middleware mounted without a path will be executed for every request to the app.

```
// this middleware will be executed for every request to the app
app.use((req, res, next)=> {
        console.log('Time: %d', Date.now());
next(); });
```

- Middleware functions are executed sequentially; therefore, the order of middleware inclusion is important:

```
// this middleware will not allow the request to go beyond it
app.use( (req, res, next)=> {
        res.send('Hello World'); });
// requests will never reach this route
app.get('/', (req, res)=> {
        res.send('Welcome');
});
```

Middleware sample

- http://expressjs.com/en/guide/writing-middleware.html
- Middleware function requestTime add the current time to the req (the request object).
- app_middleware.js

Express Routers

- A router object is an isolated instance of middleware and routes. You can think of it as a “mini-application,” capable only of performing middleware and routing functions. Every Express application has a built-in app router.
- A router behaves like middleware itself, so you can use it as an argument to app.use() or as the argument to another router’s use() method.
- The top-level express object has a Router() method that creates a new router object.

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

var app = express();

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

Express Routers 2

- Once you've created a router object, you can add middleware and HTTP method routes (such as get, put, post, and so on) to it just like an application.

```
// invoked for any requests passed to this router
router.use( (req, res, next) =>{
        // .. some logic here .. like any other middleware
        next(); });


// will handle any request that ends in /events
// depends on where the router is "use()'d"
router.get('/events', (req, res, next)=> {
        // .. });
```

- You can then use a router for a particular root URL in this way separating your routes into files or even mini-apps.

```
// only requests to /calendar/* will be sent to our "router"
app.use('/calendar', router);
```

Router-level Middleware

- Router-level middleware works in the same way as application-level middleware, except it is bound to an instance of express.Router().

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

- Load router-level middleware by using the router.use() and router.METHOD() functions.
- The following example code replicates the middleware system that is shown above for application-level middleware, by using router-level middleware.

Built-in Middleware

- Starting with version 4.x, Express no longer depends on Connect. With the exception of express.static, all of the middleware functions that were previously included with Express' are now in separate modules. Please view the list of middleware functions.
- The only built-in middleware function in Express is express.static. This function is based on serve-static, and is responsible for serving static assets such as HTML files, images, and so on.
- The function signature is:

  express.static(root, [options])

- The root argument specifies the root directory from which to serve static assets.
- For information on the options argument and more details on this middleware function, see express.static.

Serving Static files in Express

- To serve static files such as images, CSS files, and JavaScript files, use the express.static built-in middleware function in Express.
- Pass the name of the directory that contains the static assets to the express.static middleware function to start serving the files directly. For example, use the following code to serve images, CSS files, and JavaScript files in a directory named public:

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

- Now, you can load the files that are in the public directory:

  http://localhost:3000/images/kitten.jpg

  http://localhost:3000/css/style.css

  http://localhost:3000/js/app.js

  http://localhost:3000/images/bg.png

  http://localhost:3000/hello.html

- To use multiple static assets directories, call the express.static middleware function multiple times:

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

Serving Static files in Express 2

- Express looks up the files in the order in which you set the static directories with the express.static middleware function.
- To create a virtual path prefix (where the path does not actually exist in the file system) for files that are served by the express.static function,specify a mount path for the static directory, as shown below:

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

- Now, you can load the files that are in the public directory from the /static path prefix.

  http://localhost:3000/static/images/kitten.jpg

  http://localhost:3000/static/css/style.css

  http://localhost:3000/static/js/app.js

  http://localhost:3000/static/images/bg.png

  http://localhost:3000/static/hello.html

- However, the path that you provide to the express.static function is relative to the directory from where you launch your node process. If you run the express app from another directory, it's safer to use the absolute path of the directory that you want to serve:

```
app.use('/static', express.static(__dirname + '/public'));
```

Express application generator

- Follow instructions from http://expressjs.com/en/starter/generator.html
- Use the application generator tool, express-generator, to quickly create an application skeleton.
- Install express-generator in myapp directory:

  npm install express-generator -g
- Display the command options with the –h option:

  express -h
- Create an Express app named firstApplication in the current working directory:

  express firstApplication
- Then, install dependencies:

  cd firstApplication

  npm install

  set DEBUG=firstApplication:* & npm start
- Then load http://localhost:3000/ in your browser to access the app.

References

- Node.js the Right Way, Jim R. Wilson, The Programatic Bookshelf, ISBN 978-1-937785-73-4
- NodeJS: Practical Guide for Beginners, Matthew Gimson, ISBN 978-1-519354-07-5
- Express.js: Node.js Framework for Web Application Development, Daniel Green, ISBN 978-1-530204-06-9
- Express.com
- Tutorials Point
- The Node Beginner Book, Manuel Kiessling, Leanpub, Link.

Exercise

- Use browser first
- Use **postman** for testing your api requests