503106

ADVANCED WEB PROGRAMMING

CHAPTER 2: Middleware

LESSON 06 – Middleware

Content

- Middleware
- Middlewares example
- Third-party Middleware
- Express application generator

Middleware

- We've used existing middleware (body-parser, cookie-parser, static, and connect-session), and we've even written some of our own
- Middleware is simply a function that takes three arguments: a request object, a response object, and a "next" function
- Middleware is executed in what's known as a *pipeline*. (*order matters*)
- In an Express app, you insert middleware into the pipeline by calling app.use.
- Where should we add "404 error middleware"?

Middleware

- How is a request “terminated” in the pipeline? When you don’t call next()

Middleware and route handlers

- Route handlers (app.get, app.post, etc.—often referred to collectively as app.VERB) can be thought of as middleware that handle only a specific HTTP verb (GET, POST, etc.).
- Conversely, middleware can be thought of as a route handler that handles all HTTP verbs (this is essentially equivalent to app.all, which handles any HTTP verb)
- Route handlers require a path as their first parameter.
- Route handlers and middleware take a callback function
- If you don't call next(), the pipeline will be terminated, and no more route handlers or middleware will be processed.
- If you do call next(), middleware or route handlers further down the pipeline will be executed, but any client responses they send will be ignored.

Middlewares example

```
app.use(function(req, res, next){
        console.log('processing request for "' + req.url + '"....');
        next();
});

app.use(function(req, res, next){
        console.log('terminating request');
        res.send('thanks for playing!');
        // note that we do NOT call next() here...this terminates the request
});

app.use(function(req, res, next){
        console.log('whoops, i\'ll never get called!');
});
```

a. localhost:3000/ ==>
b. localhost:3000/a ==>
c. localhost:3000/b ==>
d. localhost:3000/c ==>
e. localhost:3000/d ==>
Thời gian 5p =>

Middlewares example

```
app.use(function(req, res, next){
        console.log('\n\nALLWAYS');
        next();
});

app.get('/a', function(req, res){
        console.log('/a: route terminated');
        res.send('a');
});
app.get('/a', function(req, res){
        console.log('/a: never called');
});
app.get('/b', function(req, res, next){
        console.log('/b: route not terminated');
        next();
});
app.use(function(req, res, next){
        console.log('SOMETIMES');
        next();
});
app.get('/b', function(req, res, next){
        console.log('/b (part 2): error thrown' );
        throw new Error('b failed');
});
```

```
app.use('/b', function(err, req, res, next){
        console.log('/b error detected and passed on');
        next(err);
});
app.get('/c', function(err, req){
        console.log('/c: error thrown');
        throw new Error('c failed');
});
app.use('/c', function(err, req, res, next){
        console.log('/c: error deteccted but not passed on');
        next();
});

app.use(function(err, req, res, next){
        console.log('unhandled error detected: ' + err.message);
        res.send('500 - server error');
});

app.use(function(req, res){
        console.log('route not handled');
        res.send('404 - not found');
});

app.listen(3000, function(){
        console.log('listening on 3000');
});
```

Third-party Middleware

- Use third-party middleware to add functionality to Express apps.
- Install the Node.js module for the required functionality, then load it in your app at the application level or at the router level.
- The following example illustrates installing and loading the cookie-parsing middleware function cookie-parser.

```
$ npm install cookie-parser
```

```
var express = require('express');
var app = express();
var cookieParser = require('cookie-parser');
// load the cookie-parsing middleware
app.use(cookieParser());
```

- For a partial list of third-party middleware functions that are commonly used with Express, see: Third-party middleware.

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.

1. Tạo thư mục bai6
2. CMD vào thư mục bai6
3. Chạy lệnh "npm install express-generator -g"
4. Chạy lệnh "express -h"
5. Chạy lệnh tạo project: express myweb --hbs
6. Chuyển vào thư mục myweb
7. Chạy lệnh "npm upgrade" và "npm audit fix --force"
8. Chạy lệnh "npm start"
9. Vào trình duyệt truy cập localhost:3000

Node.js - Streams

- What are Streams?
  - Streams are objects that let you **read** data from a source **or write data** to destination **in continuous fashion**.
- In Node, there are four types of streams:
  - Readable: stream which is used for read operation.
  - Writable: stream which is used for write operation.
  - Duplex: stream which can be used for both read and write operation.
  - Transform: a type of duplex stream where the output is computed based on input.

Streams and EventEmitter

- Each type of stream from previous page is an EventEmitter and throws several events at times.
- For example, some of the commonly used events are:
  - data: This event is fired when there is data is available to read.
  - end: This event is fired when there is no more data to read.
  - error: This event is fired when there is any error receiving or writing data.
  - finish: This event is fired when all data has been flushed to underlying system.

Streams samples

- Reading from stream: wk3_01_reading_stream.js
- Writing to stream: wk3_02_writing_stream.js
- Piping streams: Piping is a mechanism to connect output of one stream to another stream. It is normally used to get data from one stream and to pass output of that stream to another stream. Consider the above example, where we have read the input file using readStream and write to an output file using writeStream. Now we will use the piping to simplify our operation or reading from one file and writing to another file. (wk3_03_piping_stream.js)

Read and Write

```
var fs = require('fs');
var my_data = '';
// create a readable stream
var readerStream =
fs.createReadStream('./hw1_test.txt');
// set the encoding to be utf8
readerStream.setEncoding('UTF8');
// handle stream events - data, end, error
events.
readerStream.on('data', function(chunk) {
my_data += chunk;
});
readerStream.on('end', function() {
console.log(my_data);
});
readerStream.on('error', function(err) {
console.log(err.stack);
});
console.log('Program Ended.');
```

```
var fs = require('fs');
var my_data = 'Output to a file\nHello World.';
// create a writable stream
var writeStream =
fs.createWriteStream('./wk3_02_output.txt');
// write the data to stream
// set the encoding to be utf8
writeStream.write(my_data, 'UTF8');
// mark the end of file
writeStream.end();
// handle stream events - finish, error events.
writeStream.on('finish', function(chunk) {
console.log('write completed.');
});
writeStream.on('error', function(err) {
console.log(err.stack);
});
console.log('Program Ended.');
```

Piping stream

```
var fs = require('fs');
// create a readable stream
var readerStream =
fs.createReadStream('./hw1_test.txt');
// create a writable stream
var writeStream =
fs.createWriteStream('./wk3_03_output.txt');
// pipe the read and write operations
// read hw1_test.txt and write data to wk3_03_output.txt
readerStream.pipe(writeStream);
console.log('Program Ended.');
```

Node.js – File System

- **fs** module is used for File I/O.
- fs module can be imported using following syntax:

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

- Synchronous vs Asynchronous [wk3_04_sync_async_read.js]
  - Every method in fs module have synchronous as well as asynchronous form. **Asynchronous methods [readFile()]** takes a **last parameter [function (err, p2)]** as completion function **callback** and **first parameter of the callback function is error**. It is preferred to use asynchronous method instead of synchronous method as former is non-blocking where later is blocking process.

```
fs.readFile('test.txt', function(err, data) {


});
```

Flags

Flag	Description
r	Open file for reading. An exception occurs if the file does not exist.
r+	Open file for reading and writing. An exception occurs if the file does not exist.
w	Open file for writing. The file is created ifitdoesnotexist or truncated ifitexists.
w+	Open file for reading and writing. The file is created ifitdoesnotexist or truncated ifitexists.
a	Open file for appending. The file is created if it does not exist.
a+	Open file for reading and appending. The file is created if it does not exist.

Node.js – File System Methods

fs method	Description
fs.close(fd, callback)	Asynchronous close. No arguments other than a possible exception are given to the completion callback
fs.ftruncate(fd, len, callback)	Asynchronous ftruncate (remove data from the file).
fs.mkdir(path [, mode], callback)	Asynchronous mkdir. Mode defaults to 0777.
fs.open(path, flags [,mode], callback)	Asynchronous file open.
fs.rmdir(path, callback)	Asynchronous rmdir.
fs.stat(path, callback)	Asynchronous stat. The callback gets two arguments err, stats where stats is a fs.stats object.
fs.unlink(path, callback)	Asynchronous unlink (physically remove a file).
fs.write(fd, buffer, offset, length, position, callback)	Write buffer to the file specified by fd. Note: buffer can be Buffer or String; offset and length determine the part of the buffer to be written; position refers to the offset from the beginning of the file where this data should be written.

Node.js – File System sample

- Please refer to wk3_05_fs.js

C:\app\web
/app/web

path.join(__dirname,'web','bai1','phan1')
__dirname + path.sep + 'web'

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 – console module

- console is a global object and is used to print to stdout and stderr.

Method	Description
console.log([data][, …])	Prints to stdout with newline. This function can take multiple arguments in a printf-like way.
console.info([data][,…])	Prints to stdout with newline. (support printf-like way).
console.error([data][,…])	Prints to stderr with newline. (support printf-like way).
console.warn([data][,…])	Prints to stderr with newine. (support printf-like way).
console.time(label)	Make a timer.
console.timeEnd(label)	Finish timer. Record output.

Node.js – process module

- process is a global object and is used to represent Node process.
- Exit Codes: Node normally exit with a 0 status code when no more async operations are pending. There are other exit codes:

Code	Name	Description
1	Uncaught Fatal Exception	There was an uncaught exception, and it was not handled by a domain or an uncaughtException event handler.
2	Unused	Reserved by Bash.
3	Internal JavaScript Parse Error	The JavaScript source code internal in Node's bootstrapping process caused a parse error.
…		
>128	Signal Exits	If Node receives a fatal signal such as SIGKILL or SIGHUP, then its exit code will be 128 plus the value of the signal code.

Node.js – process events

- process is an eventEmitter and it emits the following events.

Event	Description
exit	Emitted when the process is about to exit.
beforeExit	This event is emitted when node empties it's event loop and has nothing else to schedule.
uncaughtException	Emitted when an exception bubbles all the way back to the event loop.
Signal Events	Emitted when the processes receives a signal such as SIGNT, SIGHUP, etc.

Node.js – process properties

- process provides many useful properties to get better control over system interactions.

Property	Description
stdout	A Writable Stream to stdout.
stderr	A Writable Stream to stderr.
stdin	A Writable Stream to stdin.
argv	An array containing the command line arguments.
execPath	This is the absolute pathname of the executable that started the process.
env	An object containing the user environment.
exitCode	A number which will be the process exit code.
version	A compiled-in property that exposes NODE-VERSION.
platform	What platform you are running on.

Node.js – process methods

- process provides many useful methods to get better control over system interactions.

Method	Description
abort()	This causes node to emit an abort. This will cause node to exit and generate a core file.
chdir(dir)	Changes the current working directory of the process or throws an exception if that fails.
cwd()	Returns the current working directory of the process.
exit(code)	Ends the process with the specified code. If omitted, exit uses the “success” cdoe 0.
uptime()	Number of seconds Node has been running.

Node.js – os module

- os module is used for few basic operating-system related utility functions.
- os module can be imported using:

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

Method	Description
os.tmpdir()	Returns the O/S default directory for temp files.
os.hostname	Returns the hostname of the O/S.
os.type()	Returns the O/S name.
os.platform()	Returns the O/S platform.
os.arch()	Returns the O/S CPU architecture. Possible values are x64, arm, and ia32.
os.release()	Returns the O/S release.
os.totalmem()	Returns the total amount of system memory in bytes.
os.freemem()	Returns the amount of free system memory in bytes.
os.networkinterfaces()	Get a list of network interfaces.

Node.js – os property

Property	Description
os.EOL	A constant defining the appropriate End-of-line marker for the operating system.

Node.js – net module

- net module is used to create both servers and clients. It provides an asynchronous network wrapper. net module can be imported using:

```
var net = require("net");
```

Method	Description
net.createServer([options] [, connectionListener])	Creates a new TCP server. The connectionListener argument is aumatically set as a listener for the ‘connection’ event.
net.connect(port[, host][, connectionListener])	Creates a TCP connection to port on host. If host is omitted, ‘localhost’ will be assumed. The connectionListener parameter will be added as an listener for the ‘connect’ event. It is a factory method which returns a new ‘net.Socket’.
net.createConnection(port [, host][, connectionListener])	Creates a TCP connection to port on host. If host is omitted, ‘localhost’ will be assumed. The connectionListener parameter will be added as an listener for the ‘connect’ event. It is a factory method which returns a new ‘net.Socket’.

Node.js – net.Server class

- net.Server class is used to create a TCP or local server. (wk3_09_02_net_server.js)

```
var net = require('net');
var server = net.createServer(function (socket) {
        socket.end('Socket End.');
});
```

Method	Description
server.listen(port[, host][, callback]	Begin accepting connections on the specified port and host. A port value of zero will assign a random port.
server.close(callback)	Finally closed when all connections are ended and the server emits a ‘close’ event.
server.address()	Returns the bound address, the address family name and port of the server as reported by the operating system.

Node.js – other modules

- path module is used for handling and transforming file paths. Path module can be imported using:

  ```
  var path = require("path");
  ```

- dns module is used to do actual DNS lookup as well as to use underlying operating system name resolution functionalities.

  ```
  var dns = require("dns");
  ```

- domain module is used to intercept unhandled error.

  ```
  var domain = require("domain");
  ```

Upload file và lưu vào MongoDB

```
var formidable = require('formidable');
// make sure data directory exists
var dataDir = __dirname + '/data';
var vacationPhotoDir = dataDir + '/vacation-photo';
fs.existsSync(dataDir) || fs.mkdirSync(dataDir);
fs.existsSync(vacationPhotoDir) || fs.mkdirSync(vacationPhotoDir);

function saveContestEntry(contestName, email, year, month, photoPath){
    // TODO...this will come later
}
```

Upload file

Thêm vào file routes/index.js get và post sau:
router.get('/vacation-photo', (req,res)=>{ render view upload photo có thêm ô nhập email })

Định nghĩa VacationPhotoSchema và model của nó
Bổ sung code cho hàm saveContestEntry để lưu vào model
Thực hiện: 10p =>

app.use('/contest', indexRouter);
app.use('/users', usersRouter);

```
app.post('/contest/vacation-photo/:year/:month', function(req, res){
    var form = new formidable.IncomingForm();
    form.parse(req, function(err, fields, files){
        if(err) return res.redirect(303, '/error');
        if(err) {
            res.session.flash = {
                type: 'danger',
                intro: 'Oops!',
                message: 'There was an error processing your submission. ' +
                    'Pelase try again.',
            };
            return res.redirect(303, '/contest/vacation-photo');
        }
        var photo = files.photo;
        var dir = vacationPhotoDir + '/' + Date.now();
        var path = dir + '/' + photo.name;
        fs.mkdirSync(dir);
        fs.renameSync(photo.path, dir + '/' + photo.name);
        saveContestEntry('vacation-photo', fields.email,
            req.params.year, req.params.month, path);
        req.session.flash = {
            type: 'success',
            intro: 'Good luck!',
            message: 'You have been entered into the contest.',
        };
        return res.redirect(303, '/contest/vacation-photo/entries');
    });
});
```

Database Persistence

- We use MongoDB to save data
- Before we get started, we'll need to install the Mongoose module:
  - npm install --save mongoose
- Then we'll add our database credentials to the *credentials.js* file:

```
mongo: {
    development: {
        connectionString: 'your_dev_connection_string',
    },
    production: {
        connectionString: 'your_production_connection_string',
    },
},
```

Database Connections with Mongoose

```
var mongoose = require('mongoose');
var opts = {
    server: {
       socketOptions: { keepAlive: 1 }
    }
};
switch(app.get('env')){
    case 'development':
        mongoose.connect(credentials.mongo.development.connectionString, opts);
        break;
    case 'production':
        mongoose.connect(credentials.mongo.production.connectionString, opts);
        break;
    default:
        throw new Error('Unknown execution environment: ' + app.get('env'));
}
```

Creating Schemas and Models

- To work with Mongo we need to define models
- *Example: models/vacation.js*

```
var vacationSchema = mongoose.Schema({
    name: String,
    slug: String,
    category: String,
    sku: String,
    description: String,
    priceInCents: Number,
    tags: [String],
    inSeason: Boolean,
    available: Boolean,
    requiresWaiver: Boolean,
    maximumGuests: Number,
    notes: String,
    packagesSold: Number,
});
vacationSchema.methods.getDisplayPrice = function(){
    return '$' + (this.priceInCents / 100).toFixed(2);
};
var Vacation = mongoose.model('Vacation', vacationSchema);
module.exports = Vacation;
```

Seeding Initial Data

var Vacation = require('./models/vacation.js');

```
Vacation.find(function(err, vacations){
    if(vacations.length) return;

    new Vacation({
        name: 'Hood River Day Trip',
        slug: 'hood-river-day-trip',
        category: 'Day Trip',
        sku: 'HR199',
        description: 'Spend a day sailing on the Columbia and ' +
            'enjoying craft beers in Hood River!',
        priceInCents: 9995,
        tags: ['day trip', 'hood river', 'sailing', 'windsurfing', 'breweries'],
        inSeason: true,
        maximumGuests: 16,
        available: true,
        packagesSold: 0,
    }).save();
```

Retrieving Data

- To display only vacations that are currently available create a view for the *vacations* page, *views/vacations.handlebars*:

```
<h1>Vacations</h1>
{{#each vacations}}
    <div class="vacation">
        <h3>{{name}}</h3>
        <p>{{description}}</p>
        {{#if inSeason}}
            <span class="price">{{price}}</span>
            <a href="/cart/add?sku={{sku}}" class="btn btn-default">Buy Now!</a>
        {{else}}
            <span class="outOfSeason">We're sorry, this vacation is currently
            not in season.
            {{! The "notify me when this vacation is in season"
                page will be our next task. }}
            <a href="/notify-me-when-in-season?sku={{sku}}">Notify me when
            this vacation is in season.</a>
        {{/if}}
    </div>
{{/each}}
```

Retrieving Data

- Now we can create route handlers that hook it all up:

```
app.get('/vacations', function(req, res){
    Vacation.find({ available: true }, function(err, vacations){
        var context = {
            vacations: vacations.map(function(vacation){
                return {
                    sku: vacation.sku,
                    name: vacation.name,
                    description: vacation.description,
                    price: vacation.getDisplayPrice(),
                    inSeason: vacation.inSeason,
                }
            })
        };
        res.render('vacations', context);
    });
});
```

Adding Data

- Create model object and call save method
- To update data, let check other example: we create the schema and model (*models/vacationInSeasonListener.js*):

```
var vacationInSeasonListenerSchema = mongoose.Schema({
    email: String,
    skus: [String],
});
var VacationInSeasonListener = mongoose.model('VacationInSeasonListener',
    vacationInSeasonListenerSchema);

module.exports = VacationInSeasonListener;
```

Updating Data

```
var VacationInSeasonListener = require('./models/vacationInSeasonListener.js');
```

- In route handler we can call update() like:

```
VacationInSeasonListener.update(
        { email: req.body.email },
        { $push: { skus: req.body.sku } },
        { upsert: true },
        function(err){
            if(err) {
                console.error(err.stack);
                req.session.flash = {
                    type: 'danger',
                    intro: 'Ooops!',
                    message: 'There was an error processing your request.',
                };
                return res.redirect(303, '/vacations');
            }
            req.session.flash = {
                type: 'success',
                intro: 'Thank you!',
                message: 'You will be notified when this vacation is in season.',
            };
            return res.redirect(303, '/vacations');
        }
);
```

Exercise

- Define student models and add some student data

Using MongoDB for Session Storage

- We'll be using a package called session-mongoose (npm install --save session-mongoose)

```
var MongoSessionStore = require('session-mongoose')(require('connect'));
var sessionStore = new MongoSessionStore({ url:
    credentials.mongo.connectionString });

app.use(require('cookie-parser')(credentials.cookieSecret));
app.use(require('express-session')({ store: sessionStore }));
```