ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

503106

ADVANCED WEB PROGRAMMING

CHAPTER 7: ROUTING

LESSON 07 – ROUTING

Content

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.

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

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

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('/', function (req, res) {
    res.send('root'); });
```

- This route path will match requests to /about.

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

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

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

Route paths based on string patterns

- This route path will match acd and abcd.

```
app.get('/ab?cd', function(req, res) { res.send('ab?cd'); });
```

- This route path will match abcd, abbcd, abbbcd, and so on.

```
app.get('/ab+cd', function(req, res) { res.send('ab+cd'); });
```

- This route path will match abcd, abxcd, abRANDOMcd, ab123cd, and so on.

```
app.get('/ab*cd', function(req, res) { res.send('ab*cd'); });
```

- This route path will match /abe and /abcde.

```
app.get('/ab(cd)?e', function(req, res) { res.send('ab(cd)?e'); });
```

Route paths based on regular expressions

- This route path will match anything with an “a” in the route name.

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

- This route path will match butterfly and dragonfly, but not butterflyman, dragonfly man, and so on.

```
app.get(/.*fly$/, function(req, res) {
    res.send('/.*fly$/');
});
```

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', function(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', function (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', function (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], function (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('/', function(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.
- See app2.js

```
app.use('/admin', function(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(function (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(function(req, res, next) {
        res.send('Hello World'); });
// requests will never reach this route
app.get('/', function (req, res) {
        res.send('Welcome');
});
```

Middleware sample

- http://expressjs.com/en/guide/writing-middleware.html
- To load the middleware function, call app.use(), specifying the middleware function.
- See app3.js
- For example, the myLogger middleware function will be loaded before the route to the root path (/).

Middleware sample 2

- http://expressjs.com/en/guide/writing-middleware.html
- Middleware function requestTime add the current time to the req (the request object).
- See app4.js

Using Middleware

- Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.
- ***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 tasks: Execute any code, Make changes to the request and the response objects, End the request-response cycle, Call the next middleware function 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.
- An Express application can use the following types of middleware:
  - Application-level middleware, Router-level middleware, Error-handling middleware, Built-in middleware, Third-party middleware
- NOTE: You can load application-level and router-level middleware with an optional mount path. You can also load a series of middleware functions together, which creates a sub-stack of the middleware system at a mount point.

Application-level Middleware

- Bind application-level middleware to an instance of the app object by using the app.use() and app.METHOD() functions, where METHOD is the HTTP method of the request that the middleware function handles (such as GET, PUT, or POST) in lowercase.
- See app4.js. This example shows a middleware function with no mount path. The function is executed every time the app receives a request.
- This example shows a middleware function mounted on the /user/:id path. The function is executed for any type of HTTP request on the /user/:id path.

```
app.use('/user/:id', function (req, res, next) {
        console.log('Request Type:', req.method);
next(); });
```

- This example shows a route and its handler function (middleware system). The function handles GET requests to the /user/:id path.

```
app.get('/user/:id', function (req, res, next) { res.send('USER'); });
```

Application-level Middleware 2

- Here is an example of loading a series of middleware functions at a mount point, with a mount path. It illustrates a middleware sub-stack that prints request info for any type of HTTP request to the /user/:id path.

```
app.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();
});
```

- Route handlers enable you to define multiple routes for a path. The example below defines two routes for GET requests to the /user/:id path. The second route will not cause any problems, but it will **never get called** because the first route ends the request-response cycle.
- This example shows a middleware sub-stack that handles GET requests to the /user/:id path.

```
app.get('/user/:id', function (req, res, next) {
        console.log('ID:', req.params.id); next();
}, function (req, res, next) { res.send('User Info'); });
app.get('/user/:id', function (req, res, next) { res.end(req.params.id); });
```

Application-level Middleware 3

- To skip the rest of the middleware functions from a router middleware stack, call next('route') to pass control to the next route.
- **NOTE**: next('route') will work only in middleware functions that were loaded by using the app.METHOD() or router.METHOD() functions.
- This example shows a middleware sub-stack that handles GET requests to the /user/:id path. (See app5.js)

```
app.get('/user/:id', function (req, res, next) {
        if (req.params.id == 0) next('route'); // if the user ID is 0, skip to the next route
        // otherwise pass the control to the next middleware function in this stack
        else next();
}, function (req, res, next) {
        res.end('regular'); // render a regular page
});
app.get('/user/:id', function (req, res, next) { res.end('special'); });
```

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. For example: (See app6.js)

```
// invoked for any requests passed to this router
router.use(function(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', function(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 Methods

- router.all(path, [callback, …] callback)
- This method is extremely useful for mapping “global” logic for specific path prefixes or arbitrary matches. For example, if you placed the following route at the top of all other route definitions, it would require that all routes from that point on would require authentication, and automatically load a user. Keep in mind that these callbacks do not have to act as end points; loadUser can perform a task, then call next() to continue matching subsequent routes.

```
router.all('*', requireAuthentication, loadUser);
```

- Another example of this is white-listed “global” functionality. Here the example is much like before, but it only restricts paths prefixed with “/api”:

```
router.all('/api/*', requireAuthentication);
```

Router Methods 2

- **router.METHOD(path, [callback, ...] callback)**
- The router.METHOD() methods provide the routing functionality in Express, where METHOD is one of the HTTP methods, such as GET, PUT, POST, and so on, in lowercase. Thus, the actual methods are router.get(), router.post(), router.put(), and so on.
- You can provide multiple callbacks, and all are treated equally, and behave just like middleware, except that these callbacks may invoke next('route') to bypass the remaining route callback(s). You can use this mechanism to perform pre-conditions on a route then pass control to subsequent routes when there is no reason to proceed with the route matched.

```
router.get('/', function(req, res){
        res.send('hello world');
});
```

Router Methods 3

- **router.use([path], [function, ...] function)**
- Uses the specified middleware function or functions, with optional mount path path, that defaults to “/”.
- This method is similar to app.use(). A simple example and use case is described in app7.js. See app.use() for more information.
- Middleware is like a plumbing pipe: requests start at the first middleware function defined and work their way “down” the middleware stack processing for each path they match.
- The order in which you define middleware with router.use() is very important. They are invoked sequentially, thus the order defines middleware precedence. For example, usually a logger is the very first middleware you would use, so that every request gets logged.

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 (See app5.js), by using router-level middleware (See app8.js).

Organizing Routes

- A simple site may have only a dozen routes or fewer, but a larger site could have hundreds of routes.
- So how to organize your routes? Express is not opinionated about how you organize your routes, so how you do it is limited only by your own imagination.
- Four guiding principles for deciding how to organize your routes:
  - *Use named functions for route handlers*
  - *Routes should not be mysterious*
  - *Route organization should be extensible*
  - *Don’t overlook automatic view-based route handlers*

Declaring Routes in a Module

- The first step to organizing our routes is getting them all into their own module. There are multiple ways to do this. One approach is to make your module a function that returns an array of objects containing “method” and “handler” properties. Then you could define the routes in your application file:

```
var routes = require('./routes.js')();

routes.forEach(function(route){
        app[route.method](route.handler);
})
```

Declaring Routes in a Module

- Or you can pass the app instance to the module, and letting it add the routes.

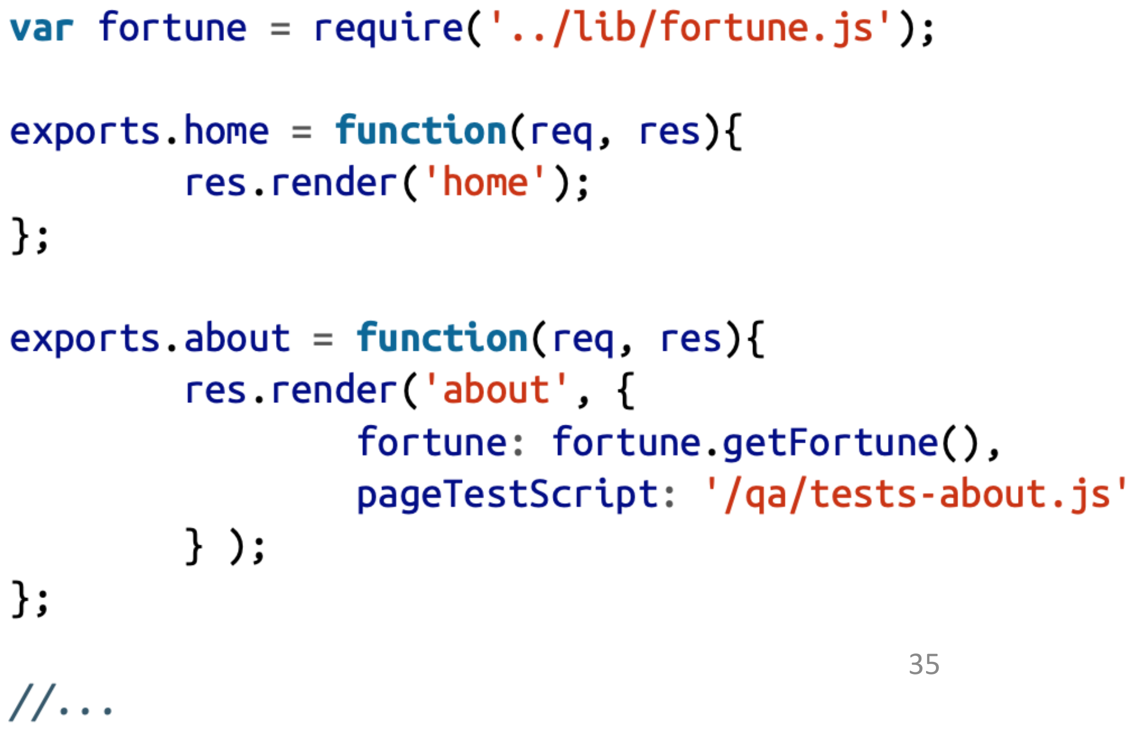
```
module.exports = function(app){

        app.get('/', function(req,res){
                app.render('home');
        }))

        //...

};
```

- Then we simply import our routes:

```
require('./routes.js')(app);
```

Grouping Handlers Logically

- To meet our first guiding principle (use named functions for route handlers), we’ll need somewhere to put those handlers.
- It’s better to somehow group related functionality together. Not only does that make it easier to leverage shared functionality, but it makes it easier to make changes in related methods.
- Consider *handlers/main.js*:



```
var fortune = require('../lib/fortune.js');

exports.home = function(req, res){
        res.render('home');
};

exports.about = function(req, res){
        res.render('about', {
                fortune: fortune.getFortune(),
                pageTestScript: '/qa/tests-about.js'
        } );
};

//...
```

Grouping Handlers Logically

- Now let’s modify *routes.js* to make use of this:

```
var main = require('./handlers/main.js');

module.exports = function(app){

        app.get('/', main.home);
        app.get('/about', main.about);
        //...

};
```

Error-handling Middleware

- Define error-handling middleware functions in the same way as other middleware functions, except with four arguments instead of three, specifically with the signature (err, req, res, next)):

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

- NOTE: Error-handling middleware always takes ***four*** arguments. You must provide four arguments to identify it as an error-handling middleware function. Even if you don’t need to use the next object, you must specify it to maintain the signature. Otherwise, the next object will be interpreted as regular middleware and will fail to handle errors.
- For details about error-handling middleware, see: Error handling.

Automatically Rendering Views

- If your website is very content-heavy without a lot of functionality, you may find it a needless hassle to add a route for every view.
- Let’s say you just want to add the file *views/foo.handlebars* and just magically have it available on the route */foo*.

```
var autoViews = {};
var fs = require('fs');

app.use(function(req,res,next){
    var path = req.path.toLowerCase();
    // check cache; if it's there, render the view
    if(autoViews[path]) return res.render(autoViews[path]);
    // if it's not in the cache, see if there's
    // a .handlebars file that matches
    if(fs.existsSync(__dirname + '/views' + path + '.handlebars')){
        autoViews[path] = path.replace(/^\//, '');
        return res.render(autoViews[path]);
    }
    // no view found; pass on to 404 handler
    next();
});
```

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.

Built-in Middleware 2

- Here is an example of using the express.static middleware function with an elaborate options object:

```
var options = { dotfiles: 'ignore',
        etag: false,
        extensions: ['htm', 'html'],
        index: false, maxAge: '1d',
        redirect: false,
        setHeaders: function (res, path, stat) { res.set('x-timestamp', Date.now()); }
}
app.use(express.static('public', options));
```

- You can have more than one static directory per app:

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

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: (See app9.js)

```
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/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/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. It’s safer to use the absolute path of the directory that you want to serve: app.use('/static', express.static(__dirname + '/public'));