Routing

Routing refers to the definition of application end points (URIs) and how they respond to client requests. For an introduction to routing, see Basic routing.

The following code is an example of a very basic route.

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

// respond with "hello world" when a GET request is made to the homepage
app.get('/', function(req, res) {
  res.send('hello world');
});
```

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 the POST methods to the root of the app.

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

// POST method route
app.post('/', function (req, res) {
  res.send('POST request to the homepage');
});
```

Express supports the following routing methods that correspond to HTTP methods: `get`, `post`, `put`, `head`, `delete`, `options`, `trace`, `copy`, `lock`, `mkcol`, `move`, `purge`, `propfind`, `proppatch`, `unlock`, `report`, `mkactivity`, `checkout`, `merge`, `m-search`, `notify`, `subscribe`, `unsubscribe`, `patch`, `search`, and `connect`.

> To route methods that translate to invalid JavaScript variable names, use the bracket notation. For example, `app['m-search']('/', function ...`

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.

> 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.

> Query strings are not part of the route path.

Here are some examples of 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');
});
```

Here are some examples of 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');
});
```

> The characters ?, +, *, and () are subsets of their regular expression counterparts. The hyphen (-) and the dot (.) are interpreted literally by string-based paths.

Examples of 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 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, as shown in the following examples.

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!');
});
```

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.

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 view 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.

app.route()

You can create chainable route handlers for a route path by using `app.route()`. Because the path is specified at a single location, creating modular routes is helpful, as is reducing redundancy and typos. For more information about routes, see: Router() documentation.

Here is an example of chained route handlers that are defined by using `app.route()`.

```
app.route('/book')
  .get(function(req, res) {
    res.send('Get a random book');
  })
  .post(function(req, res) {
    res.send('Add a book');
  })
  .put(function(req, res) {
    res.send('Update the book');
  });
```

express.Router

Use the `express.Router` class to create modular, mountable route handlers. A `Router` instance is a complete middleware and routing system; for this reason, it is often referred to as a "mini-app".

The following example creates a router as a module, loads a middleware function in it, defines some routes, and mounts the router module on a path in the main app.

Create a router file named `birds.js` in the app directory, with the following content:

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

// middleware that is specific to this router
router.use(function timeLog(req, res, next) {
  console.log('Time: ', Date.now());
  next();
});
// define the home page route
router.get('/', function(req, res) {
  res.send('Birds home page');
});
// define the about route
router.get('/about', function(req, res) {
  res.send('About birds');
});

module.exports = router;
```

Then, load the router module in the app:

```
var birds = require('./birds');
...
app.use('/birds', birds);
```

The app will now be able to handle requests to `/birds` and `/birds/about`, as well as call the `timeLog` middleware function that is specific to the route.