Building Web Applications with

express

# CS572 Modern Web Applications Programming

## Maharishi University of Management

## Department of Computer Science

## Assistant Professor Asaad Saad







PREV NEXT

OUTLINE

Search...

# Express Application Structure

The typical structure of an Express.js app (which is usually app.js file) roughly consists of these parts, in **the order shown**:

1. **Dependencies** ✓
2. **Instantiations** ✓
3. **Configurations** ✓
4. **Middleware**
5. **Routes**
6. **Error Handling**
7. **Bootup** ✓

workflow


They receive the **`request`**, **`response`**, **`next`** objects and can:

- Manipulate req, res objects
- Pass data to each other
- Pass control to error handlers
- Send the response

# Request Handlers

Request handlers are functions (anonymous, named, or methods) with req and res parameters. In addition, we can utilize the **third parameter**, `next()`, for control flow.

```
const filter = function(req, res, next) {
        console.log('filtered');
        return next();
};
const auth = function(req, res) {
        console.log('authenticated');
        res.end(200);
};
app.get('/', filter, auth);
```

# Middleware

A Middleware is Request Handler, a useful pattern that allows developers to reuse code within their applications and even share it with others in the form of NPM modules.

**This Request Handler function accepts three arguments:**

1. **request**
2. **response**
3. **next**

The request (req) and response (res) objects are the same for the subsequent middleware, so you can add properties to them (req.user = 'Asaad') to access them later.

# Using Middleware

To use a middleware, we call the `app.use()` method which accepts:

- One **optional string path**.
- One **mandatory callback function**.

Example: to implement a logger with a date, time, request method, and URL:

```
// To prefix the middleware (mounting)
app.use('/register', function(req, res, next) {
        // ...
        return next();
});
app.use(function(req, res, next) {
        console.log((new Date).toString(), req.method, req.url);
        res.end(200);
});
```

Apply this middleware only to this path

# Built-in Middleware

Express comes with fours built-in middleware (v4.16.0 onwards):

- `express.static()`
- `express.json() – (based on body-parser)`
- `express.urlencoded() – (based on body-parser)`
- `express.Router()`

# express.static()

**static** is a built-in middleware, it enables pass-through requests for static assets.

```
// will setup a middleware on the provided path and return a middleware function
// files will be read from that path and will be streamed immediately to response
app.use(express.static(__dirname + '/public'));

app.use('/css', express.static(__dirname + '/public/css'));
app.use('/img', express.static(__dirname + '/public/images'));
app.use('/js', express.static(__dirname + '/public/javascripts'));
```

Hide your real server file structure

Once Express sees a request to the following paths /css or /img or /js it will stream those resources immediately without looking at the rest of the Routes or other Middleware.

# express.urlencoded()

It parses incoming requests with urlencoded payloads and is based on body-parser.

Returns middleware that only parses urlencoded bodies. This parser supports automatic inflation of gzip and deflate encodings.

A new **body** object containing the parsed data is populated on the request object after the middleware. This object will contain key-value pairs, where the value can be a string or array (when extended is false), or any type (when extended is true).

```
app.use(express.urlencoded({extended: true}));
```

**Note:** This middleware does not support multipart. instead, use **multer**, busboy, formidable, or multiparty.

# Middleware Order

When using middleware, the order in which middleware functions are applied matters, because **this is the order in which they'll be executed**.

# Useful Middleware(s)

| Middleware package | Simple usage | Description |
|---|---|---|
| require('compression') | app.use(compression()) | Gzips transferred data |
| const logger = require('morgan') | app.use(logger('common')) | Keeps track of all the requests |
| require('connect-timeout') | In route: timeout('1s') | Sets a timeout |
| require('serve-favicon') | app.use(favicon(path.join(__dirname , 'public', 'favicon.ico'))); | Change the default favorite icon |
| require('serve-index') | app.use('/browse', serveIndex(path.join(__dirname,'node_modules'),{'icons': true})); | Enables you to create a directory listing based on a particular folder's content |
| require('response-time') | app.use(responseTime(4)); | Adds the X-Response-Time header with the time in milliseconds from the moment the request entered this middleware, it takes number of digits. |

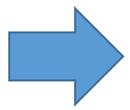
# Routing `app.VERB()`

Each route is defined by a method call on an application object with a URL pattern as the first parameter (regex are supported)

```
app.METHOD(path, [callback...], callback);
```

```
app.get('/api/', function (req, res, next) {
        next();
});

app.get('/api/', function (req, res) {
        res.end();
});
```



```
app.get('/api/',
    function (req, res, next) {
        next();
    },
    function (req, res) {
        res.end();
    });
```

# Routing `app.all()`

The `app.all()` method will execute its specified request handlers on a particular path **regardless of what the HTTP method** of the request is.

```
app.all('*', userAuth);
app.all('/api/*', apiAuth);

var userAuth = function (req, res, next) {
        return next();
});

var apiAuth = function (req, res, next) {
        return next();
});
```

# app.use() vs app.VERB() vs app.all()

| app.use() | app.VERB() | app.all() |
|---|---|---|
| Accepts **one** request handler function | Accepts **multiple** request handler functions | Accepts **multiple** request handler functions |
| Works for **all HTTP verbs** | Works for **specific HTTP verb** | Works for **all HTTP verbs** |
| URL param is optional. Can work for all URL routes or specific one. | Must specify URL param for route or * | Must specify URL param for route or * |

# Using Middlewares Only for certain route

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

var jsonParser = express.json()
var urlencodedParser = express.urlencoded({ extended: false })

app.post('/login', urlencodedParser, function (req, res) {
        res.send('welcome, ' + req.body.username)
})
app.post('/api/users', jsonParser, function (req, res) {
        // create user in req.body
})
```

https://www.npmjs.org/package/body-parser

# The `return` keyword

The **`return`** keyword is essential, because if we don't use it the application will jump and later we will **comes back to the original execution stack**.

For example, the following is probably a bad idea because after we call next(), the flow goes on and tries to render the page even when there is an error and/or we don't have any users

```
function showPosts (req, res, next) {
        req.db.get('users').find({}, function(e, users) {
                if (e) next(e); // bad
                if (!users) next(new Error('No users to display.')); // bad
                res.json(users);
        });
};
app.get('/posts', showPosts);
```

# next() vs next('route')

```
app.get('/api/', function(request, response, next) {
        // Filter Input
        return next();
}), function(request, response, next) {
        // Authorization
        if(authenticated) {
                request.auth = true;
                return next();
        }else{
                return next('route');
        }
}), function(request, response) {
        // Output the result of the database
        res.json(response.user);
});
app.get('/auth/', function(request, response, next) {
      // Will check here, but route does not match, check further..
}
```

You can also use `next()` to jump to the next route if you have multiple routes matching the same URL.

We can bypass the rest of the callbacks in the chain by calling next('route')

# next()

| next() | next('route') | next(somethingElse) |
| --- | --- | --- |
| Go to next request handler function (middleware, route), could be in the same URL route | Skip current route handlers in chain and go to next one | Go to Error handler |

# express.Router() Middleware

The Router class is a mini Express.js application that has only middleware and routes. This is useful for **abstracting modules** based on the business logic that they perform.

myRoute.js

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

router.post('/posts/', function(request, response){ })
router.get('/posts/', function(request, response){ })

module.exports = router;
```

Where options is an object :

- **caseSensitive**: Boolean
- **strict**: Boolean

```
// app.js
app.use('/blog', require("myRoute.js"));
```

# Request Object

- `request.params` Parameters middleware
- `request.query` Extract query string parameter
- `request.route` Return currently-matched route
- `request.body` After parsing the body payload

http://expressjs.com/en/api.html#req

# Request Object Examples

- `request.query`

Optional

```
http://localhost:3000/search?q=nodejs&lang=eng
{"q": "nodejs", "lang": "eng"}
```

- `request.params`

Mandatory

```
app.get('/api/:id/:name/:city',
        function(req, res) {
                res.end(req.params);
        }); // //
http://localhost:3000/api/1/Asaad/Fairfield
{ id: 1, name: 'Asaad', city: 'Fairfield' }
```

- `request.body`

```
app.use(express.urlencoded());
app.post('/api', function(req, res){
        res.end(req.body);
});
$ curl -i http://localhost:3000/api -d 'name=Asaad&lastname=Saad'
{ name: 'Asaad', lastname: 'Saad' }
```

# Other Request Header Properties

`request.get(headerKey)` Value for the header key
`request.accepts(type)` Checks if the type is accepted
`request.acceptsLanguage(language)` Checks language
`request.acceptsCharset(charset)` Checks charset
`request.is(type)` Checks the type
`request.ip` IP address
`request.ips` IP addresses (with trust-proxy on)
`request.path` URL path
`request.host` Host without port number
`request.fresh` Checks freshness
`request.stale` Checks staleness
`request.xhr` True for AJAX-y requests
`request.protocol` Returns HTTP protocol
`request.secure` Checks if protocol is https
`request.subdomains` Array of subdomains
`request.originalUrl` Original URL

# Response Object

- `response.redirect(status, url)` Redirect request
- `response.redirect(url)` Redirect to new path with status 302
- `response.send(status, data)` Send response
- **`response.json(status, data)` Send JSON and force proper headers**
- `response.jsonp(data)` JSON data will be wrapped in JS function call
- `response.sendfile(path, options, callback)` Send a file
- `response.status(status)` Send status code

http://expressjs.com/en/api.html#res

# Response Object Examples

```
app.get('/render-title', function(req, res) {
        res.status(200).send(`Welcome`)
});

// a common way to send status number
response.status(200).json({ name: "Asaad" })
```

The `response.send()` method conveniently outputs any data application thrown at it (such as strings, JavaScript objects, and even Buffers) with automatically generated proper HTTP headers (Content-Length, ETag, or Cache-Control).

# Handling Errors Problem!

The standard way to error handling and debugging looks like this:

```
try {
        throw new Error('Fail!');
} catch (e) {
        console.log('Custom Error: ' + e.message);
}
```

**Custom Error: Fail!**

Let’s add some asynchronous code:

```
try {
        setTimeout(function () { throw new Error('Fail');}, 1000));
} catch (e) {
        console.log('Custom Error: ' + e.message);
}
```

After 1 sec we will see: **unhandled error (“Error: Fail”)**

`try/catch` fails at asynchronous errors. Use `try/catch` only for synchronous JavaScript/Node.js code only.

# Error Handling in Express

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

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

Responses from within a middleware function can be in any format that you prefer, such as an HTML error page, a simple message, or a JSON string.

**IMPORTANT:** You define error-handling middleware last, after other middleware and routes calls.

# Multiple Error Handlers

For organizational (and higher-level framework) purposes, you can define several error-handling middleware functions:

```
// ... routes here ...
app.use(logErrors);
app.use(clientErrorHandler);
app.use(errorHandler);

function logErrors(err, req, res, next) {
        console.error(err.stack);
        next(err);
}
function clientErrorHandler(err, req, res, next) {
        if(req.xhr){res.status(500).send({error:'Something went wrong!' })}
        else { next(err); }
}
function errorHandler(err, req, res, next) {
        res.status(500);
        res.json({ err });
}
```

There are all error handlers because they are middlewares that have (err,req,res,next) signature

Print error to console

Will work only for XHR requests

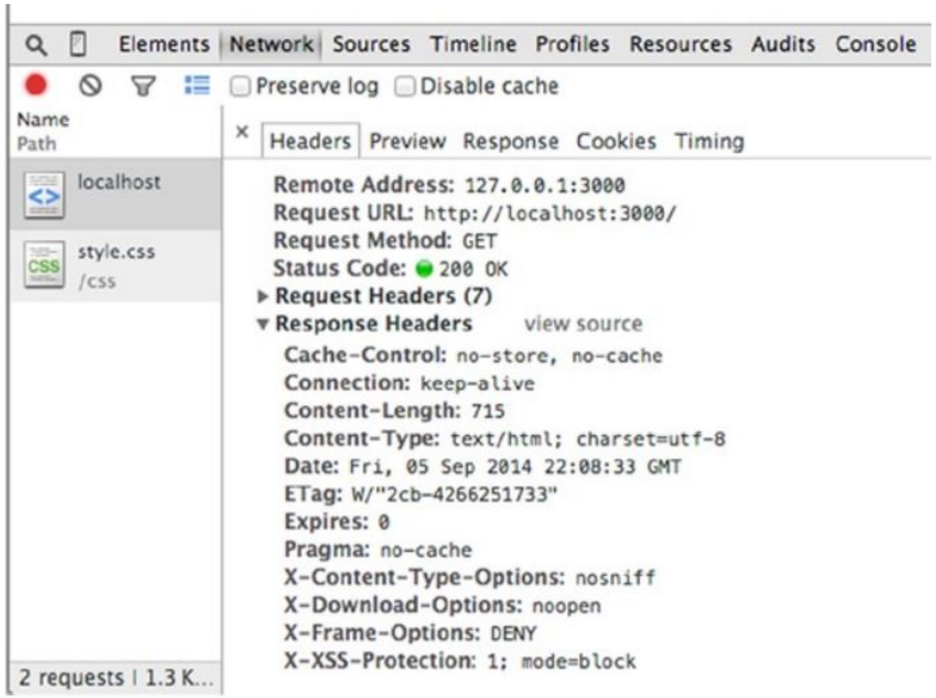
Send the error page response

# Helmet

`Helmet` is a collection of security related middleware that provides most of the security headers

https://www.npmjs.org/package/helmet

```
var helmet = require('helmet');
app.use(helmet());
```




Further reading: Seven Web Server HTTP Headers that Improve Web Application Security

# Accepting CORS in Express

```
app.use(function(req, res, next) {
    res.header("Access-Control-Allow-Origin", "*");
    res.header('Access-Control-Allow-Methods','GET,PUT,POST,DELETE,PATCH,OPTIONS');
    res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With,
                            Content-Type, Accept, Authorization, Content-Length");
    next();
});
```

# Accepting CORS in Express

```
$ npm install cors --save

var express = require('express')
var cors = require('cors')
var app = express()

app.use(cors())
```