A short guide to Connect Middleware

This guide will introduce you to connect and the concept of middleware. If you are familiar with middleware from other systems such as WSGI or Rack, you can probably skip the middle section and go straight to the Examples.

What is connect?

From the README:

> *Connect is an extensible HTTP server framework for node, providing high performance "plugins" known as middleware.*

More specifically, connect wraps the Server, ServerRequest, and ServerResponse objects of node.js' standard `http` module, giving them a few nice extra features, one of which is allowing the `Server` object to `use` a stack of *middleware*.

What is middleware?

Simply put, middleware are functions that handle requests. A server created by `connect.createServer` can have a stack of middleware associated with it. When a request comes in, it is passed off to the first middleware function, along with a wrapped ServerResponse object and a `next` callback. Each middleware can decide to respond by calling methods on the response object, and/or pass the request off to the next layer in the stack by calling `next()`. A simple no-op middleware would look like this:

```
function uselessMiddleware(req, res, next) { next() }
```

A middleware can also signal an error by passing it as the first argument to `next`:

```
// A middleware that simply interrupts every request
function worseThanUselessMiddleware(req, res, next) {
  next("Hey are you busy?")
}
```

When a middleware returns an error like this, all subsequent middleware will be skipped until connect can find an error handler. (See Error Handling for an example).

To add a middleware to the stack of middleware for a server, we `use` it like so:

```
connect = require('connect')
stephen = connect.createServer()
stephen.use(worseThanUselessMiddleware)
```

Finally, you can also specify a path prefix when adding a middleware, and the middleware will only be asked to handle requests that match the prefix:

```
connect = require('connect')
bob = connect.createServer()
bob.use('/attention', worseThanUselessMiddleware)
```

What can I use it for?

Plenty of things! Common examples are logging, serving static files, and error handling. Note that all three of the above (and more) are standard middleware included with connect itself, so you probably won't need to implement them yourself. Another common middleware is routing requests to controllers or callback methods (for this and *way* more, check out TJ Holowaychuk's express).

Really, you can use middleware anywhere you might want to have some sort of generic request handling logic applied to all requests. For example, in my Lazorse project, request routing and response rendering are separate middleware, because it's designed around building API's that may want to use different rendering backends depending on the clients `Accept` header.

Examples

URL based authentication policy

Authentication policy is often app-specific, so this middleware wraps the `connect.basicAuth` middleware with a list of URL patterns that should be authenticated.

```
function authenticateUrls(urls /* basicAuth args*/) {
  basicAuthArgs = Array.slice.call(arguments, 1)
  basicAuth = require('connect').basicAuth.apply(basicAuthArgs)
  function authenticate(req, res, next) {
    // Warning! assumes that urls is amenable to iteration
    for (var pattern in urls) {
      if (req.path.match(pattern)) {
        basicAuth(req, res, next)
        return
      }
    }
    next()
  }
  return authenticate
}
```

See the basicAuth docs for the various options it takes.

Role base authorization

```
// @urls - an object mapping patterns to lists of roles.
// @roles - an object mapping usernames to lists of roles
function authorizeUrls(urls, roles) {
  function authorize(req, res, next) {
    for (var pattern in urls) {
      if (req.path.match(pattern)) {
        for (var role in urls[pattern]) {
          if (users[req.remoteUser].indexOf(role) < 0) {
            next(new Error("unauthorized"))
            return
          }
        }
      }
    }
    next()
  }
  return authorize
}
```

These examples demonstrate how middleware can help isolate cross-cutting request logic into modules that can be swapped out. If we later decide to replace basicAuth with OAuth, the only dependency our authorization module has is on the `.remoteUser` property.

Error handling

Another common cross-cutting concern is error handling. Again, connect ships with an handler that will serve up nice error responses to clients, but finding out about errors in production via your customers is usually not a good business practice ;). To help with that, let's implement a simple error notification middleware using node_mailer:

```
email = require('node_mailer')

function emailErrorNotifier(generic_opts, escalate) {
  function notifyViaEmail(err, req, res, next) {
    if (err) {
      var opts = {
        subject: "ERROR: " + err.constructor.name,
        body: err.stack,
      }
      opts.__proto__ = generic_opts
      email.send(opts, escalate)
    }
    next()
  }
}
```

Connect checks the arity of middleware functions and considers the function to be an error handler if it is 4, meaning errors returned by earlier middleware will be passed as the first argument for inspection.

Putting it all together

Here's a simple app that combines all of the above middleware:

```
private_urls = {
  '^/attention': ['coworker', 'girlfriend'],
  '^/bank_balance': ['me'],
}
```

```
roles = {
  stephen: ['me'],
  erin:    ['girlfriend'],
  judd:    ['coworker'],
  bob:     ['coworker'],
}

passwords = {
  me:    'doofus',
  erin: 'greatest',
  judd: 'daboss',
  bob:   'anachronistic discomBOBulation',
}

function authCallback(name, password) { return passwords[name] === password }

stephen = require('connect').createServer()
stephen.use(authenticateUrls(private_urls, authCallback))
stephen.use(authorizeUrls(private_urls, roles))
stephen.use('/attention', worseThanUselessMiddleware)
stephen.use(emailErrorNotifier({to: 'stephen@betsmartmedia.com'}))
stephen.use('/bank_balance', function (req, res, next) {
  res.end("Don't be Seb-ish")
})
stephen.use('/', function (req, res, next) {
  res.end("I'm out of coffee")
})
```

But I want to use objects!

Perfectly reasonable, pass an object with a `handle` method to `use` and connect will call that method in the exact same manner.

Contact

Please submit all feedback, flames and compliments to Stephen.