503106

ADVANCED WEB PROGRAMMING

CHAPTER 9: STATIC CONTENT

LESSON 09 – STATIC CONTENT

Performance Considerations

- The two primary perfor- mance considerations are *reducing the number of requests* and *reducing content size*.
  - Combining resources
  - Reduce the size of static resources.

Static Mapping

- you want to be able to write <img src="***/img/meadowlark_logo.png***" alt="Meadowlark Travel Logo">, not <img src="**//s3-us- west-2.amazonaws.com/meadowlark/img/meadowlark_logo-3.png**" alt="Meadow lark Travel Logo">
- we wish to map less specific paths (*/img/ meadowlark_logo.png*) to more specific paths (*//s3-us-west-2.amazonaws.com/mead-owlark/img/meadowlark_logo-3.png*)
- Let's create a file called *lib/static.js*:

```
var baseUrl = '';

exports.map = function(name){
        return baseUrl + name;
}
```

Static Resources in Views

- We can create a Handlebars helper

```
// set up handlebars view engine
var handlebars = require('express3-handlebars').create({
    defaultLayout:'main',
    helpers: {
        static: function(name) {
            return require('./lib/static.js').map(name);
        }
    }
});
```

- We added a Handlebars helper called static

```
<header><img src="{{static '/img/logo.jpg'}}"
    alt="Meadowlark Travel Logo"></header>
```

Static Resources in CSS

Cài package cần:
npm install -g grunt-cli
npm install grunt

```
module.exports = function(grunt) {
    grunt.initConfig({
        less:{
            development:{
                files:{
                    "public/css/main.css":"less/main.less",
                }
            }
        }
    });
    grunt.loadNpmTasks('grunt-contrib-less');
};
```

rouind.png");

. We'll use a Grunt

t of Grunt tasks to

ate:

" href="{{static /css/main.css}}">

- grunt less to create CSS file

</head>

Static Resources in CSS

- Link our static mapper as a LESS custom function. This can all be accomplished in *Gruntfile.js*:
- Now all we have to do is modify our LESS file, *less/main.less*:

```
body {
        background-image: static("/img/background.png");
}
```

```
less: {
    development: {
        options: {
            customFunctions: {
                static: function(lessObject, name) {
                    return 'url("' +
                        require('./lib/static.js').map(name.value) +
                        '")';
                }
            }
        },
        files: {
            'public/css/main.css': 'less/main.less',
        }
    }
}
```

Static Resources in Server-Side JavaScript

- Using our static mapper in server-side JavaScript is really easy, as we've already written a module to do our mapping. For example, we want to change logo on 19 – Nov every month (Boss birthday)

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

app.use(function(req, res, next){
        var now = new Date();
        res.locals.logoImage = now.getMonth()==11 && now.getDate()==19 ?
                static('/img/logo_bud_clark.png') :
                static('/img/logo.png');
        next();
});
```

- Then link in view:

```
<header><img src="{{logoImage}}" alt="Meadowlark Travel Logo"></header>
```

Static Resources in Client-Side JavaScript

- The solution is just to do the mapping on the server, and set custom JavaScript variables.
- Our two images are called */img/shop/cart_empty.png* and */img/shop/cart_full.png*. Without mapping, we might use something like this:

```
$(document).on('meadowlark_cart_changed'){
        $('header img.cartIcon').attr('src', cart.isEmpty() ?
                '/img/shop/cart_empty.png' : '/img/shop/cart_full.png' );
}
```

- What happen when we change static root?

Static Resources in Client-Side JavaScript

- In view we can do

```
<!-- ... -->
<script>
    var IMG_CART_EMPTY = '{{static '/img/shop/cart_empty.png'}}';
    var IMG_CART_FULL = '{{static '/img/shop/cart_full.png'}}';
</script>
```

- Then our jQuery simply uses those variables:

```
$(document).on('meadowlark_cart_changed', function(){
        $('header img.cartIcon').attr('src', cart.isEmpty() ?
                IMG_CART_EMPTY : IMG_CART_FULL );
});
```

- If you do a lot of image swapping on the client side, you'll probably want to consider organizing all of your image variables in an object

```
<!-- ... -->
<script>
        var static = {
                IMG_CART_EMPTY: '{{static '/img/shop/cart_empty.png'}}',
                IMG_CART_FULL: '{{static '/img/shop/cart_full.png'}}
        }
</script>
```

Bundling and Minification

- In an effort to reduce HTTP requests *and* reduce the data sent over the wire, “bundling and minification” has become popular. Bundling takes like files (CSS or JavaScript) and bundles multiple files into one (thereby reducing HTTP requests). Minification removes anything unnecessary from your source, such as whitespace (outside of strings), and it can even rename your variables to something shorter.
- For example, we have *public/js/contact.js, public/js/cart.js, less/main.less and less/cart.less*
- Now in *Gruntfile.js* add it to the list of LESS files to compile:

```
files: {
    'public/css/main.css': 'less/main.less',
    'public/css/cart.css': 'less/cart.css',
}
```

Bundling and Minification

- Let's go ahead and install those modules now:

  npm install --save-dev grunt-contrib-uglify
  npm install --save-dev grunt-contrib-cssmi
  npm install --save-dev grunt-hashres

- Then load these tasks in the Gruntfile

```
[
    // ...
    'grunt-contrib-less',
    'grunt-contrib-uglify',
    'grunt-contrib-cssmin',
    'grunt-hashres',
].forEach(function(task){
    grunt.loadNpmTasks(task);
});
```

- And set up the tasks:

```
grunt.initConfig({
    // ...
    uglify: {
        all: {
            files: {
                'public/js/meadowlark.min.js': ['public/js/**/*.js']
            }
        }
    },
    cssmin: {
        combine: {
            files: {
                'public/css/meadowlark.css': ['public/css/**/*.css',
                    '!public/css/meadowlark*.css']
            }
        },
        minify: {
            src: 'public/css/meadowlark.css',
            dest: 'public/css/meadowlark.min.css',
        }
    },
    hashres: {
        options: {
            fileNameFormat: '${name}.${hash}.${ext}'
        },
        all: {
            src: [
                'public/js/meadowlark.min.js',
                'public/css/meadowlark.min.css',
            ],
            dest: [
                'views/layouts/main.handlebars',
            ]
        },
    }
});
};
```

Bundling and Minification

- Let's modify our layout file:

```
        <!-- ... -->
        <script src="http://code.jquery.com/jquery-2.0.2.min.js"></script>
        <script src="{{static '/js/meadowlark.min.js'}}"></script>
        <link rel="stylesheet" href="{{static '/css/meadowlark.min.css'}}">
    </head>
```

- hashres will generate a hash of the file (a mathematical fingerprinting) and append it to the file. So now, instead of */js/meadowlark.min.js*, you'll have */js/meadowlark.min.62a6f623.js*

Bundling and Minification

- So now let's give it a try. It's important that we do things in the right order, because these tasks have dependencies:
  - grunt less
  - grunt cssmin
  - grunt uglify
  - grunt hashres
- That's a lot of work every time we want to change our CSS or JavaScript, so let's set up a Grunt task so we don't have to remember all that. Modify *Gruntfile.js*:

```
grunt.registerTask('default', ['cafemocha', 'jshint', 'exec']);
grunt.registerTask('static', ['less', 'cssmin', 'uglify', 'hashres']);
```

- Now all we have to do is type ***grunt static***

Skipping Bundling and Minification in Development Mode

- One problem with bundling and minification is that it makes frontend debugging all but impossible. We need to disable bundling and minification in development mode
- First install connect-bundle then create *config.js and* modify *views/layouts/main.handlebars*:

```
<!-- ... -->
{{#each _bundles.css}}
    <link rel="stylesheet" href="{{static .}}">
{{/each}}
{{#each _bundles.js.head}}
    <script src="{{static .}}"></script>
{{/each}}
</head>
```

```
module.exports = {
    bundles: {

        clientJavaScript: {
            main: {
                file: '/js/meadowlark.min.js',
                location: 'head',
                contents: [
                    '/js/contact.js',
                    '/js/cart.js',
                ]
            }
        },

        clientCss: {
            main: {
                file: '/css/meadowlark.min.css',
                contents: [
                    '/css/main.css',
                    '/css/cart.css',
                ]
            }
        }
    }
}
```

Skipping Bundling and Minification in Development Mode

- Finally, modify *Gruntfile.js*
- Now you can run grunt static; you'll see that *config.js* has been updated

```
hashres: {
    options: {
        fileNameFormat: '${name}.${hash}.${ext}'
    },
    all: {
        src: [
            'public/js/meadowlark.min.js',
            'public/css/meadowlark.min.css',
        ],
        dest: [
            'config.js',
        ]
    },
}
```

Security

HTTPS

- The first step in providing secure services is using HTTP Secure (HTTPS). The nature of the Internet makes it possible for a third party to intercept packets being transmitted between clients and servers. HTTPS encrypts those packets, making it extremely difficult for an attacker to get access to the information being transmitted.
- The HTTPS protocol is based on the server having a *public key certificate*, sometimes called an SSL certificate.
- The current standard format for SSL certificates is called X. 509.

Generating Your Own Certificate

- Generating your own certificate is easy, but generally suitable only for development and testing purposes (and possibly for intranet deployment).
- You'll need an OpenSSL

Platform	Instructions
OS X	`brew install openssl`
Ubuntu, Debian	`sudo apt-get install openssl`
Other Linux	Download from *http://www.openssl.org/source/*; extract tarball and follow instructions
Windows	Download from *http://gnuwin32.sourceforge.net/packages/openssl.htm*

- Then run command below to create it:

openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout meadowlark.pem -out meadowlark.crt

Generating Your Own Certificate

- The result of the command is two files, *meadowlark.pem* and *meadowlark.crt*. The PEM (Privacy-enhanced Electronic Mail) file is your private key, and should not be made available to the client. The CRT file is the self-signed certificate that will be sent to the browser to establish a secure connection.
- Alternatively, there are websites that will provide free self-signed certificates, such as *http://www.selfsignedcertificate.com*.

Enabling HTTPS for Your Express App

- Put your private key and SSL cert in a subdirectory called *ssl t*hen you just use the https module instead of http

```
var https = require('https');   // usually at top of file

var options = {
        key: fs.readFileSync(__dirname + '/ssl/meadowlark.pem');
        cert: fs.readFileSync(__dirname + '/ssl/meadowlark.crt');
};

https.createServer(options, app).listen(app.get('port'), function(){
        console.log('Express started in ' + app.get('env') +
                ' mode on port ' + app.get('port') + '.');
});
```

- You can now connect to ***https****://localhost:3000*. If you try to connect to ***http****://localhost:3000*, it will simply time out.

Cross-Site Request Forgery

- Cross-site request forgery (CSRF) attacks exploit the fact that users generally trust their browser and visit multiple sites in the same session. In a CSRF attack, script on a malicious site makes requests of another site: if you are logged in on the other site, the malicious site can successfully access secure data from another site.
- To prevent CSRF attacks, you must have a way to make sure a request legitimately came from your website. The way we do this is to pass a unique token to the browser. When the browser then submits a form, the server checks to make sure the token matches.

Cross-Site Request Forgery

- The csurf middleware will handle the token creation and verification for you; all you'll have to do is make sure the token is included in requests to the server. Install the csurf middleware (npm install --save csurf), then link it in and add a token to res.locals:

```
// this must come after we link in cookie-parser and connect-session
app.use(require('csurf')());
app.use(function(req, res, next){
        res.locals._csrfToken = req.csrfToken();
        next();
});
```

- Now on all of your forms (and AJAX calls), you'll have to provide a field called _csrf, which must match the generated token.

```
<form action="/newsletter" method="POST">
        <input type="hidden" name="_csrf" value="{{_csrfToken}}">
        Name: <input type="text" name="name"><br>
```

Authentication

Storing Users in Your Database

- Whether or not you rely on a third party to authenticate your users, you will want to store a record of users in your own database.
- So let's create a model for our users, *models/user.js*:

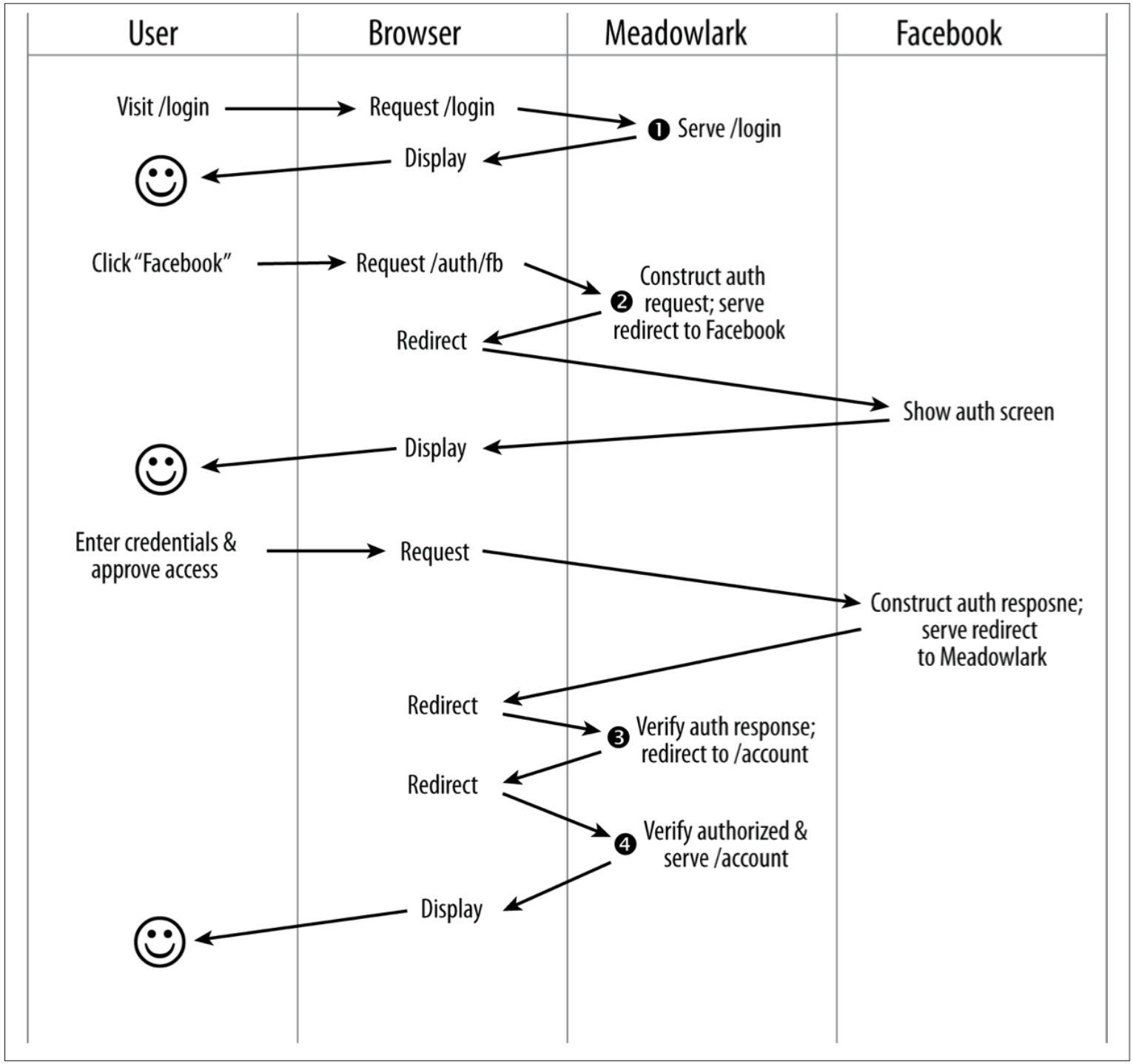
```
var mongoose = require('mongoose');

var userSchema = mongoose.Schema({
        authId: String,
        name: String,
        email: String,
        role: String,
        created: Date,
});

var User = mongoose.model('User', userSchema);
module.exports = User;
```

Third-Party Authentication

- Third-party authentication takes advantage of the fact that pretty much everyone on the Internet has an account on at least one major service, such as Google, Facebook, Twitter, or LinkedIn. All of these services provide a mechanism to authenticate and identify your users through their service.

Passport

- Passport is a very popular and robust authentication module for Node/Express. It is not tied to any one authentication mechanism; rather, it is based on the idea of pluggable authentication *strategies* (including a local strategy if you don't want to use third-party authentication).
- When you use Passport, there are four steps that your app will be responsible for. Consider a more detailed view of the third-party authentication flow



Setting up Passport

- To keep things simple, we'll start with a single authentication provider. Arbitrarily, we'll choose Facebook. Before we can set up Passport and the Facebook strategy, we'll need to do a little configuration in Facebook. For Facebook authentication, you'll need a *Facebook app*.
- Then let's install Passport, and the Facebook authentication strategy:

```
npm install --save passport passport-facebook
```

Setting up Passport

- We'll start with the imports and two methods that Passport requires, serializeUser and deserializeUser.
- Passport uses serializeUser and deserializeUser to map requests to the authenticated user, allowing you to use whatever storage method you want. In our case, we are only going to store the MongoDB-assigned ID (the _id property of User model instances) in the session.

```
var User = require('../models/user.js'),
        passport = require('passport'),
        FacebookStrategy = require('passport-facebook').Strategy;

passport.serializeUser(function(user, done){
        done(null, user._id);
});

passport.deserializeUser(function(id, done){
        User.findById(id, function(err, user){
                if(err || !user) return done(err, null);
                done(null, user);
        });
});
```

Setting up Passport

- Once these two methods are implemented, as long as there is an active session, and the user has successfully authenticated, req.session.passport.user will be the corresponding User model instance.
- Next, we're going to choose what to export. To enable Passport's functionality, we'll need to do two distinct activities: initialize Passport and register routes that will handle authentication and the redirected callbacks from our third-party authentication services.

Setting up Passport

- Also, since we need to link the Passport middleware into our application, a function is an easy way to pass in the Express application object:
- Before we get into the details of the init and registerRoutes methods, let's look at how we'll use this module

```
var auth = require('./lib/auth.js')(app, {
        providers: credentials.authProviders,
        successRedirect: '/account',
        failureRedirect: '/unauthorized',
});
// auth.init() links in Passport middleware:
auth.init();

// now we can specify our auth routes:
auth.registerRoutes();
```

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

        // if success and failure redirects aren't specified,
        // set some reasonable defaults
        if(!options.successRedirect)
                options.successRedirect = '/account';
        if(!options.failureRedirect)
                options.failureRedirect = '/login';

        return {

                init: function() { /* TODO */ },

                registerRoutes: function() { /* TODO */ },

        };
};
```

Setting up Passport

- We'll need to add the authProviders property to *credentials.js*:

```
module.exports = {
        mongo: {
                //...
        },

        authProviders: {
            facebook: {
                development: {
                    appId: 'your_app_id',
                    appSecret: 'your_app_secret',
                },
            },
        },
}
```

```
init: function() {
    var env = app.get('env');
    var config = options.providers;

    // configure Facebook strategy
    passport.use(new FacebookStrategy({
        clientID: config.facebook[env].appId,
        clientSecret: config.facebook[env].appSecret,
        callbackURL: '/auth/facebook/callback',
    }, function(accessToken, refreshToken, profile, done){
        var authId = 'facebook:' + profile.id;
        User.findOne({ authId: authId }, function(err, user){
            if(err) return done(err, null);
            if(user) return done(null, user);
            user = new User({
                authId: authId,
                name: profile.displayName,
                created: Date.now(),
                role: 'customer',
            });
            user.save(function(err){
                if(err) return done(err, null);
                done(null, user);
            });
        });
    }));

    app.use(passport.initialize());
    app.use(passport.session());
},
```

Setting up Passport

- The last thing we have to do is create our registerRoutes method

```
registerRoutes: function(){
    // register Facebook routes
    app.get('/auth/facebook', function(req, res, next){
        passport.authenticate('facebook', {
                callbackURL: '/auth/facebook/callback?redirect=' +
                        encodeURIComponent(req.query.redirect),
        })(req, res, next);
    });
    app.get('/auth/facebook/callback', passport.authenticate('facebook',
        { failureRedirect: options.failureRedirect },
        function(req, res){
                // we only get here on successful authentication
                res.redirect(303, req.query.redirect || options.successRedirect);
                }
        ));
},
```

Setting up Passport

- Now we have the path */auth/facebook*; visiting this path will automatically redirect the visitor to Facebook's authentication screen
- Let's look at our /account handler to see how it checks to make sure the user is authenticated

```
app.get('/account', function(req, res){
        if(!req.session.passport.user)
                return res.redirect(303, '/unauthorized');
        res.render('account');
});
```

Role-Based Authorization

- Let’s say we only want customers to see their account views (employees might have an entirely different view where they can see user account information).
- Let’s create a function called customerOnly that will allow only customers:

```
function customerOnly(req, res){
        var user = req.session.passport.user;
        if(user && req.role==='customer') return next();
        res.redirect(303, '/unauthorized');
}
```

Role-Based Authorization

- Here's how easy it is to put these functions to use:
- Write your employeeOnly function.

```
// customer routes

app.get('/account', customerOnly, function(req, res){
        res.render('account');
});
app.get('/account/order-history', customerOnly, function(req, res){
        res.render('account/order-history');
});
app.get('/account/email-prefs', customerOnly, function(req, res){
        res.render('account/email-prefs');
});

// employer routes
app.get('/sales', employeeOnly, function(req, res){
        res.render('sales');
});
```

Role-Based Authorization

- It should be clear that role-based authorization can be as simple or as complicated as you wish. For example, what if you want to allow multiple roles? You could use the following function and route:

```
function allow(roles) {
        var user = req.session.passport.user;
        if(user && roles.split(',').indexOf(user.role)!==-1) return next();
        res.redirect(303, '/unauthorized');
}

app.get('/account', allow('customer,employee'), function(req, res){
        res.render('account');
});
```

Adding Additional Authentication Providers

- We want to authenticate with Google. In the case of Google, we don't even need to get an app secret or modify our *authProviders.js* file. We simply add the following to the init method of *lib/auth.js*:

```
passport.use(new GoogleStrategy({
    returnURL: 'https://' + host + '/auth/google/return',
    realm: 'https://' + host +'/',
}, function(identifier, profile, done){
    var authId = 'google:' + identifier;
    User.findOne({ authId: authId }, function(err, user){
        if(err) return done(err, null);
        if(user) return done(null, user);
        user = new User({
            authId: authId,
            name: profile.displayName,
            created: Date.now(),
            role: 'customer',
        });
        user.save(function(err){
            if(err) return done(err, null);
            done(null, user);
        });
    });
}));
```

Adding Additional Authentication Providers

- And the following to the registerRoutes method:

```
// register Google routes
app.get('/auth/google', function(req, res, next){
        passport.authenticate('google', {
                callbackURL: '/auth/google/callback?redirect=' +
                        encodeURIComponent(req.query.redirect),
        })(req, res, next);
});
app.get('/auth/google/callback', passport.authenticate('google',
        { failureRedirect: options.failureRedirect },
        function(req, res){
                // we only get here on successful authentication
                res.redirect(303, req.query.redirect || options.successRedirect);
        }
));
```

LocalStrategy

```
var passport = require('passport')
var LocalStrategy = require('passport-local').Strategy;

passport.use(new LocalStrategy(
   function(username, password, done) {
      User.findOne({ username: username }, function (err, user) {
         if (err) { return done(err); }
         if (!user)
            return done(null, false, { message: 'Incorrect username.' });
         if (!user.validPassword(password))
            return done(null, false, { message: 'Incorrect password.' });
         return done(null, user);
                  });
   }
));
```

LocalStrategy

```
passport.use(new LocalStrategy({
  usernameField: 'email’,
  passwordField: 'passwd’
  },
  function(username, password, done) {
    // ...
  })
);

app.post('/login', passport.authenticate('local’,
      { successRedirect: '/', failureRedirect: '/login' }));
```

View:

```
<form action="/login" method="post">
  <div>
    <label>Username:</label>
    <input type="text" name="username"/>
  </div>
  <div>
    <label>Password:</label>
    <input type="password" name="password"/>
  </div>
  <div>
    <input type="submit" value="Log In"/>
  </div>
</form>
```

Q & A