Emerging Technologies

COMP-308

Winter 2018

Lesson 6 Review

- **REST API with Express and Mongoose**
  - **HTTP methods: POST, GET, PUT, DELETE**
- **Finding multiple user documents** using find() method:

  User.**find**({}, (err, user) => {…});
- Reading a single user document using findOne():

  User.**findOne**({_id: id}, (err, user) => {…});
- Using request paths with parameters:
  - **param** middleware pattern

  app.route('/users/:userId').get(users.read);

  app.**param**('userId', users.userByID);
- **Saving documents**
  - save() method
- Updating an existing user document
  - update(), findOneAndUpdate(), and findByIdAndUpdate()

  User.**findByIdAndUpdate**(req.user.id, req.body, function(err, user) {…});
  - Use **PUT** method in routing code

Lesson 6 Review

- **Deleting an existing user document**
  - remove(), findOneAndRemove(), and findByIdAndRemove() methods

  req.user.**remove**(function(err){…});

  - Use **DELETE** method in routing code
- Reading a single user document using findOne():

  User.**findOne**({_id: id}, (err, user) => {…});
- Using schema modifiers
  - **Predefined: trim**
- **Custom modifiers**
  - set modifiers - **handle data manipulation before saving the document**
  - **get** - to modify existing data before outputting the documents to next layer
- **Virtual attributes**
  - dynamically calculated document properties
    - **get**
    - **set**
- Using indexes
  - **unique**
  - **index** for secondary index
- Custom **static** methods
- Custom **instance** methods

Lesson 6 Review

- **Model validation**
  - validate that information before passing it on to MongoDB:
    - Predefined
      - **required**
      - **enum**
      - **match**
    - Custom using **validate** property
- **pre middleware and post middleware**
  - Executed at instance level
    - Pre-save
    - Post-save

- **Using Mongoose DBRef**
  - **reference of a document to another document**

```
author: {
    type: Schema.ObjectId, ref: 'User'
}
```

- find() method that **populates the author property** will look like the following code snippet:

```
Post.find().populate('author').exec(
(err, posts) =>{
...
});
```

Managing User Authentication

Objectives:

- ❑ Explain Passport module
- ❑ Describe Passport strategies
- ❑ Integrate Passport into your users' MVC architecture
- ❑ Use Passport's local strategy to authenticate users
- ❑ Utilize Passport OAuth strategies
- ❑ Implement authentication through social OAuth providers.

Introducing Passport

- **Passport** is a Node.js module that uses the middleware design pattern **to authenticate requests**.
- To install the Passport base module in your application's modules folders, change your ***package.json*** file as follows:

```
{
"name": "MEAN",
"version": "0.0.6",
"dependencies": {
        "body-parser": "1.15.2",
        "compression": "1.6.0",
        "ejs": "2.5.2",
        "express": "4.14.0",
        "express-session": "1.14.1",
        "method-override": "2.3.6",
        "mongoose": "4.6.5",
        "morgan": "1.7.0",
        "passport": "0.3.2"
        }
}
```

- Run **npm install**

Passport Authentication Steps

- Install passport module
- Configure passport module in *config* folder
- Register passport module in express.js file
- Install authentication strategies modules
- Configure authentication strategies in separate files in *strategies* subfolder of *config* folder
- Authenticated user must be **serialized** to the session
- Serialized user should be **deserialized** when requests are made

Configuring Passport

- ❑ Create the Passport configuration file, in *config* folder:
  - ➢ create a new empty file named *passport.js*.
- ❑ Change the *server.js* file by requiring *passport.js:*

```
process.env.NODE_ENV = process.env.NODE_ENV || 'development';
const mongoose = require('./config/mongoose'),
express = require('./config/express'),
passport = require('./config/passport');
const db = mongoose();
const app = express();
const passport = passport();
app.listen(3000);
module.exports = app;
console.log('Server running at http://localhost:3000/');
```

Configuring Passport

- **Register the Passport middleware** in *config/express.js* file:

```
const config = require('./config'),
const express = require('express'),
const morgan = require('morgan'),
const compress = require('compression'),
const bodyParser = require('body-parser'),
const methodOverride = require('method-override'),
const session = require('express-session'),
const passport = require('passport');
module.exports = function() {
        const app = express();
        if (process.env.NODE_ENV === 'development') {
                app.use(morgan('dev'));
        } else if (process.env.NODE_ENV === 'production') {
                app.use(compress());
}
app.use(bodyParser.urlencoded({extended: true}));
```

Configuring Passport

```
app.use(bodyParser.json());
app.use(methodOverride());
app.use(session({
saveUninitialized: true,
resave: true,
secret: config.sessionSecret
}));
app.set('views', './app/views');
app.set('view engine', 'ejs');
app.use(passport.initialize()); //bootstrapping the Passport module
app.use(passport.session()); //keep track of your user's session
require('../app/routes/index.server.routes.js')(app);
require('../app/routes/users.server.routes.js')(app);
app.use(express.static('./public'));
return app;
};
```

Passport strategies

- Passport uses separate modules that implement **different authentication strategies**:
  - **Local strategy**:
    - to implement a **username/password authentication mechanism.**
    - **install** it like any other module and **configure** it **to use your User Mongoose model**.
  - **OAuth strategies -** authentication protocol that allows users to **register with your web application using an external provider**, without the need to input their username and password.
    - mainly used by social platforms, such as Facebook, Twitter, and Google, to **allow users to register with other websites using their social account**.

Installing local strategy module

- Change your *package.json* file, as follows:

```
{
"name": "MEAN",
"version": "0.0.6",
"dependencies": {
      "body-parser": "1.15.2",
      "compression": "1.6.0",
      "ejs": "2.5.2",
      "express": "4.14.0",
      "express-session": "1.14.1",
      "method-override": "2.3.6",
      "mongoose": "4.6.5",
      "morgan": "1.7.0",
      "passport": "0.3.2",
      "passport-local": "1.0.0"
      }
}
```

- Run **npm install**

Configuring local strategy

- To maintain a clear separation of logic, **each strategy should be configured in its own separated file**.
  - In your *config* folder, create a new folder named *strategies* - inside this new folder, create a file named ***local.js*** that contains the following code snippet:

```
const passport = require('passport'),
const LocalStrategy = require('passport-local').Strategy,
const User = require('mongoose').model('User');
//Register the strategy
module.exports = function() {
    passport.use(new LocalStrategy(function(username, password, done) {
    //find a user with that username and authenticate it
    User.findOne({username: username}, (err, user) => {
    if (err) {
    return done(err); //done is a Passport function
    }
```

Configuring local strategy

```
            if (!user) {
                return done(null, false, {message: 'Unknown user'});
            }
            if (!user.authenticate(password)) {
                return done(null, false, {message: 'Invalid password'});
            }
            return done(null, user); // user is authenticated
            });
        }));
    };
```

Configuring local strategy

- Require the **Passport module**, the local strategy module's **Strategy object**, and your **User Mongoose model**.
- **Register the strategy** using the **passport.use()** method that uses an instance of the LocalStrategy object.
- LocalStrategy constructor takes also a callback function as an argument
- The callback function accepts three arguments - **username**, **password**, and a ***done*** callback - which will be called when the authentication process is over.
  - Inside the callback function, you will **use the User Mongoose model to find a user with that username** and try to authenticate it.
  - In the event of an error, you will **pass the error object to the *done* callback**.
  - When the user is authenticated, you will **call the *done* callback with the *user* Mongoose object**.

Configuring local strategy

- **Configure the local authentication** - paste the following lines of code to *config/passport.js* file:

```
const passport = require('passport'),
const mongoose = require('mongoose');
//handling user serialization
module.exports = function() {
const User = mongoose.model('User');
//authenticated user must be serialized to the session
passport.serializeUser(function(user, done) {done(null, user.id);});
//deserialize when requests are made
passport.deserializeUser(function(id, done) {User.findOne({ _id: id },
'-password -salt', function(err, user) {done(err, user);});
});
require('./strategies/local.js')(); //include the local strategy config file
};
```

Configuring local strategy

- The *passport.serializeUser()* and *passport.deserializeUser()* methods are used to define how Passport will handle user **serialization**:
  - When a user is authenticated, Passport will save its **_id** property to the session.
  - Later on when the user object is needed, Passport will use the **_id** property to retrieve the user object from the database.
  - field options argument '-password -salt', is used to make sure Mongoose doesn't fetch the user's *password* and *salt* properties.
- The second thing the preceding code does is **including the local strategy configuration file**.
- This way, your *server.js* file will load the Passport configuration file, which in turn will load its strategies configuration file.
- Next, you'll need to modify your User model to support Passport's authentication.

Adapting the User model

- In order to use the User model in your MEAN application, you'll have to modify it to address a few authentication process requirements:
  - **modifying UserSchema**
  - **adding a pre middleware**
  - **add some new instance methods**
- Change *app/models/user.js* file as follows:

```
const mongoose = require('mongoose'),
const crypto = require('crypto'),
const Schema = mongoose.Schema;
const UserSchema = new Schema({
firstName: String,
lastName: String,
email: {
type: String,
match: [/.+\@.+\..+/, "Please fill a valid e-mail address"]
},
```

Adapting the User model

```
username: {
type: String,
unique: true,
required: 'Username is required',
trim: true
},
password: {
type: String,
validate: [
function(password) {
return password && password.length > 6;
}, 'Password should be longer'
]
},
salt: { //to hash the password
type: String
},
```

Adapting the User model

```
provider: { // strategy used to register the user
type: String,
required: 'Provider is required'
},
providerId: String, // user identifier for the authentication strategy
providerData: {},  // to store the user object retrieved from OAuth providers
created: {
type: Date,
default: Date.now
}
});
UserSchema.virtual('fullName').get(function() {
return this.firstName + ' ' + this.lastName;
}).set(function(fullName) {
const splitName = fullName.split(' ');
this.firstName = splitName[0] || '';
this.lastName = splitName[1] || '';
});
```

Adapting the User model

```
// pre-save middleware to handle the hashing of your users‘ passwords
UserSchema.pre('save', function(next) {
if (this.password) {
// creates an autogenerated pseudo-random hashing salt
this.salt = new Buffer(crypto.randomBytes(16).toString('base64'), 'base64');
this.password = this.hashPassword(this.password); //returns hashed password
}
next();
});
// replaces the current user password with a hashed password (more secure)
UserSchema.methods.hashPassword = function(password) {
return crypto.pbkdf2Sync(password, this.salt, 10000,64).toString('base64');
};
//authenticates the password
UserSchema.methods.authenticate = function(password) {
return this.password === this.hashPassword(password);
};
```

Adapting the User model

```
UserSchema.statics.findUniqueUsername = function(username, suffix,callback)
{ // find an available unique username for new users
        var _this = this;
        var possibleUsername = username + (suffix || '');
        _this.findOne({
                username: possibleUsername
        }, function(err, user) {
        if (!err) {
                if (!user) {
                        callback(possibleUsername);
                } else {
                return _this.findUniqueUsername(username, (suffix || 0) + 1, callback);
        }
        } else {
                callback(null);
        }
        });
};
```

Adapting the User model

```
UserSchema.set('toJSON', {
getters: true,
virtuals: true
});
mongoose.model('User', UserSchema);
```

- **Explanation of the changes:**
  - First, you **added four fields to your UserSchema object**:
    - a **salt** property, which you'll use **to hash your password**;
    - a **provider** property, which will **indicate the strategy** used to register the user;
    - a **providerId** property, which will indicate the user identifier for the authentication strategy;
    - a **providerData** property, which you'll later use to store the user object retrieved from OAuth providers.

Adapting the User model

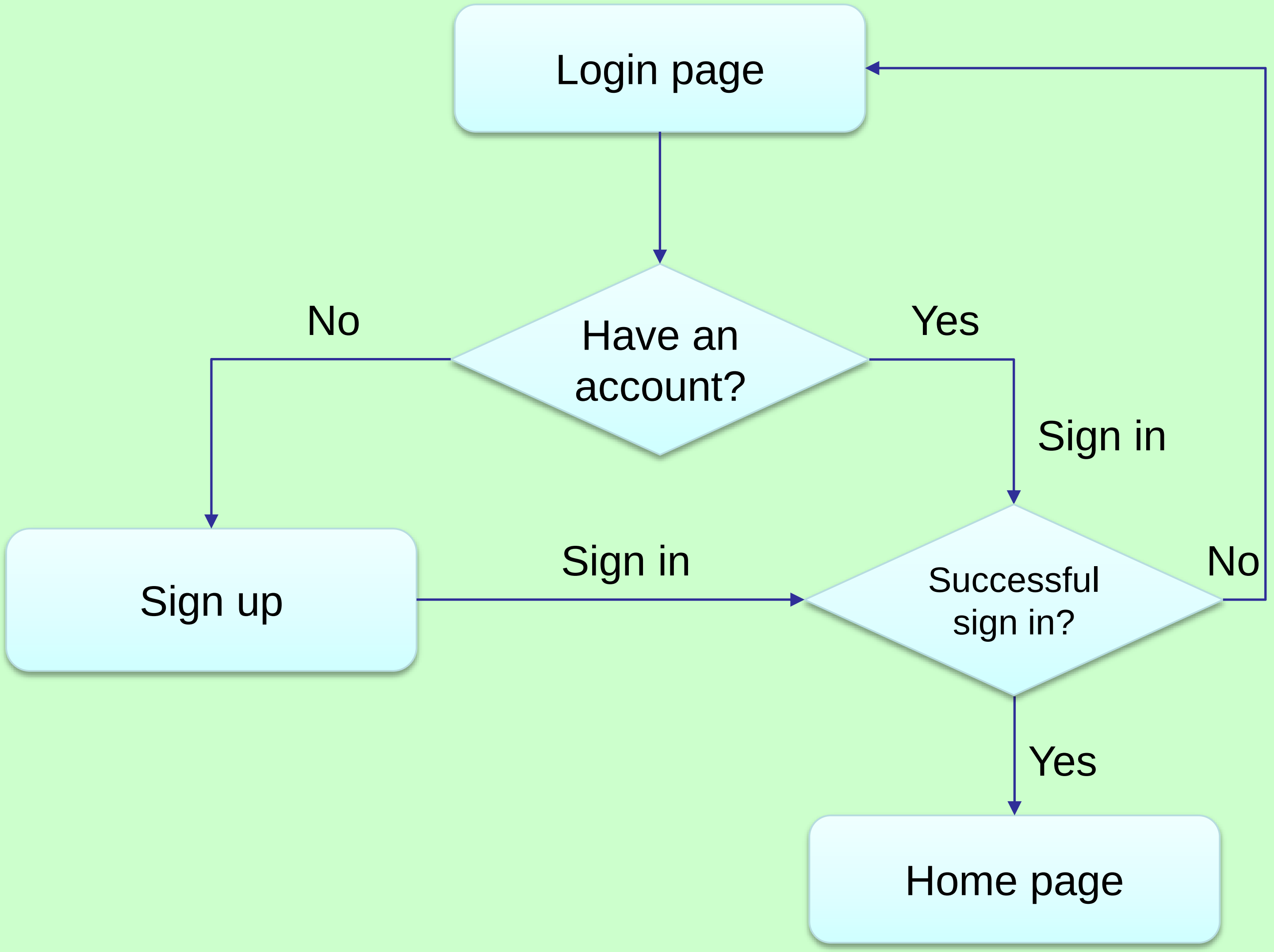
- Next, you **created a pre-save middleware to handle the hashing of your users' passwords**.
- Your **pre-save middleware** performs two important steps:
  - first, it creates an **autogenerated pseudo-random hashing salt**
  - then it **replaces the current user password with a hashed password** using the hashPassword() instance method.
- You also added **two instance methods**:
  - a **hashPassword() instance method**, which is used to hash a password string by utilizing Node.js' **crypto** module,
  - an **authenticate() instance method**, which accepts a string argument, hashes it, and compares it to the current user's hashed password.
  - Finally, you **added the findUniqueUsername() static method**, which is **used to find an available unique username for new users**.
    - You'll use this method later to deal with OAuth authentication.

Creating the authentication views

- *Create signup* and *sign-in* pages handle user authentication.
  - In your *app/views* folder, create a new file named ***signup.ejs* view** that simply contains an HTML form, an EJS tag, which **renders the title variable**, and an EJS loop, which **renders the messages list variable**.
  - In your *app/views* folder create another file named ***signin.ejs*** *that* is even simpler and also contains an HTML form, an EJS tag, which **renders the title variable**, and an EJS loop, which **renders the messages list variable**.

- Connect the model and views using Users controller.

Creating the authentication views



Modifying the user controller

- Go to your *app/controllers/users.server.controller.js* file, and change its content, as follows:

```
const User = require('mongoose').model('User'),
const passport = require('passport');
// returns a unified error message from a Mongoose error object
var getErrorMessage = function(err) {
var message = '';
if (err.code) {
switch (err.code) { //using error codes
case 11000:
case 11001:
message = 'Username already exists';
break;
default:
message = 'Something went wrong';
}
```

Modifying the user controller

```
} else { // a Mongoose validation error
for (var errName in err.errors) {
if (err.errors[errName].message) message = err.errors[errName].
message;
}
}
return message;
};
exports.renderSignin = function(req, res, next) {
if (!req.user) {
res.render('signin', {title: 'Sign-in Form',messages: req.flash('error') ||
req.flash('info')});
} else {
return res.redirect('/');
}
};
```

Modifying the user controller

```
exports.renderSignup = function(req, res, next) {
if (!req.user) {
res.render('signup', {
title: 'Sign-up Form',
messages: req.flash('error')
});
} else {
return res.redirect('/');
}
};
exports.signup = function(req, res, next) { //uses user model to create new
                                             //users
if (!req.user) {
var user = new User(req.body);
var message = null;
user.provider = 'local';
```

Modifying the user controller

```
user.save(function(err) {
if (err) {
var message = getErrorMessage(err);
req.flash('error', message);
return res.redirect('/signup');
}
req.login(user, function(err) { //req.login is Passport method
if (err) return next(err);
return res.redirect('/');
});
});
} else {
return res.redirect('/');
}
};
exports.signout = function(req, res) {
req.logout(); // invalidate the authenticated session using a Passport method
res.redirect('/');
};
```

Modifying the user controller

- The **getErrorMessage()** method is a private method that **returns a unified error message from a Mongoose error object**.
  - There are two possible errors here: a **MongoDB indexing error** handled using the error code and **a Mongoose validation error** handled using the *err.errors* object.
- The next two controller methods will be used to render the sign-in and signup pages:
  - The **signout() method** uses the **req.logout()** method, which is provided by the Passport module **to invalidate the authenticated session**.
  - The **signup() method** uses your **User model to create new users**.
    - it first **creates a user object from the HTTP request body**.
    - Then, **try saving it to MongoDB** - if an error occurs, the signup() method will use the **getErrorMessage()** method to provide the user with an appropriate error message.
  - If the user creation was successful, the **user session will be created** using the **req.login()** method.
    - The **req.login()** method is exposed by the Passport module and is used to establish a successful login session.
- After the login operation is completed, a user object will be signed to the **req.user** object.

Displaying flash error messages

- When redirecting to another page, you cannot pass variables to that page.
- The solution would be to use some sort of mechanism to pass temporary messages between requests.
- The **Connect-Flash module is a node module that allows you to store temporary messages in an area of the session object called *flash***.
    - Messages stored on the flash object **will be cleared once they are presented to the user**.
- This architecture makes the Connect-Flash module perfect to **transfer messages before redirecting the request to another page**.

Installing the Connect-Flash module

- To install the Connect-Flash module in your application's modules folders, you'll need to change your *package.json* file, as follows:

```
{
"name": "MEAN",
"version": "0.0.6",
"dependencies": {
        "body-parser": "1.15.2",
        "compression": "1.6.0",
        "connect-flash": "0.1.1",
        "ejs": "2.5.2",
        "express": "4.14.0",
        "express-session": "1.14.1",
        "method-override": "2.3.6",
        "mongoose": "4.6.5",
        "morgan": "1.7.0",
        "passport": "0.3.2",
        "passport-local": "1.0.0"
        }
}
```

Run **npm install**

Configuring Connect-Flash module

- To configure your Express application to use the new Connect-Flash module, you'll have to require the new module in your Express configuration file and use the **app.use()** method to register it with your Express application
- Change *config/express.js* file:

```
const config = require('./config'),
const express = require('express'),
const morgan = require('morgan'),
const compress = require('compression'),
const bodyParser = require('body-parser'),
const methodOverride = require('method-override'),
const session = require('express-session'),
const flash = require('connect-flash'),
const passport = require('passport');
```

Configuring Connect-Flash module

```
module.exports = function() {
const app = express();
if (process.env.NODE_ENV === 'development') {
app.use(morgan('dev'));
} else if (process.env.NODE_ENV === 'production') {
app.use(compress());
}
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
app.use(methodOverride());
app.use(session({
saveUninitialized: true,
resave: true,
secret: config.sessionSecret
}));
```

Configuring Connect-Flash module

```
app.set('views', './app/views');
app.set('view engine', 'ejs');
app.use(flash());
app.use(passport.initialize());
app.use(passport.session());
require('../app/routes/index.server.routes.js')(app);
require('../app/routes/users.server.routes.js')(app);
app.use(express.static('./public'));
return app;
};
```

Using Connect-Flash module

- Connect-Flash module **exposes the req.flash() method**, **to create and retrieve flash messages**.
- renderSignup() and renderSignin() methods in Users controller which are responsible for rendering the sign-in and signup pages, use **req.flash** method **to read the messages written to the flash**:

```
exports.renderSignin = function(req, res, next) {
if (!req.user) {
res.render('signin', {
title: 'Sign-in Form',
messages: req.flash('error') || req.flash('info')
});
} else {
return res.redirect('/');
}
};
```

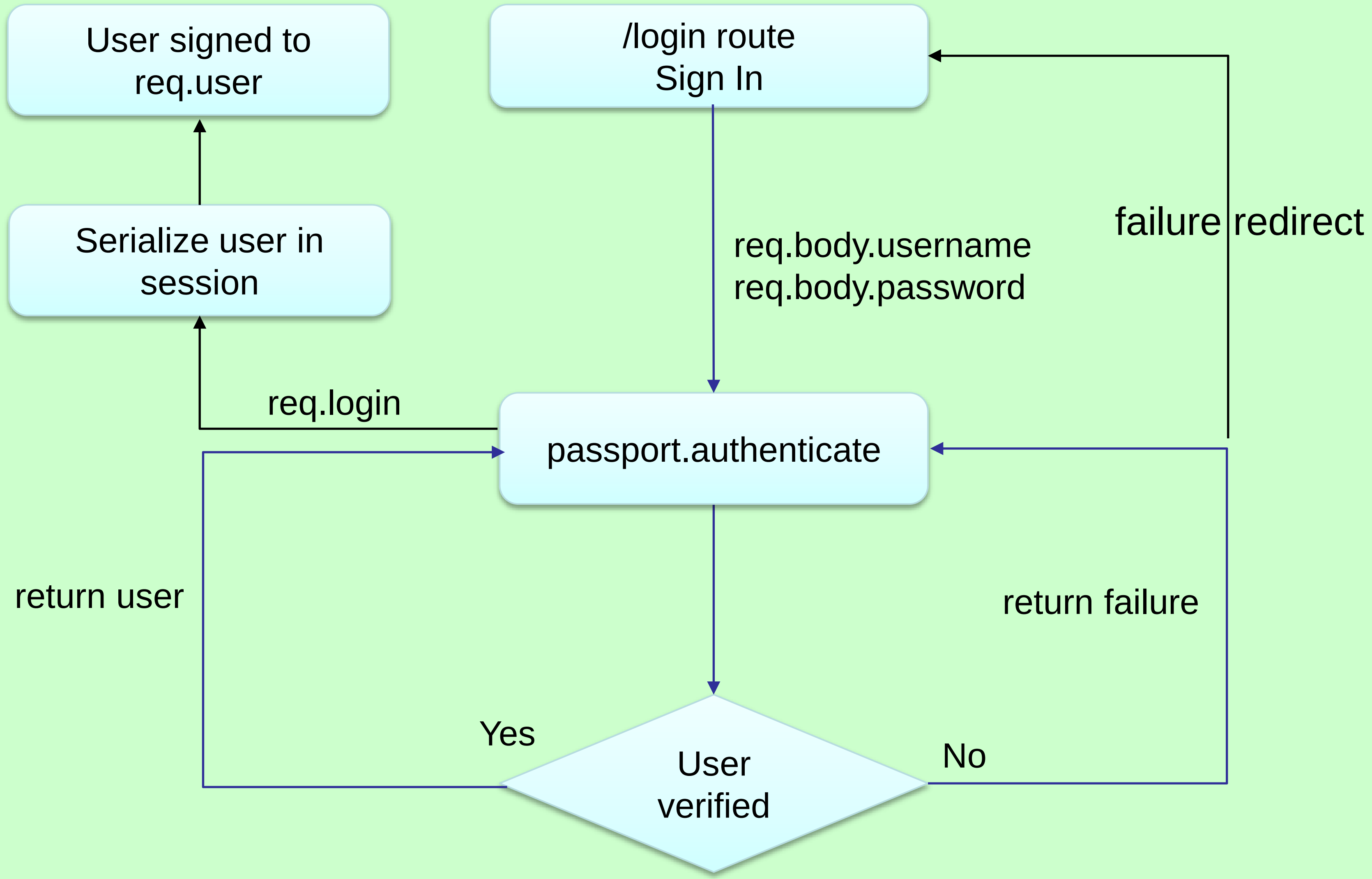
Using Connect-Flash module

```
exports.renderSignup = function(req, res, next) {
if (!req.user) {
res.render('signup', {
title: 'Sign-up Form',
messages: req.flash('error')
});
} else {
return res.redirect('/');
}
};
```

- The res.render() method is executed with the *title* and *messages* variables.
- The signup() method uses **req.flash() method** to write **error messages to the flash** :

req.flash('error', message);

Passport Authentication Flowchart



Wiring the user's routes

- Once you have your model, controller, and views configured, all that is left to do is **define the user's routes** - make the following changes in your *app/routes/users.server.routes.js* file:

```
var users = require('../../app/controllers/users.server.controller'),
passport = require('passport');
module.exports = function(app) {
app.route('/signup')
.get(users.renderSignup)
.post(users.signup);
app.route('/signin')
.get(users.renderSignin)
.post(passport.authenticate('local', {
successRedirect: '/',
failureRedirect: '/signin',
failureFlash: true
}));
app.get('/signout', users.signout);
};
```

Wiring the user's routes

- Routes definitions here are basically **directing to methods from your user controller**.
- The only different route definition is the one where you're **handling any POST request made to the** */signin* **path using the passport.authenticate() method of Passport module**.
  - That’s why we did not create a signin method in user controller
- When the passport.authenticate() method is executed, it will try to **authenticate the user request using the strategy defined by its first argument**.
  - In this case, it will try to **authenticate the request using the local strategy**.

Wiring the user's routes

- The second parameter this method accepts is an **options object, which contains three properties**:
  - **successRedirect**: tells Passport where to redirect the request once it successfully authenticated the user
  - **failureRedirect**: tells Passport where to redirect the request once it failed to authenticate the user
  - **failureFlash**: tells Passport whether or not to use flash messages

- This completes the basic authentication implementation.

Testing your app

- To test it out, make the following changes to the app/controllers/index.server.controller.js file:

```
exports.render = function(req, res) {
res.render('index', {
title: 'Hello World',
userFullName: req.user ? req.user.fullName : ''
});
};
```

- This will pass the authenticated user's full name to your home page template

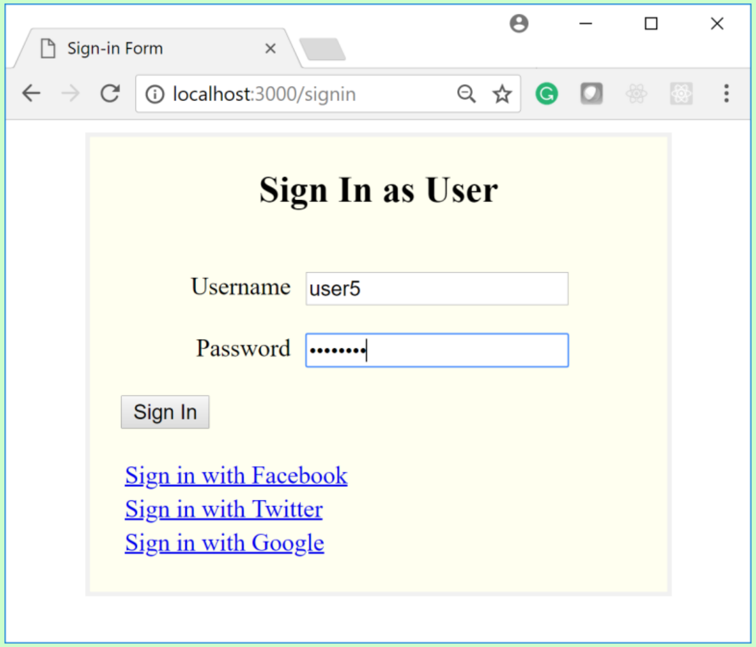
Testing your app

- Make the following changes in your app/views/index.ejs file:

```
<!DOCTYPE html>
<html>
<head>
<title><%= title %></title>
</head>
<body>
<% if ( userFullName ) { %>
<h2>Hello <%=userFullName%> </h2>
<a href="/signout">Sign out</a>
<% } else { %>
<a href="/signup">Signup</a>
<a href="/signin">Signin</a>
<% } %>
<br>
<img src="img/logo.png" alt="Logo">
</body>
</html>
```

Testing your app

- **Run node server**
- Test your application by visiting *http://localhost:3000/signin* and *http://localhost:3000/signup*.
- Try signing up, and then sign in and don't forget to go back to your home page to see how the user details are saved through the session.

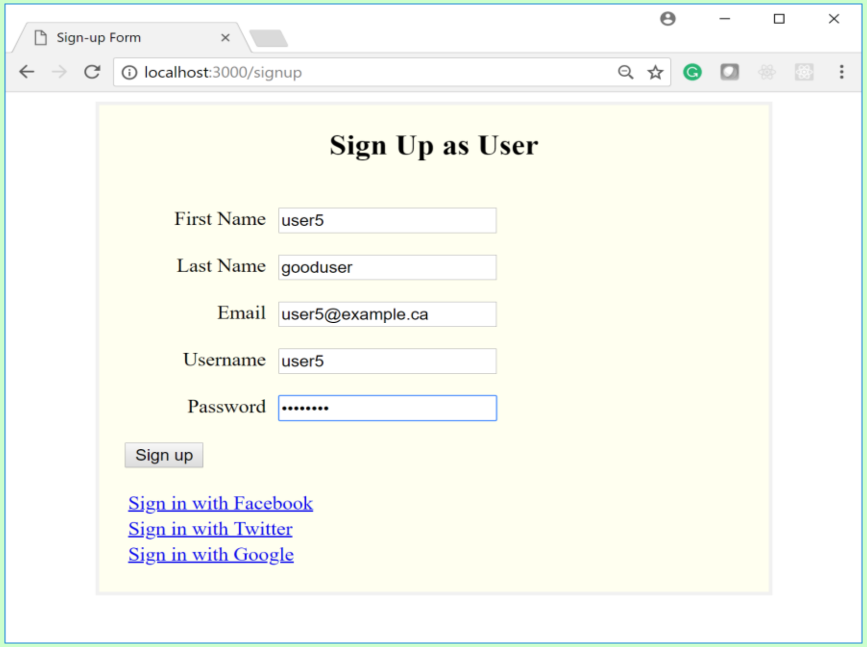


Sign In as User

Username	user5
Password	••••••••

Sign In

Sign in with Facebook
Sign in with Twitter
Sign in with Google



Sign Up as User

First Name	user5
Last Name	gooduser
Email	user5@example.ca
Username	user5
Password	••••••••

Sign up

Sign in with Facebook
Sign in with Twitter
Sign in with Google

Passport OAuth strategies

- OAuth is an authentication protocol that **allows users to register with your web application using an external provider**, without the need to input their username and password.
- OAuth is mainly used by social platforms, such as **Facebook**, **Twitter**, and **Google**, to allow users to register with other websites using their social account.
- Passport support the basic **OAuth strategy**, which enables you to implement any OAuth-based authentication.
- It also supports a user authentication through major OAuth providers using wrapper strategies that help you avoid the need to implement a complex mechanism by yourself.

Handling OAuth user creation

- The OAuth user creation should be a bit different than the local signup() method.
- Since users are signing up using their profile from other providers, the profile details are already present, which means you will need to validate them differently.
- To do so, go back to your *app/controllers/users.server.controller.js* file, and add the following module method:

```
exports.saveOAuthUserProfile = function(req, profile, done) {
        User.findOne({
        provider: profile.provider,
        providerId: profile.providerId
        }, function(err, user) {
        if (err) {
        return done(err);
        } else {
```

Handling OAuth user creation

```
if (!user) {
        var possibleUsername = profile.username || ((profile.email) ? profile.email.split('@')[0] : '');
        User.findUniqueUsername(possibleUsername, null, function(availableUsername) {
                profile.username = availableUsername;
                user = new User(profile);
                user.save(function(err) {
                        if (err) {
                        var message = _this.getErrorMessage(err);
                        req.flash('error', message);
                        return res.redirect('/signup');
                        }
                        return done(err, user); //saves new user instance
                });
        });
        } else {
        return done(err, user); //finds the user
        }
}
});
};
```

Handling OAuth user creation

- This method accepts a user profile, and then **looks for an existing user with these providerId and provider properties**.
  - If it finds the user, it calls the done() callback method with the user's MongoDB document.
  - If it cannot find an existing user, it will find a unique username using the User model's findUniqueUsername() static method and **save a new user instance**.
  - If an error occurs, the saveOAuthUserProfile() method will use the req.flash() and getErrorMessage() methods to report the error; otherwise, it will pass the user object to the done() callback method.

Using Passport's Facebook strategy

- ❑ Facebook is probably the world's largest OAuth provider.
- ❑ Many modern web applications offer their users the ability to register with the web application using their Facebook profile.
- ❑ Passport supports **Facebook OAuth authentication** using the **passport-facebook** module.
- ❑ Let's see how you can implement a Facebook-based authentication in a few simple steps.
- ❑ To install Passport's Facebook module in your application's modules folders, you'll need to change your *package.json* file as follows:

Using Passport's Facebook strategy

```
{
"name": "MEAN",
"version": "0.0.6",
"dependencies": {
"express": "~4.8.8",
"morgan": "~1.3.0",
"compression": "~1.0.11",
"body-parser": "~1.8.0",
"method-override": "~2.2.0",
"express-session": "~1.7.6",
"ejs": "~1.0.0",
"connect-flash": "~0.1.1",
"mongoose": "~4.3.7",
"passport": "~0.2.1",
"passport-local": "~1.0.0",
"passport-facebook": "~1.0.3"
}
}
```

```
npm install
```

Configuring Passport's Facebook strategy

- Before you begin configuring your Facebook strategy, you will have to go to Facebook's developer home page at *https://developers.facebook.com/*, create a **new Facebook application**, and **set the local host as the application domain**.
- After configuring your Facebook application, you will get a *Facebook application ID* and *secret*.
- You'll **need those to authenticate your users via Facebook**, so let's save them in our environment configuration file.
- Go to the *config/env/development.js* file and change it as follows:

```
module.exports = {
db: 'mongodb://localhost/mean-book',
sessionSecret: 'developmentSessionSecret',
facebook: {
clientID: 'Application Id',
clientSecret: 'Application Secret',
callbackURL: 'http://localhost:3000/oauth/facebook/callback'
}
};
```

Configuring Passport's Facebook strategy

- Replace *Application Id* and Application *Secret* with your Facebook application's ID and secret.
- The callbackURL property will be passed to the Facebook OAuth service, which will redirect to that URL after the authentication process is over.
- In *config/strategies* folder, create a new file named *facebook.js* that contains the following code snippet:

```
var passport = require('passport'),
url = require('url'),
FacebookStrategy = require('passport-facebook').Strategy,
config = require('../config'),
users = require('../../app/controllers/users.server.controller');
```

Configuring Passport's Facebook strategy

```
module.exports = function() {
passport.use(new FacebookStrategy({
clientID: config.facebook.clientID,
clientSecret: config.facebook.clientSecret,
callbackURL: config.facebook.callbackURL,
passReqToCallback: true
},
function(req, accessToken, refreshToken, profile, done) {
var providerData = profile._json;
providerData.accessToken = accessToken;
providerData.refreshToken = refreshToken;
```

Configuring Passport's Facebook strategy

- The preceding code begins by requiring the Passport module, the Facebook Strategy object, your environmental configuration file, your User Mongoose model, and the Users controller.
- Then, you register the strategy using the passport.use() method and creating an instance of a FacebookStrategy object. The FacebookStrategy constructor takes two arguments:
- the *Facebook application information* and a *callback function* that it will call later when trying to authenticate a user.
- The callback function takes five arguments:
  - the HTTP **request object**,
  - an **accessToken object** to validate future requests
  - a **refreshToken object** to grab new access tokens
  - a **profile object** containing the user profile
  - a ***done*** callback to be called when the authentication process is over

Configuring Passport's Facebook strategy

- Inside the callback function, you will create a new user object using the Facebook profile information and the controller's saveOAuthUserProfile() method, which you previously created, to authenticate the current user.
- Now that you have your Facebook strategy configured, you can go back to it and load the strategy file.
- Change the *config/passport.js* file as follows:

```
var passport = require('passport'),
mongoose = require('mongoose');
module.exports = function() {
var User = mongoose.model('User');
passport.serializeUser(function(user, done) {
done(null, user.id);
});
```

Configuring Passport's Facebook strategy

```
passport.deserializeUser(function(id, done) {
User.findOne({
_id: id
}, '-password -salt', function(err, user) {
done(err, user);
});
});
require('./strategies/local.js')();
require('./strategies/facebook.js')();
};
```

- This will load your Facebook strategy configuration file

Wiring Passport's Facebook strategy routes

- Passport OAuth strategies support the ability to authenticate users directly using the passport.authenticate() method.
- To do so, go to *app/routes/users.server.routes.js*, and append the following lines of code after the local strategy routes definition:

```
app.get('/oauth/facebook', passport.authenticate('facebook', {
failureRedirect: '/signin'
}));
app.get('/oauth/facebook/callback',
passport.authenticate('facebook',
{
failureRedirect: '/signin',
successRedirect: '/'
}));
```

Wiring Passport's Facebook strategy routes

- The first route will use the passport.authenticate() method to start the user authentication process, while the second route will use the passport.authenticate() method to finish the authentication process once the user has linked their Facebook profile.
- All you have to do now is go to your *app/views/signup.ejs and app/views/signin.ejs* files, and add the following line of code right before the closing BODY tag:

```
<a href="/oauth/facebook">Sign in with Facebook</a>
```

- This will allow your users to click on the link and register with your application via their Facebook profile.

Configuring Passport's Facebook strategy

```
var providerUserProfile = {
firstName: profile.name.givenName,
lastName: profile.name.familyName,
fullName: profile.displayName,
email: profile.emails[0].value,
username: profile.username,
provider: 'facebook',
providerId: profile.id,
providerData: providerData
};
users.saveOAuthUserProfile(req, providerUserProfile, done);
}));
};
```

References

- Textbook
- http://passportjs.org/docs
- https://github.com/jaredhanson/passport
- http://code.tutsplus.com/tutorials/authenticating-nodejs-applications-with-passport--cms-21619
- http://code.tutsplus.com/articles/social-authentication-for-nodejs-apps-with-passport--cms-21618
- https://scotch.io/courses/easy-node-authentication
- http://expressjs.com/
- http://www.tutorialspoint.com/mongodb/mongodb_quick_guide.htm
- http://mongoosejs.com/docs/
- http://www.tutorialspoint.com//nodejs/nodejs_express_framework.htm
- http://toon.io/understanding-passportjs-authentication-flow/