CWEB280 -wk5 – LO2 -Secure apps using objects

# Nav Bar Setup

We are moving onto the second learning outcome of the course – time to add a navbar to make navigation easier through the project and learning out comes

**\view\layout.hbs – the code in violet should be changed to your preferred bootswatch style**

<!DOCTYPE html>  
<html lang="en">  
 <head>  
 <!-- Required meta tags -->  
 <meta charset="utf-8">  
 <meta name="viewport" content="width=device-width, initial-scale=1">  
 <title>{{title}}</title>  
 <link rel='stylesheet' href='/bw/quartz/bootstrap.css' />  
 </head>  
 <body>  
 <nav class="navbar sticky-top navbar-expand-md navbar-dark bg-dark bg-opacity-50">  
 <div class="container-fluid">  
 <a class="navbar-brand" href="/">Server Only</a>  
 <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent" aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">  
 <span class="navbar-toggler-icon"></span>  
 </button>  
 <div class="collapse navbar-collapse" id="navbarSupportedContent">  
 <ul class="navbar-nav me-auto mb-2 mb-lg-0">  
<li class="nav-item dropdown">  
 <a class="nav-link dropdown-toggle" href="/examples/" id="navbarDropdown" role="button" data-bs-toggle="dropdown" aria-expanded="false">  
 LO1 Server-side Basics  
 </a>  
 <ul class="dropdown-menu bg-dark bg-opacity-50" aria-labelledby="navbarDropdown">  
 <li><a class="dropdown-item" href="/examples/simple-code/">Simple Code</a></li>  
 <li><a class="dropdown-item" href="/examples/form/">Simple Form</a></li>  
 <li><a class="dropdown-item" href="/examples/upload/">Upload Files</a></li>  
 </ul>  
</li>  
<li class="nav-item dropdown">  
 <a class="nav-link dropdown-toggle" href="/secure/" id="navbarDropdown" role="button" data-bs-toggle="dropdown" aria-expanded="false">  
 LO2 Secure Objects  
 </a>  
 <ul class="dropdown-menu bg-dark bg-opacity-50" aria-labelledby="navbarDropdown">  
 <li><a class="dropdown-item" href="/secure/{{#if token}}?access\_token={{token}}{{/if}}">Login</a></li>  
 <li><hr class="dropdown-divider"></li>  
 <li><a class="dropdown-item" href="/secure/dashboard/{{#if token}}?access\_token={{token}}{{/if}}">Admin Dashboard</a></li>  
 <li><a class="dropdown-item" href="/secure/profile/{{#if token}}?access\_token={{token}}{{/if}}">Employee Profile</a></li>  
 <li><a class="dropdown-item" href="/secure/booking/{{#if token}}?access\_token={{token}}{{/if}}">Client Booking</a></li>  
 <li><hr class="dropdown-divider"></li>  
 <li><a class="dropdown-item" href="/secure/passport/{{#if token}}?access\_token={{token}}{{/if}}">Passport</a></li>  
 </ul>  
</li>

<li class="nav-item dropdown">  
 <a class="nav-link dropdown-toggle" href="/state/" id="navbarDropdown" role="button" data-bs-toggle="dropdown" aria-expanded="false">  
 LO3 Cookies and Sessions  
 </a>  
 <ul class="dropdown-menu bg-dark bg-opacity-50" aria-labelledby="navbarDropdown">  
 <li><a class="dropdown-item" href="/state/cookie/">Set and View Cookies</a></li>  
 <li><a class="dropdown-item" href="/state/session/">Set and View Session Data</a></li>  
 </ul>  
</li>  
 </ul>  
 </div>  
 </div>  
 </nav>  
 <div class="container">  
 {{{body}}}  
 </div>  
  
 <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.1/dist/js/bootstrap.bundle.min.js"  
 integrity="sha384-/bQdsTh/da6pkI1MST/rWKFNjaCP5gBSY4sEBT38Q/9RBh9AH40zEOg7Hlq2THRZ"  
 crossorigin="anonymous"></script>  
 </body>  
</html>

# JSON Web Tokens

JSON Web Token (JWT) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object

* JWTs should be encrypted to provide secrecy between server and clients
  + Encrypted tokens hide the payload within the token from other parties that do not know the secret
* JWTs can also be “signed” to verify the payload contained within the toke.
  + tokens are signed using public/private key pairs
  + Only the party holding the private key is the one that signed it.
  + The public key is shared with pother parties to decrypt the token
* JWTs are traditionally used to secure a web API (Application Programming Interface)
  + We learn more about web APIs later in the course
  + Just for an example, we will use a JWT to secure some web pages (not the best way to secure webpages)
* JWTs have expiry dates
  + Unlike cookies, JWT cannot be expired after they are created
  + Once the expiry is set the JWT will remain valid until the expiry date is reached
  + RECOOMENED: set expiry to 30 minutes to an hour

Learn More about JWTs:

* <https://jwt.io/>
* <https://curity.io/resources/learn/jwt-best-practices/>

## The jsonwebtoken package

The jsonwebtoken package will make it easier to **sign**(encode/encrypt) a JWT and will also **verify**(decode/decrypt) the JWT

The module requires 3 parameters

1. Payload object – a JS object that contains any number of key value pairs
   1. Should not be used for constantly changing values
2. Secret or Key - a large random string or a private/public key file buffer
3. Options object – parameters describing the configuration of the JWT

Example:

// require a module to generate and decode JWT  
const ***jwt*** = require('jsonwebtoken');

const secret ='6ImRhc2hib2FyZCIsImlhdCI6MTYzMjMwNTA1OX0.iaVBoA0v3cVdK5PzLQs8A4XAdvCNb';

//to sign/encode

const token = ***jwt***.sign(payload, secret, {algorithm: 'HS256', expiresIn: '10s'} );

//to verify/decode

const decoded = ***jwt***.verify(token, secret, {algorithm: 'HS256'} );

Learn more about jsonwebtoken: <https://www.npmjs.com/package/jsonwebtoken#usage>

# Use JWT in Express Project

In the terminal run

npm i jsonwebtoken

## Create a new route file

**\routes\secure.js**

*/\*\*  
 \* secure.js  
 \* router for the secure/ path examples  
 \*/*/\* eslint-disable max-len \*/  
  
const ***express*** = require('express');  
const ***router*** = ***express***.Router();  
  
// require fs package to do some file IO (move uploaded files)  
const fs = require('fs');  
  
// require a module to generate and decode JWT  
const ***jwt*** = require('jsonwebtoken');  
  
// BEST PRACTICE: is to store the secret in an environment variable or file  
// the line below is bad practice but good enough for an example  
const secret ='6ImRhc2hib2FyZCIsImlhdCI6MTYzMjMwNTA1OX0.iaVBoA0v3cVdK5PzLQs8A4XAdvCNb';  
  
  
*/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \* LOGIN GET  
 \*/****router***.get('/', function(req, res, next) {  
 res.render('secure-login', {  
 title: 'GET LOGIN FORM',  
 submittedEmail: 'admin@t.ca', // hard code an email to save on typing it all the time  
 submittedPassword: '123456Pw', // hard code a password to save on typing it all the time  
 });  
});  
  
  
*/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \* LOGIN POST  
 \*/****router***.post('/', function(req, res, next) {  
 res.render('secure-login', {  
 title: 'POST LOGIN FORM',  
 isSubmitted: true,  
 err: {email: 'Email is not recognized', pwd: 'Password is not recognized'},  
 submittedEmail: req.body.email,  
 submittedPassword: req.body.pwd,  
 });  
});  
  
*/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \* GET Dashboard Profile Booking  
 \*/****router***.get([  
 '/dashboard', '/profile', '/booking',  
], (req, res, next)=>{  
// get the current scope from the request path (remove the slashes from /dashboard/)

const scope = req.path.replace(/^**\/**+|**\/**+$/g, '');  
 res.render('secure-generic', {  
 title: scope.toUpperCase(),  
 action: req.baseUrl + req.path,  
 });  
} );  
  
  
*/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \* POST Dashboard Profile Booking  
 \*/****router***.post([  
 '/dashboard', '/profile', '/booking',  
], (req, res, next)=>{  
  
 const scope = req.path.replace(/^**\/**+|**\/**+$/g, '');  
 res.render('secure-generic', {  
 title: 'POST' + scope.toUpperCase(),  
 });  
} );  
  
*/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \* GET PASSPORT - http://localhost:3000/secure/passport  
 \*/****router***.get('/passport', (req, res, next)=>{  
 res.render('secure-generic', {  
 title: 'GET - PASSPORT',  
 action: req.baseUrl+req.path, // result '/secure/passport/'  
 });  
});

*/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \* POST PASSPORT - http://localhost:3000/secure/passport  
 \*/****router***.post('/passport', (req, res, next)=>{  
 res.render('secure-generic', {  
 title: 'POST - PASSPORT',  
   
 });  
});

module.exports = ***router***;

Add the route and path in app.js

**\app.js - add the code in violet**

const logger = require('morgan');

const ***indexRouter*** = require('./routes/index');  
const ***usersRouter*** = require('./routes/users');  
const ***examplesRouter*** = require('./routes/examples');  
const ***secureRouter*** = require('./routes/secure');  
const ***app*** = ***express***();  
  
// view engine setup  
***app***.set('views', path.join(\_\_dirname, 'views'));  
***app***.set('view engine', 'hbs');  
  
***app***.use(logger('dev'));  
***app***.use(***express***.***json***());  
***app***.use(***express***.***urlencoded***({extended: false}));  
***app***.use(cookieParser());  
***app***.use(***express***.***static***(path.join(\_\_dirname, 'public')));  
  
***app***.use('/', ***indexRouter***);  
***app***.use('/users', ***usersRouter***);  
// add path for our new examples router  
***app***.use('/examples', ***examplesRouter***);  
***app***.use('/secure', ***secureRouter***);

## Create New Handlebar templates

**\view\secure-login.hbs**

<!--  
secure-login.hbs  
template for GET and POST form examples  
-->  
<form action="/secure/" method="post">  
<h1>{{title}}</h1>  
 <h2 class="text-danger">{{err.auth}}</h2>  
 <div class="mb-2">  
 <label class="form-label" for="email">Email address:</label>  
 <input type="email" class="form-control {{#if err.email}}is-invalid{{/if}}"  
 placeholder="Enter email" id="email" name="email" value="{{submittedEmail}}">  
 <div class="invalid-feedback"> {{err.email}} </div>  
 </div>  
 <div class="mb-2">  
 <label class="form-label" for="pwd">Password:</label>  
 <input type="text" class="form-control {{#if err.pwd}}is-invalid{{/if}}"  
 placeholder="Enter password" id="pwd" name="pwd" value="{{submittedPassword}}">  
 <div class="invalid-feedback"> {{err.pwd}} </div>  
 </div>  
 <button type="submit" class="btn btn-primary">Submit</button>  
</form>  
{{#if isSubmitted}}  
 <div class="card p-2 bg-gradient">  
 <h2>Submitted Values</h2>  
 <p>Submitted Email: {{submittedEmail}}</p>  
 <p>Submitted Password: {{submittedPassword}}</p>  
 </div>  
{{/if}}

**\view\secure-generic.hbs - will be used by 3 different paths**

<div class="p-5 mb-4 bg-gradient rounded-3">  
 <div class="container-fluid py-5">  
 <h1 class="display-5 fw-bold">{{title}}</h1>  
<!-- loop through the token payload object to display the properties and values-->  
 {{#each payload}}  
 <p class="col-md-8 fs-4">{{@key}}: {{this}}</p>  
 {{/each}}  
<!-- Only show button if action is specified-->  
 {{#if action}}  
 <form action="{{action}}" method="post">  
<!-- pass the JWT as a value in a hidden input-->  
 <input type="hidden" name="access\_token" value="{{token}}"/>  
 <button type="submit" class="btn btn-primary">Submit</button>  
 </form>  
 {{/if}}  
 </div>  
</div>

Restart the webserver for the changes to take effect - Navigate to <http://localhost:3000/examples/secure/>

Ensure you can view all the pages in LO1 and LO2

## Determine Access

This example is going to ask for an email and password to authenticate a user.

* Normally a web app would check a database to ensure the email and password correspond to an existing user.
* For this example, we are going to use a simple hard coded function which will allow on 3 emails and 1 passwords so as not complicate the example.
  + The emails are [admin@t.ca](mailto:admin@t.ca), [work@t.ca](mailto:work@t.ca), and [client@t.ca](mailto:client@t.ca) .
  + The password is just going to be ***123456Pw***
* Each user will be store 2 properties in the token payload: **role** and **scope** ( the page they get redirected to when they log in)
  + Admin user – role: ‘admin’ and scope: ’dashboard’
  + Work user – role: ‘employee and scope: ‘profile’
  + Client user – role: ‘client and scope: ‘booking’
* The function will take in the request object as a parameter and return a payload object with role and scope

**\routes\secure.js - add the following function**

*/\*\*  
 \* Determine role and scope from email and password  
 \** ***@param*** *{Request} req  
 \** ***@return*** *{{role: null, scope: null}}  
 \*/*const determineAccess = (req)=> {  
 const payload = {role: null, scope: null};  
 // authenicate password  
 if (req.body.pwd === '123456Pw') {  
 // authenticate email  
 switch (req.body.email.toLowerCase()) {  
 case 'admin@t.ca':  
 payload.role = 'admin';  
 payload.scope = 'dashboard'; // webpage they have access to  
 break;  
 case 'work@t.ca':  
 payload.role = 'employee';  
 payload.scope = 'profile';  
 break;  
 case 'client@t.ca':  
 payload.role = 'client';  
 payload.scope = 'booking';  
 break;  
 default:  
 // do nothing  
 }  
 }  
 return payload;  
};

## Encode a JWT

This function will encode the token given the payload object as a parameter

We are first going to use a secret and the **HS256** algorithm to encode the token, but later we will key files to encode the token more securely

**\routes\secure.js - add the following function**

*/\*\*  
 \* generate at JWT from the payload and options  
 \** ***@param*** *{{role: '', scope: ''}} payload  
 \** ***@return*** *{String} token  
 \*/*const encodeJWT = (payload)=>{  
 const token = ***jwt***.sign(payload, secret, {algorithm: 'HS256'} );  
 return token;  
};

## Passing the JWT in the URL (/?access\_token=…)

Now we call the above functions when the user submits /posts the login form

Notice that we use the scope to redirect the user to the appropriate page along with the access\_token in the URL

res.redirect('/secure/'+ payload.***scope*** + '/?access\_token=' + encodeJWT(payload));

If the user is not redirected, then show the error messages for the email and password

err: {email: 'Email is not recognized', pwd: 'Password is not recognized'},

**\routes\secure.js - add code in violet**

*/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \* LOGIN POST  
 \*/****router***.post('/', function(req, res, next) {  
 const payload = determineAccess(req);  
 if (payload.***role***) {  
 // redirect to the appropriate webpage for the user  
 res.redirect('/secure/'+ payload.***scope*** + '/?access\_token=' + encodeJWT(payload));  
 } else {  
 // show error messages - let the user try again  
 res.render('secure-login', {  
 title: 'POST LOGIN FORM',  
 isSubmitted: true,  
 err: {email: 'Email is not recognized', pwd: 'Password is not recognized'},  
 submittedEmail: req.body.email,  
 submittedPassword: req.body.pwd,  
 });  
 }  
});

Restart the webserver for the changes to take effect - Navigate to <http://localhost:3000/examples/secure/>

**Exercise:  
What happens when you login with the 3 different emails and the password?  
What happens when you navigate to the other pages in the LO2? Does the token get passed around?  
What happens when you navigate to the other pages in the LO1? Does the token get passed around?**

## Check the JWT

You will notice the access\_token is passed around in the url, but as soon as you got to LO1 page the token is lost.

Also, you will notice that you can navigate to any of the pages in LO2. That is because we have not verified the token and restricted access to the pages

Now we need add some code to allow admin users to visit any page, but the work and client users can only visit their scope page.

*/\*\*  
 \* Check the payload role and scope and compare to the current scope path  
 \** ***@param*** *{String} token  
 \** ***@param*** *{String} scope  
 \** ***@return*** *{{scope}}  
 \*/*const checkJWT = (token, scope)=>{  
 let decoded;  
 try {  
 // verify /decode the token using secret key and H256 scheme  
 decoded = ***jwt***.verify(token, secret, {algorithm: 'HS256'} );  
  
 // check that the scope in the token matches the scope passed in as a param  
 if (!decoded.role || decoded.role != 'admin') { // this allows admin user to go to any page  
 if (!decoded.scope || decoded.scope != scope) throw new ***Error***('scope is not permitted');  
 }  
 } catch (err) {  
 ***console***.log(`JWT Error:\n ${err}`);  
 // redirect back to login page and specify the error message in the query string  
 decoded = {redirectURL: '/secure/?err=' + err.message};  
 }  
 return decoded;  
};

The checkJWT function will return an object with a **role** and **scope**, but if there is an error, the returned object will have a **redirectURL** instead

Now we need to call the checkJWT function in the GET handler for dashboard, profile, booking.

If there is a redirect url then redirect ,

otherwise display the page along with the access\_token and the decoded payload from the JWT

**\routes\secure.js - add code in violet**

*/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \* GET Dashboard Profile Booking  
 \*/****router***.get([  
 '/dashboard', '/profile', '/booking',  
], (req, res, next)=>{  
// get the current scope from the request path (remove the slashes from /dashboard/)

const scope = req.path.replace(/^**\/**+|**\/**+$/g, '');  
 // check the user has access to the current scope  
 const decoded = checkJWT(req.query['access\_token'], scope);  
  
 if (decoded.redirectURL) {  
 // show the error on the login page  
 res.redirect(decoded.redirectURL);  
 } else {  
 // display the webpage with title and token payload  
 res.render('secure-generic', {  
 title: scope.toUpperCase(),  
 token: req.query['access\_token'],  
 payload: decoded,  
 action: req.baseUrl + req.path,  
 });  
 }  
} );

**Exercise:  
What happens when you login as client? Can you navigate to the profile or dashboard pages?  
What happens when you login as work? Can you navigate to the booking or dashboard pages?  
What happens when you login as admin? Can you navigate to the other pages?**

## Passing the JWT in a Hidden Form Input

Passing the JWT in the URL has disadvantages and normally the JWT would be passed in a cookie or as the Authorization header.  
A slightly better way to pass around the JWT from page to page is in a hidden form input

Review the code already in secure-generic.hbs

{{#if action}}  
<form action="{{action}}" method="post">  
 <input type="hidden" name="access\_token" value="{{token}}">  
 <button type="submit" class="btn btn-primary">Submit</button>  
</form>  
{{/if}}

Not let add an option in the post handlers accept the hidden input when the form is posted

*/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \* POST Dashboard Profile Booking  
 \*/****router***.post([  
 '/dashboard', '/profile', '/booking',  
], (req, res, next)=>{  
 // console.log(req.path);  
 const scope = req.path.replace(/^**\/**+|**\/**+$/g, '');  
 // check the user has access to the current scope  
 const decoded = checkJWT(req.body['access\_token'], scope);  
  
 if (decoded.redirectURL) {  
 // show the error on the login page  
 res.redirect(decoded.redirectURL);  
 } else {  
 // display the webpage with title and token payload  
 res.render('secure-generic', {  
 title: 'POST' + scope.toUpperCase(),  
 token: req.body['access\_token'],  
 payload: decoded,  
 });  
 }  
} );

**Exercise:  
Notice that then you submit the dashboard, profile or booking pages that the access\_token in the URL disappears, but you are still authorized**

# **Using files to Encode/Decode the JWT**

As part of the best practices for using JWT, developers should use private and public keys

First create, the following files in the lo1serveronly folder

**CREATE - \es256private.key**

-----BEGIN PRIVATE KEY-----  
MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgevZzL1gdAFr88hb2  
OF/2NxApJCzGCEDdfSp6VQO30hyhRANCAAQRWz+jn65BtOMvdyHKcvjBeBSDZH2r  
1RTwjmYSi9R/zpBnuQ4EiMnCqfMPWiZqB4QdbAd0E7oH50VpuZ1P087G  
-----END PRIVATE KEY-----

**CREATE - \es256public.pem**

-----BEGIN PUBLIC KEY-----  
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEEVs/o5+uQbTjL3chynL4wXgUg2R9  
q9UU8I5mEovUf86QZ7kOBIjJwqnzD1omageEHWwHdBO6B+dFabmdT9POxg==  
-----END PUBLIC KEY-----

Now change the encodeJWT and checkJWT function to use the files (along with the **ES256** algorithm) instead of the secret

**\routes\secure.js - add code in violet**

*/\*\*  
 \* generate at JWT from the payload and options  
 \** ***@param*** *{{role: '', scope: ''}} payload  
 \** ***@return*** *{String} token  
 \*/*const encodeJWT = (payload)=>{  
 // get a private key stored in a file  
 const privateKey = ***fs***.readFileSync('es256private.key'); // gets file buffer with binary file content  
  
 // use private key to encrypt the token with ES256 encoding scheme  
 const token = ***jwt***.sign(payload, privateKey, {algorithm: 'ES256', expiresIn: '30s',  
 } );  
 return token;  
};

**\routes\secure.js - add code in violet**

*/\*\*  
 \* Check the payload role and scope and compare to the current scope path  
 \** ***@param*** *{String} token  
 \** ***@param*** *{String} scope  
 \** ***@return*** *{{scope}}  
 \*/*const checkJWT = (token, scope)=>{  
 let decoded;  
 try {  
  
 // get a public key stored in a file  
 const cert = ***fs***.readFileSync('es256public.pem'); // get file buffer of binary file data  
 // use public key to decrypt the token with ES256 scheme  
 decoded = ***jwt***.verify(token, cert, {algorithm: 'ES256',  
 } );  
  
 // check that the scope in the token matches the scope passed in as a param  
 if (!decoded.role || decoded.role != 'admin') { // this allows admin user to go to any page  
 if (!decoded.scope || decoded.scope != scope) throw new ***Error***('scope is not permitted');  
 }  
 } catch (err) {  
 ***console***.log(`JWT Error:\n ${err}`);  
 // redirect back to login page and specify the error message in the query string  
 decoded = {redirectURL: '/secure/?err=' + err.message};  
 }  
 return decoded;  
};

Restart the webserver for the changes to take effect - Navigate to <http://localhost:3000/examples/secure/>

Nothing should change other than the files are being used for the JWT

# Passport

In the previous sections we create our own process for checking the JWT, but there exists a very popular package that can be used to check the JWT easier.

The package is called Passport and it can not only verify the JWT but many other ways to authenticate a user to a web app

Passport is flexible authentication middleware for Node.js.

* Passport can be implemented quite easily in any Express-based web application.
* Over 500 authentication strategies support authentication using a username and password, Facebook, Twitter, and more
* Easily handles success and failure
* Make your own custom strategies
* Lightweight code base

Learn More about Passport: <http://www.passportjs.org/docs/>

## Verify Callback (<http://www.passportjs.org/docs/authenticate/>)

This example introduces an important concept. Strategies require what is known as a *verify callback*. The purpose of a verify callback is to find the user that possesses a set of credentials.

When Passport authenticates a request, it parses the credentials contained in the request. It then invokes the verify callback with those credentials as arguments, in this case username and password. If the credentials are valid, the verify callback invokes done to supply Passport with the user that authenticated.

**return** done(null, user);

If the credentials are not valid (for example, if the password is incorrect), done should be invoked with false instead of a user to indicate an authentication failure.

**return** done(null, false);

An additional info message can be supplied to indicate the reason for the failure. This is useful for displaying a flash message prompting the user to try again.

**return** done(null, false, { message: 'Incorrect password.' });

Finally, if an exception occurred while verifying the credentials (for example, if the database is not available), done should be invoked with an error, in conventional Node style.

**return** done(err);

Note that it is important to distinguish the two failure cases that can occur. The latter is a server exception, in which err is set to a non-null value. Authentication failures are natural conditions, in which the server is operating normally. Ensure that err remains null, and use the final argument to pass additional details.

By delegating in this manner, the verify callback keeps Passport database agnostic. Applications are free to choose how user information is stored, without any assumptions imposed by the authentication layer.

# Use Passport to Verify the JWT (does not encode a JWT)

In the terminal run

npm i passport passport-jwt

First, we need to configure passport in the app.js

**\routes\secure.js - add code in violet**

const createError = require('http-errors');  
const ***express*** = require('express');  
const ***path*** = require('path');  
const cookieParser = require('cookie-parser'); // exposes the req.cookies property  
const logger = require('morgan');  
// require passport to make authentication / authorization easier  
const ***passport*** = require('passport');

.

.

.

***app***.use(logger('dev'));  
***app***.use(***express***.***json***());  
***app***.use(***express***.***urlencoded***({extended: false}));  
***app***.use(cookieParser());// should have the same secret as the session cookie  
***app***.use(***express***.***static***(***path***.join(\_\_dirname, 'public')));  
// configure passport to create the req.currentUser property with the token payload  
// in order to use passport with express we need to initialize passport  
***app***.use(***passport***.initialize({userProperty: 'currentUser'}));

Passport will now add a new property to the request object – **req.currentUser**

We must then further configure passport for our secure section

Normally the passport config would be done in a separate file and then required in all the routers, but for this example we will just add it to the secure.js

**\routes\secure.js - add the code below**

// require passport and configure to use the JWT scheme  
// remember passport can use over 500 different schemes to authenticate  
// https://www.passportjs.org/packages/passport-jwt/  
const ***passport*** = require('passport');  
// declare alias for the JWT Strategy  
const JwtStrategy = require('passport-jwt').Strategy;  
// declare alias for Extractor - because we are not using the default Authenticate Header for the token  
const ExtractJwt = require('passport-jwt').***ExtractJwt***; // set of pre-defined extractors - where to get the token  
  
// configure passport to use JWT  
***passport***.use( new JwtStrategy({

// look the token in the URL or BODY specifically the access\_token parameter

jwtFromRequest: ExtractJwt.fromExtractors([  
 ExtractJwt.fromUrlQueryParameter('access\_token'),  
 ExtractJwt.fromBodyField('access\_token'),  
 ]),  
 // specify the public key we need to decrypt the token  
 secretOrKey: ***fs***.readFileSync('es256public.pem'), // get file buffer of binary file data  
 algorithm: 'ES256',  
},  
(payload, done)=>{ // verify handler  
 // ensure the role is allowed  
 if (!['admin', 'employee', 'client'].includes(payload.role)) {  
 // specify that the role is invalid  
 return done(null, false, {message: 'Specified role is not allowed'} );  
 }  
 // ensure the scope is allowed  
 if (!['dashboard', 'profile', 'booking'].includes(payload.scope)) {  
 // specify that the scope is invalid  
 return done(null, false, {message: 'Specified scope is not allowed'} );  
 }  
 // return a success notification to the passport process  
 // normally we would use the payload to search a database then pass along a user object  
 // - but for new we are passing along the payload from the JWT  
 return done(null, payload);  
}),  
);

Next we can use the middleware in and handler, but let’s just then to the passport get and post handlers

*/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \* GET PASSPORT - http://localhost:3000/secure/passport  
 \*/****router***.get('/passport', ***passport***.authenticate('jwt', {  
 session: false, // be default passport stores the data in a session - but not for us in this example  
 failureRedirect: '/secure/?err=jwt+not+verified', // catch all unauthorized error action  
}),  
(req, res, next)=>{  
 // when inside the handler function we can perform additional authorization  
 // passport has expose a new req.currentUser property which contains the token payload becaue that is what we  
 // passed along in the passport.use function  
  
 // Example check that user is an admin or employee  
 if (!['admin', 'employee'].includes(req.currentUser.role)) {  
 res.redirect('/secure/?err=insufficient+permissions');  
 } else {  
 res.render('secure-generic', {  
 title: 'GET - PASSPORT',  
 token: req.query['access\_token'],  
 payload: req.currentUser,

action: req.baseUrl+req.path, // result '/secure/passport/'  
 });  
 }  
});

*/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \* POST PASSPORT  
 \*/*// POST content for path: http://localhost:3000/secure/passport/  
***router***.post('/passport', ***passport***.authenticate('jwt', {  
 session: false,  
 failureRedirect: '/secure/?err=jwt+expired',  
}),  
(req, res, next)=>{  
   
 if (!['admin', 'employee'].includes(req.currentUser.role)) {  
 res.redirect('/secure/?err=insufficient+permissions');  
 } else {  
 res.render('secure-generic', {  
 title: 'POST - PASSPORT',  
 token: req.body['access\_token'],  
 payload: req.currentUser,  
 });  
 }  
});

Learn more about the other authorize options available: <http://www.passportjs.org/docs/authorize/>