**Authentication and Authorization in Node.js**

Previously we have learned how to create JSON Web Token (JWT) by demo login data, passed through request header to verify user by using middleware. Before starting with this post it’s recommended to overview previous post on ” [Token-Based Authentication In Node.js Using JWT](https://www.dotnettricks.com/learn/nodejs/token-based-authentication-using-json-web-token)”.

In this article we are going to learn how to perform user authentication using “*Passport*” then create JWT token to verify user with access permission on each request. We are going to use MSSQL server for database operations. Previously we have tested the sample application using postman, in this post we are going to configure handlebars as view engine to prepare user interface for user operations.

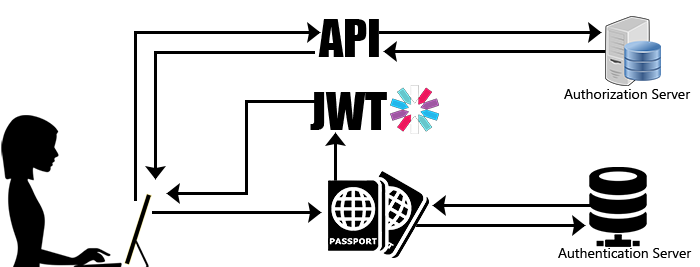
Download the previous sample from above link to get started with or you may start with a new sample by getting help from this post “[Getting Started with ExpressJS](https://www.dotnettricks.com/learn/nodejs/getting-started-with-expressjs)”. I am going to use the previous sample application and make the necessary changes in it to configure the view engine with further authentication and authorization process step by step. First let’s clarify the new term called “Passport”.

**What is passport?**

**According to passportjs.org:** “*Passport is authentication middleware for Node.* *It is designed to serve a singular purpose: authenticate requests.”*

Simply we can say passport is only for authenticate user, that has different authentication options like local and third party (Facebook, Twitter, Github etc).

We are going to implement local authentication in our sample application using passport local strategy. Overview the full process from following diagram.



Let’s get started with package installation and implementing passport for authentication. Later we will learn how to implement the authorization based on token and custom user permission.

**1. Authentication:**

First install all required package using following command:

*“npm install express-handlebars mssql nodemon passport passport-local”*

**Passport:**

After successful package installation go to app.js, enable passport module with following lines of code.

var passport = require('passport');

app.use(passport.initialize());

**Handlebars:**

Let’s configure view engine, in app.js register the view engine with the following code snippet. As you can see from below code sample we have commented the auto generated view engine.

Here app.engine() is registering the template engine to our application, then set the engine as hbs using app.set() function.

var exphbs = require('express-handlebars');

// view engine setup

/\* app.set('views', path.join(\_\_dirname, 'views'));

app.set('view engine', 'jade'); \*/

app.engine('.hbs', exphbs({ defaultLayout: 'main', extname: '.hbs' }));

app.set('view engine', '.hbs');

For more information on handlebars, follow the link: <https://www.npmjs.com/package/express-handlebars>

Add new layout page with the extension of .hbs in views>layouts folder, then copy paste below html to the newly created file.

**Main Layout:**

As you can see the main layout is nothing but a HTML page with render section {{body}}, where other pages going to render with different route path.

<!DOCTYPE html>

<html>

<head>

<title>{{title}}</title>

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="stylesheet" href="/stylesheets/bootstrap.min.css" crossorigin="anonymous">

<link rel='stylesheet' href='/stylesheets/style.css' />

</head>

<body>

<div class="container">

{{{body}}}

</div>

</body>

</html>

**Index Route:**

Following code snippet will render the index.hbs page on application startup with passing object.

var express = require('express');

var router = express.Router();

/\* GET home page. \*/

router.get('/', function (req, res, next) {

res.render('index', { title: 'Welcome to ExpressJS + Passport + JSON Web Token (JWT)' });

});

module.exports = router;

Let’s modify the index page with the below HTML code sample.

**Index page:**

<div class="container">

<div class="row">

<h3>{{ this.title }} <img src="../images/jwt.png" alt="" width="30px"></h3>

<hr>

<a href="/users" class="btn btn-warning">User Login </a>

<hr>

<p><b>From jwt.io:</b> JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and

self-contained way for

securely transmitting information between parties as a JSON object. This information can be verified and

trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a

public/private key pair using RSA or ECDSA.</p>

<img src="../images/process.png" alt="" width="700px">

<br><br>

<p><b>From passportjs.org:</b> Passport is authentication middleware for Node. It is designed to serve a

singular purpose: authenticate requests.</p>

</div>

</div>

Next create a login page where user can submit form and login by providing login details. Copy paste below code sample to login page.

**Login Page:**

<div class="container">

<form action="/users" method="POST">

<div class='row'>

<h3>User Login</h3>

<hr>

<div class='col-sm-4'>

<div class='form-group'>

<label for="username">User Name</label>

<input class="form-control" id="username" name="username" size="250" type="text" required="true" />

</div>

</div>

<div class='col-sm-4'>

<div class='form-group'>

<label for="password">Password</label>

<input class="form-control" id="password" name="password" required="true" size="250" type="password" />

</div>

</div>

</div>

<div class='row'>

<div class="col-md-1">

<button type="button" class="btn btn-warning" onclick="location.href='/';">Back</button>

</div>

<div class="col-md-1">

<button type="button" class="btn btn-danger" onclick="location.href='/users';">Reset</button>

</div>

<div class="col-md-2">

<button type="submit" class="btn btn-success">Login</button>

</div>

</div>

</form>

</div>

In user.js file, change previous response by rendering “users” view with empty object passing.

/\* GET users listing. \*/

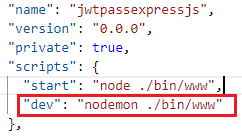
router.get('/', function (req, res, next) {

/\* res.send('respond with a resource'); \*/

res.render('users', {});

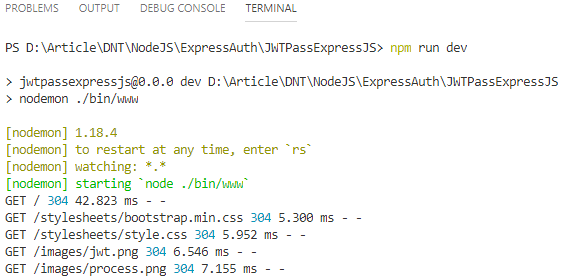
});

Go to package.json then add new script object "dev": "nodemon ./bin/www" like below image



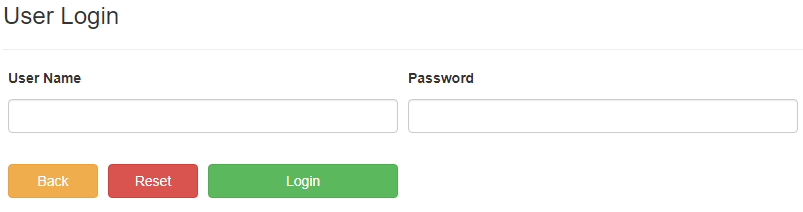
Nodemon will automatically restart our application when we make any save changes to our application.

Let’s run the application using command “npm run dev” like following image



As you can see the application has run with port 3000. Go to brouser then browse with URL “http://localhost:3000”

From home page, click on button “User Login” to see below login screen:



As you can see the login page is rendered with user login form.

So far we have successfully implemented the view engine to our sample application. Next we are going to authenticate user by using the login form with local passport authentication. Create a database by using below SQL script.

**Create Database**

USE [master]

GO

/\*\*\*\*\*\* Object: Database [ExpressJWTPass] Script Date: 9/14/2018 11:38:36 AM \*\*\*\*\*\*/

CREATE DATABASE [ExpressJWTPass]

GO

USE [ExpressJWTPass]

GO

/\*\*\*\*\*\* Object: Table [dbo].[users] Script Date: 9/14/2018 11:38:36 AM \*\*\*\*\*\*/

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

CREATE TABLE [dbo].[users](

[userId] [int] NOT NULL,

[userName] [nvarchar](50) NULL,

[password] [nvarchar](50) NULL,

[usertype] [nvarchar](50) NULL,

[canEdit] [bit] NULL,

[canDelete] [bit] NULL,

[canCreate] [bit] NULL,

[canView] [bit] NULL,

CONSTRAINT [PK\_users] PRIMARY KEY CLUSTERED

(

[userId] ASC

)WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]

) ON [PRIMARY]

GO

INSERT [dbo].[users] ([userId], [userName], [password], [usertype], [canEdit], [canDelete], [canCreate], [canView]) VALUES (1, N'shashangka', N'12345', N'admin', 1, 1, 1, 1)

INSERT [dbo].[users] ([userId], [userName], [password], [usertype], [canEdit], [canDelete], [canCreate], [canView]) VALUES (2, N'ishani', N'12345', N'editor', 0, 0, 1, 1)

/\*\*\*\*\*\* Object: StoredProcedure [dbo].[GetUserAuthentication] Script Date: 9/14/2018 11:38:36 AM \*\*\*\*\*\*/

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

-- =============================================

-- Author: <Author,,Name>

-- Create date: <Create Date,,>

-- Description: <Description,,>

-- =============================================

CREATE PROCEDURE [dbo].[GetUserAuthentication]

-- Add the parameters for the stored procedure here

@username nvarchar(50),

@password nvarchar(50)

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

-- Insert statements for procedure here

SELECT [userId],[userName],[usertype]--,[canEdit],[canDelete],[canCreate],[canView]

FROM [dbo].[users]

WHERE [userName] = @username AND [password] = @password

END

GO

For now we are done with the database, next we will create a service module to interact with database.

**Data Service:**

Create a folder named “data”, then add the service module named “dbService.js”. Copy paste below code snippet to the newly created file.

Change the config section according to your SQL server details. Here the max pool is 10 & min is 0 & default connection timeout is 30000 millisecond.

var dbConfig = {

user: "user",

password: "password",

server: " YourDatabaseServer",

database: " YourDatabaseName",

port : 1433,

pool: {

max: 10,

min: 0,

idleTimeoutMillis: 30000

}

};

Get more details on Pool by following link: <https://github.com/coopernurse/node-pool>

**ConnectionPool:**

In query execute section create connection object using connectionpool.

const conn = new mssql.ConnectionPool(dbConfig);

If the connection is ok then perform database operation using the opened connection.

const req = new mssql.Request(conn);

**Finally the service:**

var mssql = require('mssql');

var dbConfig = {

user: "sa",

password: "sa@12345",

server: "DESKTOP-DUDGUJT\\SQLEXPRESS",

database: " ExpressJWTPass",

port : 1433,

pool: {

max: 10,

min: 0,

idleTimeoutMillis: 30000

}

};

var executeQuery = function (sql, res) {

const conn = new mssql.ConnectionPool(dbConfig);

conn.connect().then(function () {

const req = new mssql.Request(conn);

req.query(sql).then(function (data) {

res(data);

}).catch(function (err) {

res(null, err);

})

}).catch(function (err) {

res(null, err);

})

}

module.exports = {

executeQuery

}

Get more details on SQL connection by following link: <https://www.npmjs.com/package/mssql#connections-1>

In the next portion we are going to check user login details from database, if the user is valid then we are creating the JWT and passing it back to the client to store it.

By using cookies we are storing the user information and generated token to client machine so that client can use it on each request to revalidate the token and get API access according to user access permission.

**Authentication:**

Let’s add passport.js file for passport local authentication, include the required module packages, as we are using passport local strategy to authenticate the user so we have included the “*passport-local*” module package by creating object named “*strategy*”.

**passport.js:**

var passport = require('passport');

var strategy = require('passport-local');

var dbService = require('../data/dbService');

let jwt = require('jsonwebtoken');

passport.use(new strategy({ session: false }, function (username, password, callback) {

//Stored Procedure

var query = "[GetUserAuthentication] '" + username + "', '" + password + "'";

//Get Data From Database

dbService.executeQuery(query, function (data, err) {

if (err) {

callback(null, err);

} else {

var result = data.recordset

if (result.length > 0) {

let token = jwt.sign(result[0], global.config.secretKey, {

algorithm: global.config.algorithm,

/\* expiresIn: '1m' \*/

});

callback({ user: result[0], token: token });

}

else {

callback({ user: null, token: null });

}

}

});

}));

module.exports = passport;

From above code sample, using “*passport.use()”* function we are going to validate user by passing local account details. We are not using any session to storing authentication information by defining session value false. If the user is valid then a token is generated by JWT. The generated token is stored in client machine for further API request.

Next client is submitting the post request with user login data for authentication using “*passport.authenticate()”* function. This is where we are storing the returned token and logged user information in client browser after successful authentication.

**HTTP Login POST:**

/\* Post users Login. \*/

router.post('/', function (req, res, next) {

passport.authenticate('local', function (data, err) {

if (err) {

res.redirect('/users?status=' + encodeURIComponent('Error Login!!'));

console.log(err.name + ':' + err.message);

} else {

if (data.user != null) {

res.cookie('jwtoken', data.token);

res.cookie('loggeduser', data.user);

res.redirect('/');

}

else {

res.redirect('/users?status=' + encodeURIComponent('Incorrect login details!!'));

}

}

})(req, res, next);

});

Let’s modify our index page with the below code to display the logged user details with logout button.

**Index Page:**

{{#if user }}

<h4>Hello, {{ user.userName }} [{{ user.usertype }}]</h4>

<a href="/users/logout" class="btn btn-danger">Logout</a>

{{ else }}

<a href="/users" class="btn btn-warning">User Login </a>

{{/if}}

Using below code sample, on user logout button click application is clearing the stored information and redirecting to user login page.

**HTTP Logout GET:**

// GET user Logout

router.get('/logout', function (req, res, next) {

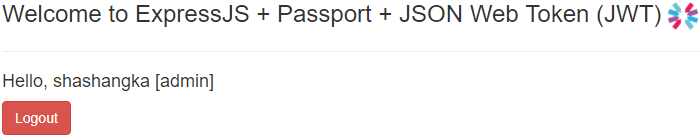
res.clearCookie('jwtoken');

res.clearCookie('loggeduser');

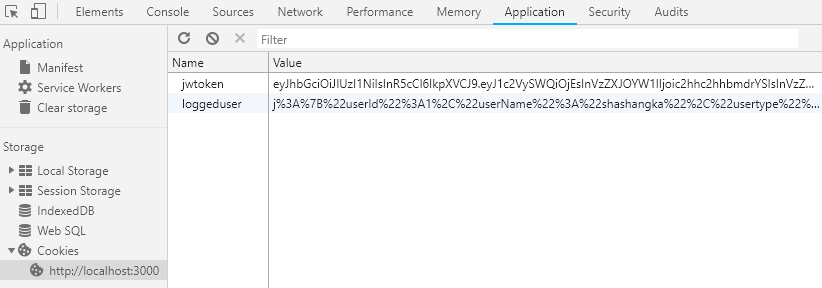
res.redirect('/users');

});

Let’s test the application at this stage with user authentication, go to login page fill the login information.



As you can see from above image, user login successful with displaying the logged user details. Now go to application tab in browser to see the cookies where the token and logged user information is stored.

****

Next section we are going to manage list of customers in database by particular user access authorization. This is where we are protecting the API’s from unauthorized access by using the token which is going to validate on each request.

**2. Authorization:**

Let’s create a customer table to store customer’s information. Below script will create the required table and stored procedure. Copy paste the script to query window then execute.

USE [ExpressJWTPass]

GO

/\*\*\*\*\*\* Object: Table [dbo].[customers] Script Date: 9/14/2018 11:38:36 AM \*\*\*\*\*\*/

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

CREATE TABLE [dbo].[customers](

[customerId] [int] NOT NULL,

[customerName] [nvarchar](50) NULL,

[customerContact] [nvarchar](50) NULL,

[customerEmail] [nvarchar](50) NULL,

CONSTRAINT [PK\_customers] PRIMARY KEY CLUSTERED

(

[customerId] ASC

)WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]

) ON [PRIMARY]

GO

/\*\*\*\*\*\* Object: StoredProcedure [dbo].[DeleteCustomersByID] Script Date: 9/14/2018 11:38:36 AM \*\*\*\*\*\*/

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

-- =============================================

-- Author: <Author,,Name>

-- Create date: <Create Date,,>

-- Description: <Description,,>

-- =============================================

CREATE PROCEDURE [dbo].[DeleteCustomersByID]

-- Add the parameters for the stored procedure here

@customerId INT

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

-- Insert statements for procedure here

DELETE FROM [dbo].[customers] WHERE [customerId] = @customerId

END

GO

/\*\*\*\*\*\* Object: StoredProcedure [dbo].[GetCustomers] Script Date: 9/14/2018 11:38:36 AM \*\*\*\*\*\*/

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

-- =============================================

-- Author: <Author,,Name>

-- Create date: <Create Date,,>

-- Description: <Description,,>

-- =============================================

CREATE PROCEDURE [dbo].[GetCustomers]

-- Add the parameters for the stored procedure here

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

-- Insert statements for procedure here

SELECT [customerId],[customerName],[customerContact],[customerEmail]

FROM [dbo].[customers]

END

GO

/\*\*\*\*\*\* Object: StoredProcedure [dbo].[GetCustomersByID] Script Date: 9/14/2018 11:38:36 AM \*\*\*\*\*\*/

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

-- =============================================

-- Author: <Author,,Name>

-- Create date: <Create Date,,>

-- Description: <Description,,>

-- =============================================

CREATE PROCEDURE [dbo].[GetCustomersByID]

-- Add the parameters for the stored procedure here

@customerId INT

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

-- Insert statements for procedure here

SELECT [customerId],[customerName],[customerContact],[customerEmail]

FROM [dbo].[customers] WHERE [customerId] = @customerId

END

GO

/\*\*\*\*\*\* Object: StoredProcedure [dbo].[GetUserAuthorization] Script Date: 9/14/2018 11:38:36 AM \*\*\*\*\*\*/

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

CREATE PROCEDURE [dbo].[GetUserAuthorization]

@Userid NVarchar(250),

@Methodtype NVarchar(250)

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

-- Insert statements for procedure here

SELECT CASE WHEN result = 1 THEN 'true' ELSE 'false' END AS permission FROM

(

SELECT CASE

WHEN @Methodtype = 'GET' THEN [CanView]

WHEN @Methodtype = 'POST' THEN [canCreate]

WHEN @Methodtype = 'PUT' THEN [canEdit]

WHEN @Methodtype = 'DELETE' THEN [CanDelete]

ELSE 0

END AS result

FROM [dbo].[users] WHERE [userId] = @Userid

)AUTH

END

--EXEC GetUserAuthorization '2', 'POST'

GO

/\*\*\*\*\*\* Object: StoredProcedure [dbo].[SaveCustomer] Script Date: 9/14/2018 11:38:36 AM \*\*\*\*\*\*/

SET ANSI\_NULLS ON

GO

SET QUOTED\_IDENTIFIER ON

GO

-- =============================================

-- Author: <Author,,Name>

-- Create date: <Create Date,,>

-- Description: <Description,,>

-- =============================================

CREATE PROCEDURE [dbo].[SaveCustomer]

-- Add the parameters for the stored procedure here

@customerId INT,

@customerName Nvarchar(50),

@customerContact Nvarchar(50),

@customerEmail Nvarchar(50)

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

IF(@customerId>0)

BEGIN

Update [dbo].[customers]

SET [customerName] = @customerName,

[customerContact] = @customerContact,

[customerEmail] = @customerEmail

Where [customerId] = @customerId

END

ELSE

BEGIN

DECLARE @custId INT

SET @custId = ISNULL(((SELECT MAX(customerId) FROM [dbo].[customers])+1),'1');

-- Insert statements for procedure here

Insert Into [dbo].[customers]

([customerId],[customerName],[customerContact],[customerEmail])

Values

(@custId,@customerName,@customerContact,@customerEmail)

END

END

GO

After successful script execution let’s create a customer data service to perform CRUD operations in database. Go to data folder then create “*customerService.js*” file.

Copy paste below code sample to newly created CRUD service module.

**Customer Data Service:**

var dbService = require('../data/dbService');

var getData = function (req, res, callback) {

//Stored Procedure

let query = "[GetCustomers]";

//Database Query

dbService.executeQuery(query, function (data, err) {

if (err) {

callback(null, err);

} else {

callback(data);

}

});

};

var getDataById = function (req, res, callback) {

let customerId = req.params.id; let status = req.query.status;

//Stored Procedure

var query = "[GetCustomersByID] " + customerId + "";

//Database Query

dbService.executeQuery(query, function (data, err) {

if (err) {

callback(null, err);

} else {

callback(data);

}

});

};

var postData = function (req, res, callback) {

let customerId = 0;

if (isNaN(parseInt(req.body.customerId)))

customerId = 0;

else

customerId = parseInt(req.body.customerId);

//Stored Procedure

var query = "[SaveCustomer] " + customerId + ", '" + req.body.customerName + "', '" + req.body.customerContact + "', '" + req.body.customerEmail + "'";

//Database Query

dbService.executeQuery(query, function (data, err) {

if (err) {

callback(null, err);

} else {

callback(data);

}

});

};

var deleteData = function (req, res, callback) {

let customerId = req.params.id;

//Stored Procedure

var query = "[DeleteCustomersByID] " + parseInt(customerId) + "";

//Database Query

dbService.executeQuery(query, function (data, err) {

if (err) {

callback(null, err);

} else {

callback(data);

}

});

};

module.exports = {

getData,

getDataById,

postData,

deleteData

};

**Verify Token:**

Let’s modify the existing token verification module for verify process. This is where the stored token is retrieved by “req.cookies.jwtoken” for use, in previous example we used “*x-access-token*” to pass token string.

Here if token is valid, the authorization is done with the below access permission retrieving from database based on user and given permission.

Below sample access permission data worked with different logged user for multiple operations.

jwt10

**verifytoken.js:**

var express = require('express');

var router = express.Router();

var dbService = require('../data/dbService');

var jwt = require('jsonwebtoken');

router.use(function (req, res, next) {

var token = req.headers['x-access-token'] || req.cookies.jwtoken;

var httpMethod = null; //var httpMethod = req.method;

var originalUrl = req.originalUrl;

if (originalUrl.includes("save")) httpMethod = 'POST';

else if (originalUrl.includes("edit")) httpMethod = 'PUT';

else if (originalUrl.includes("delete")) httpMethod = 'DELETE';

else httpMethod = 'GET';

if (token) {

jwt.verify(token, global.config.secretKey,

{

algorithm: global.config.algorithm

}, function (err, decoded) {

if (err) {

let errordata = {

message: err.message,

expiredAt: err.expiredAt

};

console.log(errordata);

return res.render('customers', { title: 'Unauthorized Access', status: 'Token Expired at ' + err.expiredAt });

}

req.decoded = decoded;

//Stored Procedure

var query = "[GetUserAuthorization] '" + decoded.userId + "', '" + httpMethod + "'";

//Get Authorization From Database

dbService.executeQuery(query, function (data, err) {

if (err) {

console.log(err.name + ':' + err.message);

} else {

var result = data.recordset[0].permission;

if (result == 'true') {

next();

}

else {

console.log('Unauthorized Access!!');

if (httpMethod == 'GET') {

return res.redirect('/');

}

else {

return res.redirect('/customers?status=' + encodeURIComponent('Operation Restricted!!'));

}

}

}

});

});

} else {

return res.redirect('/');

}

});

module.exports = router;

We are almost done with our authorization process. Modify the existing customer routes by following code sample. As you can see the “verifyToken” middleware is passed as argument with the each HTTP request method to verify the user access validity. If the request is valid then application is going to proceed otherwise it’ll response redirect with status message passed by query string.

**Customer Route:**

var express = require('express');

var router = express.Router();

var dbService = require('../data/dbService');

var custService = require('../data/customerService');

let verifyToken = require('./verifytoken');

/\* GET customers listing. \*/

router.get('/', verifyToken, function (req, res, next) {

let status = req.query.status;

//Service Call

custService.getData(req, res, function (data, err) {

if (err) {

res.render('customers', { message: 'Error Loading Data!!' });

console.log(err.name + ':' + err.message);

} else {

res.render('customers', { customerdata: data.recordset, message: status });

}

});

});

/\* GET customers ByID \*/

router.get('/edit/:id', verifyToken, function (req, res, next) {

let status = req.query.status;

custService.getDataById(req, res, function (data, err) {

if (err) {

res.render('customers', { message: 'Error Binding Data!!' });

console.log(err.name + ':' + err.message);

} else {

var customeredit = data.recordset[0];

//Service Call

custService.getData(req, res, function (data, err) {

if (err) {

res.render('customers', { message: 'Error Loading Data!!' });

console.log(err.name + ':' + err.message);

} else {

res.render('customers', { customerdata: data.recordset, customer: customeredit, message: status });

}

});

}

});

});

/\* POST customer \*/

router.post('/save', verifyToken, function (req, res, next) {

let status = req.query.status;

//Service Call

custService.postData(req, res, function (data, err) {

if (err) {

res.render('customers', { message: 'Error Saving Data!!' });

console.log(err.name + ':' + err.message);

} else {

res.redirect('/customers');

}

});

});

/\* DELETE Delete customers \*/

router.get("/delete/:id", verifyToken, function (req, res) {

let status = req.query.status;

//Service Call

custService.deleteData(req, res, function (data, err) {

if (err) {

res.render('customers', { message: 'Error Deleting Data!!' });

console.log(err.name + ':' + err.message);

} else {

res.redirect('/customers');

}

});

});

module.exports = router;

Let’s add an access button in index page navigating to customer route:

<a href="/customers" class="btn btn-primary">Manage Customers</a>

**Customer Page:**

Finally create customer page in views folder by naming it “customers.hbs”, copy paste below code sample

<div class="container">

<form action="/customers/save" method="POST">

<div class='row'>

<h3>Customer Info {{#if message }} - {{message}}{{/ if }}</h3>

<hr>

<input type="hidden" name="customerId" value="{{customer.customerId}}" />

<div class='col-sm-4'>

<div class='form-group'>

<label for="customerName">Customer Name</label>

<input class="form-control" id="customerName" name="customerName" size="250" type="text" required="true"

value="{{customer.customerName}}" />

</div>

</div>

<div class='col-sm-4'>

<div class='form-group'>

<label for="customerContact">Contact</label>

<input class="form-control" id="customerContact" name="customerContact" required="true" size="250"

value="{{customer.customerContact}}" type="text" />

</div>

</div>

<div class='col-sm-4'>

<div class='form-group'>

<label for="customerEmail">Email</label>

<input class="form-control" id="customerEmail" name="customerEmail" required="true" size="250" type="email"

value="{{customer.customerEmail}}" />

</div>

</div>

</div>

<div class='row'>

<div class="col-md-1">

<button type="button" class="btn btn-warning" onclick="location.href='/';">Back</button>

</div>

<div class="col-md-1">

<button type="button" class="btn btn-danger" onclick="location.href='/customers';">Reset</button>

</div>

<div class="col-md-2">

<button type="submit" class="btn btn-success">Submit</button>

</div>

</div>

</form>

<div class='row'>

{{#if customerdata }}

<h3>Customer List</h3>

<hr>

<table class="table table-striped">

<thead>

<tr>

<th>Id</th>

<th>Customer Name</th>

<th>Contact</th>

<th>Email</th>

<th>Option</th>

</tr>

</thead>

{{#each customerdata}}

<tr>

<td>{{this.customerId}}</td>

<td>{{this.customerName}}</td>

<td>{{this.customerContact}}</td>

<td>{{this.customerEmail}}</td>

<td>

<a href="/customers/edit/{{this.customerId}}" title="Edit Record" class="btn btn-primary btn-xs pull-right">

Edit

</a>

<a href="/customers/delete/{{this.customerId}}" title="Delete Record" class="btn btn-danger btn-xs pull-right" onclick="return confirm('You are about to delete {{this.customerName}}, are you sure?')">

Delete

</a>

</td>

</tr>

{{/each}}

</table>

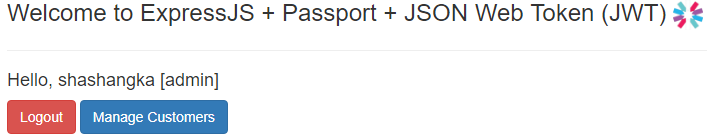
{{/if}}

</div>

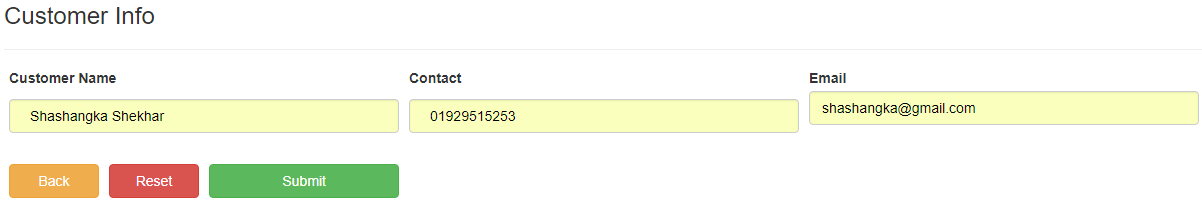
</div>

Yes, we are done with our authorization process, next we are going to test the user access by different logged user.

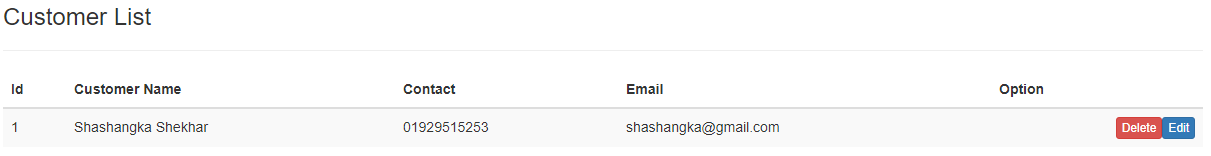
After successful login click on “Manage Customers“ button to navigate customer page.



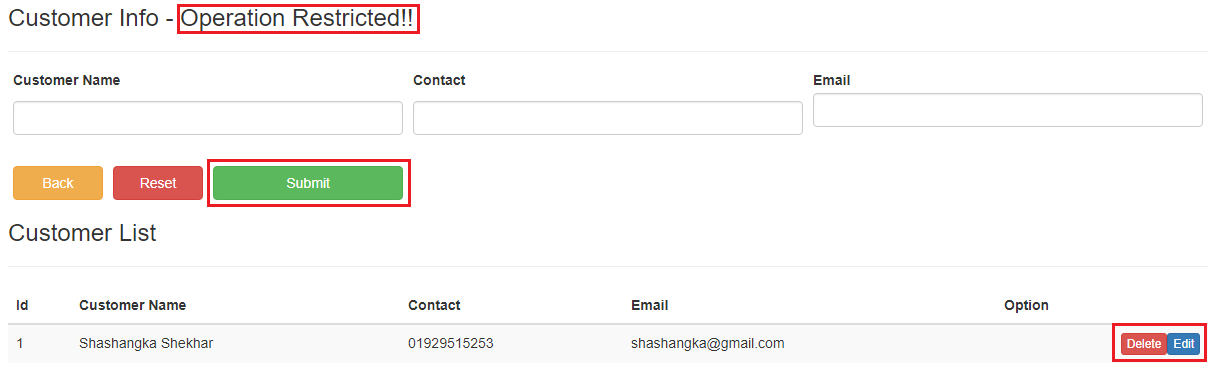
As you can see the customer page is displayed with customer form, create a new customer by submitting with providing customer details.



If the user have the create permission, it will save the customer to database and list the customer in below table like below screen.



Otherwise it will response with “Operation Restriction” message.



So far, we have seen the whole process of authentication and authorization step by step, learned how we can protect API’s from unwanted access, let’s summarize the process and learning.

**Summary:**

In this article, I have covered the following things:

* Configuring view engine.
* Enabled and used passport local for authentication.
* Generate JWT on passport authentication.
* Using JWT to secure ExpressJS API
* Using router level middleware
* Using cookies in browser to stored logged information.
* Retrieving values from cookies.
* Using custom user authorization on token validation

Hope the article will help you to create secured API’s using Passport and JWT in ExpressJS application.