**Passport in Node.JS with PostgreSQL (and cryptography)**

One of these days I found myself having to code a login system in Node.JS and PostgreSQL, I had a bit of trouble doing it and the internet didn’t seem to help me a lot. How this can happen? Maybe because the duo Node.JS/PostgreSQL is not as used as Node.JS/Mongo, and as a consequence we don’t have enough support/documentation on using SQL databases on Node. Leaving that aside, there were two articles that helped me a lot in the process, they were:

* Node, Passport and Postgres from Michael Herman (<http://mherman.org/blog/2016/09/25/node-passport-and-postgres/#.WbqMIq2ZPow)> - The Problem is that, first, he is using galvanize and knex and I’m not into these kind of stuff, I’m just using the regular PostgreSQL Promise (pgp). Also, he seems to skip some steps.
* Postgres with Passport from Carol Louie (<http://uitblog.com/postgres-with-passport/)> - This one game me some insights and she evens talks about not having tutorials about this on the internet, but this one is a bit incomplete.

That being said, my plan here is to do some kind of tutorial the set you up and going with a fully functional Authentication System, including having hashed passwords on your database.

**What are we going to do?**

The idea here is to create a simple Secret Page, it is a secret so you have to be logged in to see it. To be logged in you have to sign up. That’s about it.

**What are we going to use?**

* Node.JS
* Express
* PostgreSQL Promise (pg-p)
* PassportJS
* Passport-Local
* Bcrypt

This guide/tutorial presumes that you are fairly comfortable with JavaScript, Node.JS, Express and PostgreSQL so when we talk about them, they are not going to be explained in depth, just some steps to follow.

These steps will be very familiar for the ones who have experience on these technologies. What will be better explained is Passport, the Postgre Promise Library and Promises overall and BCryptJS.

**Steps:**

1. **Installing Node.JS**

Probably you have already done it… see <https://nodejs.org/en/> for more information

1. **New Node.JS Project**

You should already know these things, but let’s talk about this, to create a new Node.JS project:

Npm init

Fill the details as you always did, my entry point is called **app.js** and not index.js as it suggests, keep that in mind if you use a different name, app.js is the entry point.

1. **Installing Express, doing a Hello World Test**

npm install express --save

**Setting up app.js**

Create the file called **app.js** (or whatever your entry point is called)

Import express and app by using the commands

var express = require('express');

var app = express();

**Let’s listen for connection:**

app.listen(3000, function() {

console.log("The Server is running!");

});

What good are connections without routes? Let’s add a Hello World Test!

app.get("/", function(request, response) {

response.send("Hello World");

});

terminal: node app.js

Access your webpage (localhost:3000/) You should see a Hello World Message!

1. **Creating the Database and Inserting test users**

Create a database called secretpage, it will have just one table, the users table, with a username and a password.

CREATE TABLE secretuser (

username VARCHAR NOT NULL UNIQUE,

password VARCHAR NOT NULL,

CONSTRAINT pk\_user PRIMARY KEY (username)

);

We are going to add three test users to test the pg-promise further on.

INSERT INTO secretUser VALUES('landen', '306');

INSERT INTO secretUser VALUES('nicolette', '839');

INSERT INTO secretUser VALUES('arch', '888');

1. **Setting up pg-promise and performing a sample search query.**

What is Pg Promise? <https://www.npmjs.com/package/pg-promise>

Postgre Promise is built on top of the regular Postgre node library with some additions and the usage of promises.

**What are Promises?** The way promise works might seem a bit tricky at first, but you will end liking it. When you perform the line of code, let’s say, db.searchSomething(), it doesn’t return you the results immediately, it doesn’t guarantee that after the function call the query was executed. The only thing you have is a *promise* that sometime your query will be executed.

Fortunately, “sometime” is fast enough for us, so what we do is, when asking the library to execute a query, we also tell it what we want to do after the query is executed.

Seems safe, eh?

Terminal: npm install pg-promise –save

To keep code clean, we are going to create a different javascript file to handle our database connection, I will call it **dbconnection.js**

**dbconnection.js:**

var pgp = require("pg-promise")();

var db\_obj = {

host: 'localhost',

port: '5432',

database: 'secretPage',

user: 'user',

password: '1234'

}

var db = pgp(db\_obj);

module.exports = db;

Now on **app.js** we just have to require this file and we’re good to go.

**App.js:**

var db = require('./dbconnection.js');

To verify our connection, let’s do a sanity check on the **app.listen** function, trying to estabilish a connection to the database and logging if it had success or not.

**Checking Connection:**

db.connect()

.then(obj => {

console.log("connected to the database");

obj.done()

})

.catch(error => {

console.log("error: ", error.message || error);

});

Now let’s create a route that actually searches for something in our database, let’s just search for all the users in the secretuser table and log it to the console, later we will do a page do show everyone of them on it.

Here we can see the structure of a sample query, in this case we are using the *any* keyword because your query can return any amount of rows, after passing the query we have *then* and, after that, *catch.*

Something like this:

Explain the db.any(query).then(do something).catch(error);

It is a it sounds, It executes the query, returning any amount of rows and after that, it does something, if an error occur, the catch part will be executed.

**Route to test the database:**

app.get("/testdb", function(request, response) {

db.any('SELECT \* FROM secretuser')

.then(function(data) {

console.log(data);

response.send("It Works.");

})

.catch(function(error) {

console.log(error);

response.send("Error");

});

});

That’s it for the database part, for now…

(done, commit.)

1. **Creating the EJS Pages and Routing**

This is pretty dull work so let’s do it fast, we are just going to create a lot of .EJS pages and do the routing our page will need.

Now we’re going to create our pages, we will create 4 pages:

* The Home Page
* Sign Up Page
* Sign In Page
* The Secret Page

This can be tricky (or not, because it isn’t) but we’re actually going to create 7 routes:

* The Home Page Route (get “/”)
* The Sign Up Form (get “/signup”)
* The Sign Up Logic (post “/signup”)
* The Sign In Page (get “/signin”)
* The Sign In Logic (post “/signin”)
* The Secret Page Route (get “/secret”)
* The Logout Route (get “/logout”)

Before doing anything, we need to tell one thing to our app. We need to tell it that we are using ejs pages.

To do this, first we have to install it:

npm install ejs –save

And then, on the app.js file, we have to specify we are using it

app.set("view engine", "ejs");

Now we can create our pages! By the way, the pages on Node.JS are called views, so create a folder called “views” and inside it a file called home.ejs

EJS works just as regular html, but the difference is that we can use some JavaScript code if we need, more on this later.

For now we are just going to create a page with a header and three links, sign in, sign up and logout.

**home.ejs**

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Secret Page</title>

</head>

<body>

<h1>Secret Page</h1>

<ul>

<li><a href="/signup">Sign Up</a></li>

<li><a href="/signin">Sign In</a></li>

<li><a href="/logout">Logout</a></li>

</ul>

</body>

</html>

Now we change the “/” route to

app.get("/", function(request, response) {

response.render("home");

});

Restart the server and see if it works!

Now we are going to create the secret page and the secret route, they won’t be so much of a secret at the moment, but on the next section it wil be.

**Secret.ejs**

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Secret Page</title>

</head>

<body>

<h1>This is the Secret Page</h1>

</body>

</html>

**secret route**

app.get("/secret", function(request, response) {

response.render("secret");

});

**signin.ejs**

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Sign In</title>

</head>

<body>

<h1>Sign In</h1>

<form action="signin" method="post">

<input type="text" name="username" placeholder="username">

<input type="password" name="password" placeholder="password">

<button>Submit</button>

</form>

</body>

</html>

app.get("/signin", function(request, response) {

response.render("signin");

});

**signup.ejs**

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Sign Up</title>

</head>

<body>

<h1>Sign Up</h1>

<form action="signup" method="post">

<input type="text" name="username" placeholder="username">

<input type="password" name="password" placeholder="password">

<button>Submit</button>

</form>

</body>

</html>

app.get("/signup", function(request, response) {

response.render("signup");

});

Now we have our pages and get routes, we just need to create the post routes to start getting filthy, let’s create it with some placeholder responses.

**Sign up route**

app.post("/signup", function(request, response) {

response.send("You hit the signup post route!");

});

**Sign in route**

app.post("/signin", function(request, response) {

response.send("You hit the signin post route!");

});

**Logout Route**

app.get("/logout", function(request, response) {

response.redirect("/");

});

Now things start to get interesting, next section we are going to talk about passport and authentication

1. **Setting Up Passport and Passport-Local**

* **What is Passport and Passport Local?**

Passport is used to authenticate requests. That’s it. Passport uses what is called *Strategies.* We are going to use a Local Strategy, so we are going to use an username and a password on our Node.js application.

Installing passport and passport-local

Npm install passport –save

Npm install passport-local --save

* **How do they work? What they need to work?**

Passport does a lot of things behind the curtains for us. We are going to worry only about what it doesn’t need do and where we need to fill in the gaps.

For the Authentication we need a User model, Passport expects us to do have some things about the user.

1. First of all, a user model.
2. A save function ( User.save ) – A function that will save the user object into the database and return itself.
3. A findOne function ( User.findOne) – A function to see if the user trying to be inserted doesn’t already exist.
4. A findById function ( User.findById) – A function that searches an user by ID and return it, this is used to estabilish sessions.

**A note about sessions:** If you are trying to create an authentication system, chances are that you also want a session to be stabilished, we are going to do that using express-sessions.

Npm install express-session –save

Now we estabilished what we need for the user model, but we also need a passport file. Passport expects us to also write some functions, serializeUser, deserializeUser and also create our Strategies for signing up and logging in.

First of all, let’s say to our app that we are going to use passport, passport-local and that we also want sessions to be used.

App.js

// Passport and Sessions

var passport = require('passport');

app.use(require('express-session')({

secret: 'you can write anything here',

resave: false,

saveUninitialized: false

}));

app.use(passport.initialize());

app.use(passport.session());

About the secret, you can really put anything you want there, but should you? As a rule of thumb, usually we want to use a big random sequence of characters.

We are going to create two files: **user.js** and **passport.js**

**(create each function of each file, saying what they should do)**

1. **Hashing Passwords!**

**Done!**