**Security and authentication:**

“Level 1” authentication (direct storage):

Store a username and password in your database. When the user logs in, check to see if there is a matching username and password.

“Level 2” authentication (encryption):

Caesar cipher – shifts the letters of your message by a certain number.

Use an encryption key to encrypt data. Data becomes encrypted when it’s saved and decrypted when it’s searched for (find()).

Use the module mongoose-encryption (AES) for a simple way to encrypt data using mongoose.

npm i mongoose-encryption

const encrypt = require("mongoose-encryption");

const secret = "thisisourlittlesecret"; //this is your encryption key. Keep this private.

userSchema.plugin(encrypt, { secret: secret, //this assigns your encryption key as the key.

encryptedFields: ['password'] }); //this chooses which fields to encrypt.

Use dotenv to store valuable keys and sensitive information. Ad the .env file to your .gitignore file so it is not committed and available to the public. When hosting on Heroku, it has a place to store all of your private keys and information which is kept offline.

require('dotenv').config(); //add this to the top of your code, as early in the commits as possible.

SECRET=thisisourlittlesecret //this is saved in your .env file in the same directory as your app

process.env.SECRET; //this allows you to access your .env variable.

“Level 3” authentication (hashing):

No encryption is needed as you don’t need to decrypt your data. This method is almost impossible to reverse. No encryption key is needed as well. If you run the same hashing algorithm on the same string, the same encrypted hash will be created.

npm i md5 //installs the md5 module in your directory.

const md5 = require('md5'); //requires the md5 module in the md5 constant.

md5(stringHere); //hashes the string inside the parenthesis

“level 4” authentication (salting)

Salting refers to adding a random set of characters to a user’s password before encryption. These set of random characters is referred to as salt, and they are stored in the database with the user’s username. This way, if two users have the same password, since they have different salts, they will have different encrypted passwords.

Bcrypt is a modern hashing algorithm which adds a random salt to user encrypted passwords. You can adjust the salt rounds (times the password is ran through the hashing algorithm) to make it more secure although this takes exponentially more processing power.

npm i bcrypt

const bcrypt = require("bcrypt");

const saltRounds = 10;

bcrypt.hash(req.body.password, saltRounds, function(err, hash) { password = hash; }

//this hashes a users password with a randomly generated salt

bcrypt.compare(passWord, foundUser.password, function(err, result) { returns true/false }

//this compares the user’s login password with the stored one.

“Level 5” authentication (cookies and sessions):

Cookies are stored locally and store previous session data so server’s can populate your web page with previous events (ex: cart on amazon.com).

Session: A type of cookie. The period when browser interacts with server. Starts when the user logs in and ends when they log out.

Passport is a npm which allows you to create cookies (more specifically sessions) and hash passwords.

npm i passport passport-local passport-local-mongoose express-session

const session = require("express-session");

const passport = require("passport");

const passportLocalMongoose = require("passport-local-mongoose");

Paste this below your app.set and app.use but above your mongoose.connect:

app.use(session({ //use sessions with the specific options

secret: "thisisourlittlesecret2",

resave: false,

saveUninitialized: false }));

app.use(passport.initialize()); //use and initialize passport

app.use(passport.session()); //use passport for sessions

Add these:

mongoose.set('useCreateIndex', true); //removes deprecation warning

userSchema.plugin(passportLocalMongoose); //used to hash/salt passwords. Saves users to DB

passport.use(User.createStrategy());

passport.serializeUser(User.serializeUser());

passport.deserializeUser(User.deserializeUser());

read about this…\*\*\*

“Level 6” authentication (third party OAuth):

OAuth is token based authorization

Why use OAuth:

1. Granular access levels – You can request specific things from the third party accessed.
2. Read / Read + Write access – Can retrieve or retrieve and change information from the third party.
3. Revoke access – The user can revoke access of the from the site in question from the third party site.

Set up site

Redirect to third party

User logs in

User grants access

Your site receives the authentication code (1 time use) or access token (much longer use; more information is accessible)

Read about this as well… \*\*\*