Everything you need to know about the `passport-local` Passport JS Strategy

Zach Gollwitzer
Jan 11, 2020 · 23 min read ★

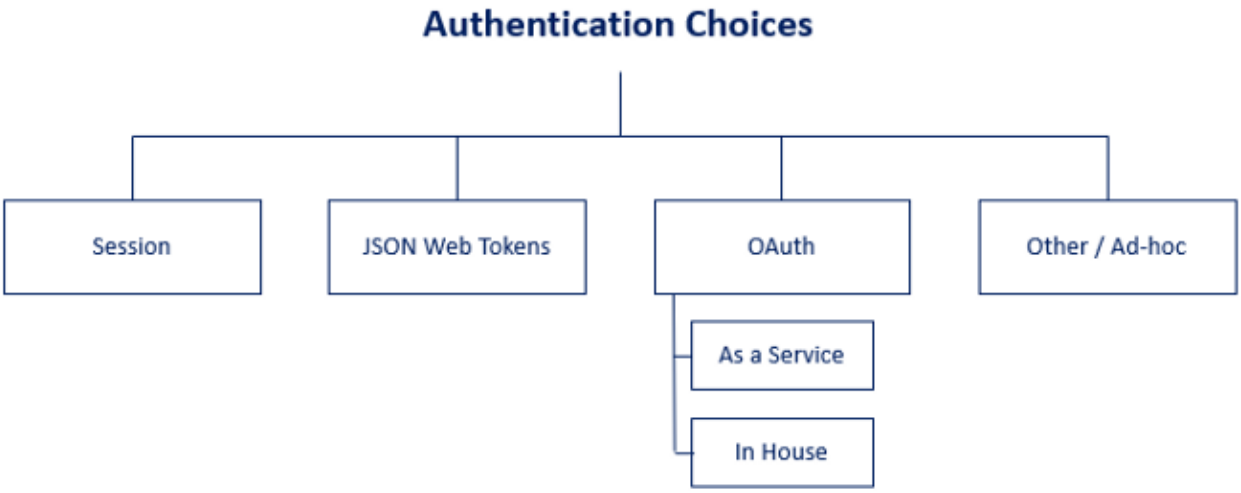
In this post, I am going to walk through why the `passport-local` authentication strategy is a simple, secure solution for small teams and startups implementing a Node/Express web app.

This post is closely tied to my post on the `passport-jwt` strategy, which you can find here.

To help you work through this post, I have created a Github repo with all the code herein: Session Based Auth Repo

Table of Contents

Authentication Choices



Your authentication choices

Above is a high-level overview of the main authentication choices available to developers today. Here is a quick overview of each:

- Session-Based Authentication — Utilizes browser Cookies along with backend "Sessions" to manage logged-in and logged-out users.
- JWT Authentication — A stateless authentication method where a JSON Web token (JWT) is stored in the browser (usually `localStorage`). This JWT has assertions about a user and can only be decoded using a secret that is stored on the server.
- OAuth and OpenID Connect Authentication — A modern authentication method where an application uses "claims" generated from other applications to authenticate its own users. In other words, this is federated authentication where an existing service (like Google) handles the authentication and storage of users while your application leverages this flow to authenticate users.

One note I'll make–Oauth can become confusing really quickly, and therefore is not fully explored in this post. Not only is it unnecessary for a small team/startup getting an application off the ground, but it is also slightly different depending on which service you are using (i.e. Google, Facebook, Github, etc.).

Finally, you might notice that OAuth is listed as "As a service" and "In house". This is a specific note made to highlight the fact that there is actually a company called "OAuth" that implements the OAuth protocol… As a service. You can implement the OAuth protocol without using OAuth the company's service!

What is Session Based Authentication?

If we were to create a lineage for these authentication methods, session based authentication would be the oldest of them all, but certainly not obsolete. Session based authentication is at the root of the `passport-local` strategy. This method of authentication is "server-side", which means our Express application and database work together to keep the current authentication status of each user that visits our application.

To understand the basic tenets of session-based-authentication, you need to understand a few concepts:

- Basic HTTP Header Protocol
- What a cookie is
- What a session is

- How the session (server) and cookie (browser) interact to authenticate a user

HTTP Headers

There are many ways to make an HTTP request in a browser. An HTTP client could be a web application, IoT device, command line (curl), or a multitude of others. Each of these clients connect to the internet and make HTTP requests which either fetch data (GET), or modify data (POST, PUT, DELETE, etc.).

For explanation purposes, let's assume that:

Server = www.google.com
Client = random guy in a coffee shop working on a laptop

When that random person from the coffee shop types `www.google.com` into their Google Chrome browser, this request will be sent with "HTTP Headers". These HTTP Headers are key:value pairs that provide additional data to the browser to help complete the request. This request will have two types of headers:

1. General Headers
2. Request Headers

To make this interactive, open Google Chrome, open your developer tools (right click, "Inspect"), and click on the "Network" tab. Now, type `www.google.com` into your address bar, and watch as the Network tab loads several resources from the server. You should see several columns like Name, Status, Type, Initiator, Size, Time, and Waterfall. Find the request that has "document" as the "Type" value and click on it. You should see all the headers for this request and response interaction.

The request that you (as the client) made will have General and Request headers resembling (but not exact) the following:

```
General Headers
  Request URL: https://www.google.com/
  Request Method: GET
  Status Code: 200

Request Headers
  Accept: text/html
  Accept-Language: en-US
  Connection: keep-alive
```

When you typed `www.google.com` into your address bar and pressed enter, your HTTP request was sent with these headers (and probably a few others). Although these headers are relatively self-explanatory, I want to walk through a few to get a better idea of what HTTP Headers are used for. Feel free to look up any you don't know on MDN.

The `General` headers can be a mix of both request and response data. Clearly, the `Request URL` and `Request Method` are part of the request object and they tell the Google Chrome browser where to route your request. The `Status Code` is clearly part of the response because it indicates that your GET request was successful and the webpage at `www.google.com` loaded okay.

The `Request Headers` only contain headers included with the request object itself. You can think of request headers as "instructions for the server". In this case, my request tells the Google server the following:

- Hey Google Server, please send me HTML or text data. I'm either incapable or not interested in reading anything else right now!
- Hey Google Server, please only send me English words
- Hey Google Server, please don't close my connection with you after the request is over

There are many more request headers that you can set, but these are just a few common ones that you will probably see on all HTTP requests.

So when you searched for `www.google.com`, you sent your request and the headers to the Google Server (for simplicity, we will just assume it is one big server). The Google Server accepted your request, read through the "instructions" (headers), and created a *response*. The response was comprised of:

- HTML data (what you see in your Browser)
- HTTP Headers

As you might have guessed, the "Response Headers" were those set by the Google Server. Here are a few that you might see:

```
Response Headers
  Content-Length: 41485
  Content-Type: text/html; charset=UTF-8
  Set-Cookie: made_up_cookie_name=some value; expires=Thu, 28-Dec-
2020 20:44:50 GMT;
```

These response headers are fairly straightforward with the exception of the `Set-Cookie` header.

I included the `Set-Cookie` header because it is exactly what we need to understand in order to learn what Session-based Authentication is all about (and will help us understand other auth methods later in this post).

How Cookies Work

Without Cookies in the browser, we have a problem.

If we have a protected webpage that we want our users to login to access, without cookies, those users would have to login every time they refresh the page! That is because the HTTP protocol is by default "stateless".

Cookies introduce the concept of "persistent state" and allow the browser to "remember" something that the server told it previously.

The Google Server can tell my Google Chrome Browser to give me access to a protected page, but the second I refresh the page, my browser will "forget" this and make me authenticate again.

This is where Cookies come in, and explains what the `Set-Cookie` header is aiming to do. In the above request where we typed in `www.google.com` into our browser and pressed enter, our client sent a request with some headers, and the Google Server responded with a response and some headers. One of these response headers was `Set-Cookie: made_up_cookie_name=some value; expires=Thu, 28-Dec-2020 20:44:50 GMT;`. Here's how this interaction works:

Server: "Hey client! I want you to set a cookie called `made_up_cookie_name` and set it equal to `some value`.

Client: "Hey server, I will set this on the `Cookie` header of all my requests to this domain until Dec 28, 2020!"

We can verify that this actually happened in Google Chrome Developer Tools. Go to "Application"->"Storage" and click "Cookies". Now click on the site that you are currently visiting and you will see all the cookies that have been set for this site. In our made-up example, you might see something like:

Name	Value	Expires / Max-Age
made_up_cookie_name	some value	2020-12-28T20:44:50.674Z

pl-1.md hosted with ♥ by GitHub view raw

example cookie

This cookie will now be set to the `Cookie` **Request Header** on all requests made to `www.google.com` until the expiry date set on the cookie.

As you might conclude, this could be extremely useful for authentication if we set some sort of "auth" cookie. An overly simplified process of how this might work would be:

1. Random person from the coffee shop types `www.example-site.com/login/` into the browser
2. Random person from the coffee shop fills out a form on this page with a username and password
3. Random person's Google Chrome Browser submits a POST request with the login data (username, password) to the server running `www.example-site.com`.
4. The server running `www.example-site.com` receives the login info, checks the database for that login info, validates the login info, and if successful, creates a response that has the header `Set-Cookie: user_is_authenticated=true; expires=Thu, 1-Jan-2020 20:00:00 GMT`.
5. The random person's Google Chrome browser receives this response and sets a browser cookie:

Name	Value	Expires / Max-Age
user_is_authenticated	true	2020-12-28T20:44:50.674Z

pl-2.md hosted with ♥ by GitHub view raw

example cookie

6. The random person now visits `www.example-site.com/protected-route/`

7. The random person's browser creates an HTTP request with the header `Cookie: user_is_authenticated=true; expires=Thu, 1-Jan-2020 20:00:00 GMT` attached to the request.

8. The server receives this request, sees that there is a cookie on the request, "remembers" that it had authenticated this user just a few seconds ago, and allows the user to visit the page.

The Reality of this Situation

Obviously, what I have just described would be a highly insecure way to authenticate a user. In reality, the server would create some sort of hash from the password the user provided, and validate that hash with some crypto library on the server.

That said, the high-level concept is valid, and it allows us to understand the value of cookies when talking about authentication.

Keep this example in mind as we move through the remainder of this post.

Sessions

Sessions and cookies are actually quite similar and can get confused because they can actually be used *together* quite seamlessly. The *main difference* between the two is the **location** of their storage.

In other words, a Cookie is *set* by the server, but stored in the Browser. If the server wants to use this Cookie to store data about a user's "state", it would have to come up with an elaborate scheme to constantly keep track of what the cookie in the browser looks like. It might go something like this:

- Server: Hey browser, I just authenticated this user, so you should store a cookie to remind me (`Set-Cookie: user_auth=true; expires=Thu, 1-Jan-2020 20:00:00 GMT`) next time you request something from me
- Browser: Thanks, server! I will attach this cookie to my `Cookie` request header

- Browser: Hey server, can I see the contents at `www.domain.com/protected`? Here is the cookie you sent me on the last request.
- Server: Sure, I can do that. Here is the page data. I have also included another `Set-Cookie` header (`Set-Cookie: marketing_page_visit_count=1; user_ip=192.1.234.21`) because the company that owns me likes to track how many people have visited this specific page and from which computer for marketing purposes.
- Browser: Okay, I'll add that cookie to my `Cookie` request header
- Browser: Hey Server, can you send me the contents at `www.domain.com/protected/special-offer`? Here are all the cookies that you have set on me so far. (`Cookie: user_auth=true; expires=Thu, 1-Jan-2020 20:00:00 GMT; marketing_page_visit_count=1; user_ip=192.1.234.21`)

As you can see, the more pages the browser visits, the more cookies the Server sets, and the more cookies the Browser must attach in each request Header.

The Server might have some function that parses through all the cookies attached to a request and perform certain actions based on the presence or absence of a specific cookie. To me, this naturally begs the question… Why doesn't the server just keep a record of this information in a database and use a single "session ID" to identify events that a user is taking?

This is exactly what a session is for. As I mentioned, the main difference between a cookie and a session is *where* they are stored. A session is stored in some Data Store (a fancy term for a database) while a Cookie is stored in the Browser. Since the session is stored on the server, it can store sensitive information. Storing sensitive information in a cookie would be highly insecure.

Now where this all gets a bit confusing is when we talk about using cookies and session *together*.

Since Cookies are the method in which the client and server communicate metadata (among other HTTP Headers), a session must still utilize cookies. The easiest way to see this interaction is by actually building out a simple authentication application in Node + Express + MongoDB. I will assume that you have a basic understanding of building apps in Express, but I will try to explain each piece as we go.

Setup a basic app:

```
mkdir session-auth-app
cd session-auth-app
npm init -y
npm install --save express mongoose dotenv connect-mongo express-session passport passport-local
```

Here is `app.js`. Read through the comments to learn more about what is going on before continuing.

app.js

The first thing we need to do is understand how the `express-session` module is working within this application. This is a "middleware", which is a fancy way of saying that it is a function that modifies something in our application.

Quick Refresher on Express Middleware

Let's say we had the following code:

middleware.js

As you can see, this is an extremely simple Express application that defines two middlewares and has a single route that you can visit in your browser at `http://localhost:3000`. If you started this application and visited that route, it would say "Custom Property Value: undefined" because defining middleware functions alone is not enough.

We need to tell the Express application to actually use these middlewares. We can do this in a few ways. First, we can do it within a route.

```
app.get('/', myMiddleware1, (req, res, next) => {
    res.send(`<h1>Custom Property Value: ${req.newProperty}`);
});
```

If you add the first middleware function as an argument to the route, you will now see "Custom Property Value: my custom property" show up in the browser. What really

happened here:

1. The application was initialized
2. A user visited `http://localhost:3000/` in the browser, which triggered the `app.get()` function.
3. The Express application first checked to see if there was any "global" middleware installed on the router, but it didn't find any.
4. The Express application looked at the `app.get()` function and noticed that there was a middleware function installed before the callback. The application ran the middleware and passed the middleware the `req` object, `res` object, and the `next()` callback.
5. The `myMiddleware1` middleware first set `req.newProperty`, and then called `next()`, which tells the Express application "Go to the next middleware". If the middleware did not call `next()`, the browser would get "stuck" and not return anything.
6. The Express app did not see any more middleware, so it continued with the request and sent the result.

This is just one way to use middleware, and it is exactly how the `passport.authenticate()` function (more on this later, so keep in mind) works.

Another way we can use middleware is by setting it "globally". Take a look at our app after this change:

middleware.js

With this app structure, you will notice that visiting `http://localhost:3000/` in the browser *still* returns the same value as before. This is because the `app.use(myMiddleware2)` middleware is happening *before* the `app.get('/', myMiddleware1)`. If we removed the middleware from the route, you will see the updated value in the browser.

```
app.use(myMiddleware2);
```

```
app.get('/', (req, res, next) => {
    // Sends "Custom Property Value: updated value
    res.send(`<h1>Custom Property Value: ${req.newProperty}`);
});
```

We could also get this result by placing the second middleware after the first within the route.

```
app.get('/', myMiddleware1, myMiddleware2, (req, res, next) => {
    // Sends "Custom Property Value: updated value
    res.send(`<h1>Custom Property Value: ${req.newProperty}`);
});
```

Although this is a quick and high-level overview of middleware in Express, it will help us understand what is going on with the `express-session` middleware.

How Express Session Middleware works

As I mentioned before, the `express-session` module gives us middleware that we can use in our application. The middleware is defined in this line:

```
// Again, here is the documentation for this -
https://www.npmjs.com/package/express-session

app.use(session({
    secret: process.env.SECRET,
    resave: false,
    saveUninitialized: true,
    store: sessionStore
}));
```

Here is a brief overview of what the Express Session Middleware is doing:

1. When a route is loaded, the middleware checks to see if there is a session established in the Session Store (MongoDB database in our case since we are using the `connect-mongo` custom Session Store).

2. If there is a session, the middleware validates it cryptographically and then tells the Browser whether the session is valid or not. If it is valid, the Browser automatically attaches the `connect.sid` Cookie to the HTTP request.

3. If there is no session, the middleware creates a new session, takes a cryptographic hash of the session, and stores that value in a Cookie called `connect.sid`. It then attaches the `Set-Cookie` HTTP header to the `res` object with the hashed value (`Set-Cookie: connect.sid=hashed value`).

You might be wondering why this is useful at all, and how all this actually works.

If you remember from the quick refresher on Express Middlewares, I said that a middleware has the ability to alter the `req` and `res` objects that are passed from one middleware to the next until it reaches the end of the HTTP request. Just like we set a custom property on the `req` object, we could also set something much more complex like a `session` object that has properties, methods, etc.

That is exactly what the `express-session` middleware does. When a new session is created, the following properties are added to the `req` object:

- `req.sessionID` - A randomly generated UUID. You can define a custom function to generate this ID by setting the `genid` option. If you do not set this option, the default is to use the `uid-safe` module.

```
app.use(session({
  genid: function (req) {
    // Put your UUID implementation here
  }
}));
```

- `req.session` - The Session object. This contains information about the session and is available for setting custom properties to use. For example, maybe you want to track how many times a particular page is loaded in a single session:

```
app.get('/tracking-route', (req, res, next) => {

  if (req.session.viewCount) {
    req.session.viewCount = req.session.viewCount + 1;
  } else {
    req.session.viewCount = 1;
  }

  res.send("<p>View count is: " + req.session.viewCount + "</p>");
```

```
});
```

- `req.session.cookie` - The Cookie object. This defines the behavior of the cookie that stores the hashed session ID in the browser. Remember, once the cookie has been set, the browser will attach it to every HTTP request automatically until it expires.

How Passport JS Local Strategy works

There is one last thing that we need to learn in order to fully understand Session-Based Authentication–Passport JS.

Passport JS has over 500 authentication "Strategies" that can be used within a Node/Express app. Many of these strategies are highly specific (i.e. `passport-amazon` allows you to authenticate into your app via Amazon credentials), but they all work similar within your Express app.

In my opinion, the Passport module could use some work in the department of documentation. Not only does Passport consist of two modules (Passport base + Specific Strategy), but it is also a middleware, which as we saw is a bit confusing in its own right. To add to the confusion, the strategy that we are going to walk through (`passport-local`) is a middleware that modifies an object created by another middleware (`express-session`). Since the Passport documentation has little to say around how this all works, I will attempt to explain it to the best of my ability in this post.

Let's first walk through the setup of the module.

If you have been following along with this tutorial, you already have the modules needed. If not, you will need to install Passport and a Strategy to your project.

```
npm install --save passport passport-local
```

Once you have done that, you will need to implement Passport within your application. Below, I have added all the pieces you need for the `passport-local` strategy. I have removed comments to simplify. Take a quick read through the code and then we will walk through all of the `// NEW` code.

app.js

Yes, I know there is a lot to take in here. Let's start with the easy parts–the helper functions. In the code above, I have two helper functions that will assist in creating and validating a password.

passwordUtils.js

In addition to the comments, I'll note that these functions require the NodeJS built-in `crypto` library. Some would argue a better crypto library, but unless your application requires a high degree of security, this library is plenty sufficient!

Next up, let's take a look at the `passport.use()` method.

Passport JS Configuration

I know the above function is quite a lot to look at, so let's explore some of its key components. First, I'll mention that with **all** Passport JS authentication strategies (not just the local strategy we are using), you will need to supply it with a callback that will be executed when you call the `passport.authenticate()` method. For example, you might have a login route in your app:

```
app.post('/login', passport.authenticate('local', { failureRedirect:
'/login' }), (err, req, res, next) => {
    if (err) next(err);
    console.log('You are logged in!');
});
```

Your user will type in their username and password via a login form, which will create an HTTP POST request to the `/login` route. Let's say your post request contained the following data:

```
{
  "email": "sample@email.com",
  "pw": "sample password"
}
```

This WILL NOT WORK. The reason? Because the `passport.use()` method *expects* your POST request to have the following fields:

```
{
  "username": "sample@email.com",
  "password": "sample password"
}
```

It looks for `username` and `password` field. If you wanted the first json request body to work, you would need to supply the `passport.use()` function with field definitions:

```
passport.use(

  {
    usernameField: 'email',
    passwordField: 'pw'
  },
  function (email, password, callback) {

    // Implement your callback function here

  }

);
```

By defining the `usernameField` and `passwordField`, you can specify a custom POST request body object.

That aside, let's return to the POST request at the `/login` route:

```
app.post('/login', passport.authenticate('local', { failureRedirect:
'/login' }), (err, req, res, next) => {
    if (err) next(err);
    console.log('You are logged in!');
});
```

When the user submits his/her login credentials, the `passport.authenticate()` method (used as middleware here) will execute the callback that you have defined and supply it with the `username` and `password` from the POST request body. The

`passport.authenticate()` method takes two parameters–the name of the strategy, and options. The default strategy name here is `local`, but you could change this like so:

```
// Supply a name string as the first argument to the passport.use()
function

passport.use('custom-name', new Strategy());

// Use the same name as above
app.post('/login', passport.authenticate('custom-name', {
failureRedirect: '/login' }), (err, req, res, next) => {
    if (err) next(err);
    console.log('You are logged in!');
});
```

The way I have used the `passport.authenticate()` strategy will first execute the callback function that we defined within `new LocalStrategy()`, and if the authentication is successful, it will call the `next()` function, and we will enter the route. If authentication was not successful (invalid username or password), the app will redirect to the `/login` route again.

Now that we understand how it is used, let's return to the callback function that we defined earlier and that `passport.authenticate()` is using.

Passport Configuration

I have commented the above in great detail, so be sure to read through before moving on.

As you may notice, the callback function is database agnostic and validation agnostic. In other words, we don't need to use MongoDB nor do we need to validate our passwords in the same way. PassportJS leaves this up to us! This can be confusing, but is also extremely powerful and is why PassportJS has such widespread adoption.

Next, you'll see two related functions:

```
passport.serializeUser(function(user, cb) {
    cb(null, user.id);
});
```

```
passport.deserializeUser(function(id, cb) {
    User.findById(id, function (err, user) {
        if (err) { return cb(err); }
        cb(null, user);
    });
});
```

Personally, I found these two functions to be the most confusing because there is not a lot of documentation around them. We will further explore what these functions are doing when we talk about how PassportJS and Express Session middleware interact, but in short, these two functions are responsible for "serializing" and "deserializing" users to and from the current session object.

Instead of storing the entire `user` object in the session, we only need to store the database ID for the user. When we need to get more information about the user in the current session, we can use the deserialize function to look the user up in the database using the ID that was stored in the session. Again, we will make more sense of this soon.

Finally, with the Passport implementation, you will see two more lines of code:

```
app.use(passport.initialize());
app.use(passport.session());
```

If you remember from earlier in the post on how middleware works, by calling `app.use()`, we are telling Express to execute the functions within the parentheses **in order** on **every request**.

In other words, for every HTTP request our Express app makes, it will execute `passport.initialize()` and `passport.session()`.

Something seem weird here??

If `app.use()` **executes** the function contained within, then the above syntax is like saying:

```
passport.initialize()();
passport.session()();
```

The reason this works is because these two functions actually return another function! Kind of like this:

```
Passport.prototype.initialize = function () {
  // Does something

  return function () {
    // This is what is called by `app.use()`
  }
}
```

This is not necessary to know to use Passport, but definitely clears up some confusion if you were wondering about that syntax.

Anyways…

These two middleware functions are necessary for integrating PassportJS with `express-session` middleware. That is why these two functions **must come AFTER** the `app.use(session({}))` middleware! Just like `passport.serializeUser()` and `passport.deserializeUser()`, these middlewares will make much more sense shortly.

Conceptual Overview of Session Based Authentication

Now that we understand HTTP Headers, Cookies, Middleware, Express Session middleware, and Passport JS middleware, it is finally time to learn how to use these to authenticate users into our application. I want to first use this section to review and explain the conceptual flow, and then dive into the implementation in the next section.

Here is a basic flow of our app:

1. Express app starts and listens on `http://www.expressapp.com` (just assume this is true for the sake of the example).
2. A user visits `http://www.expressapp.com/login` in the browser
3. The `express-session` middleware realizes that there is a user connecting to the Express server. It checks the `Cookie` HTTP header on the `req` object. Since this user is visiting for the first time, there is no value in the `Cookie` header. Because there is no `Cookie` value, the Express server returns the `/login` HTML and calls the `Set-Cookie` HTTP header. The `Set-Cookie` value is the cookie string generated by

`express-session` middleware according to the options set by the developer (assume in this case the maxAge value is 10 days).

4. The user realizes that he doesn't want to login right now, but instead, wants to go for a walk. He closes his browser.

5. The user returns from his walk, opens the browser, and returns to `http://www.expressapp.com/login` again.

6. Again, the `express-session` middleware runs on the GET request, checks the `Cookie` HTTP header, but this time, finds a value! This is because the user had previously created a session earlier that day. Since the `maxAge` option was set to 10 days on the `express-session` middleware, closing the browser does not destroy the cookie.

7. The `express-session` middleware now takes the `connect.sid` value from the `Cookie` HTTP header, looks it up in the `MongoStore` (fancy way to say that it looks up the id in the database in the `sessions` collection), and finds it. Since the session exists, the `express-session` middleware does not do anything, and both the `Cookie` HTTP header value and the `MongoStore` database entry in the `sessions` collection stays the same.

8. Now, the user types in his username and password and presses the "Login" button.

9. By pressing the "Login" button, the user sends a POST request to the `/login` route, which uses the `passport.authenticate()` middleware.

10. On every request so far, the `passport.initialize()` and `passport.session()` middlewares have been running. On each request, these middlewares are checking the `req.session` object (created by the `express-session` middleware) for a property called `passport.user` (i.e. `req.session.passport.user`). Since the `passport.authenticate()` method had not been called yet, the `req.session` object did not have a `passport` property. Now that the `passport.authenticate()` method has been called via the POST request to `/login`, Passport will execute our user-defined authentication callback using the username and password our user typed in and submitted.

11. We will assume that the user was already registered in the database and typed in the correct credentials. The Passport callback validates the user successfully.

12. The `passport.authenticate()` method now returns the `user` object that was validated. In addition, it attaches the `req.session.passport` property to the `req.session` object, serializes the user via `passport.serializeUser()`, and attaches the serialized user (i.e. the ID of the user) to the `req.session.passport.user` property. Finally, it attaches the full user object to `req.user`.

13. The user turns off his computer and goes for another walk because our application is boring.

14. The user turns on his computer the next day and visits a **protected route** on our application.

15. The `express-session` middleware checks the `Cookie` HTTP header on `req`, finds the session from yesterday (still valid since our `maxAge` was set to 10 days), looks it up in `MongoStore`, finds it, and does nothing to the `Cookie` since the session is still valid. The middleware re-initializes the `req.session` object and sets to the value returned from `MongoStore`.

16. The `passport.initialize()` middleware checks the `req.session.passport` property and sees that there is still a `user` value there. The `passport.session()` middleware uses the `user` property found on `req.session.passport.user` to re-initialize the `req.user` object to equal the user attached to the session via the `passport.deserializeUser()` function.

17. The protected route looks to see if `req.session.passport.user` exists. Since the Passport middleware just re-initialized it, it does, and the protected route allows the user access.

18. The user leaves his computer for 2 months.

19. The user comes back and visits the same protected route (hint: the session has expired!)

20. The `express-session` middleware runs, realizes that the value of the `Cookie` HTTP header has an **expired** cookie value, and replaces the `Cookie` value with a new Session via the `Set-Cookie` HTTP header attached to the `res` object.

21. The `passport.initialize()` and `passport.session()` middlewares run, but this time, since `express-session` middleware had to create a new session, there is no longer a `req.session.passport` object!

22. Since the user did not log in and is trying to access a protected route, the route will check if `req.session.passport.user` exists. Since it doesn't, access is denied!

23. Once the user logs in again and triggers the `passport.authenticate()` middleware, the `req.session.passport` object will be re-established, and the user will again be able to visit protected routes.

Phewwww…

Got all that?

Session Based Authentication Implementation

The hard part is over.

Putting everything together, below is your full functional Session Based authentication Express app. Below is the app contained within a single file, but I have also refactored this application closer to what you would use in the real world in this repository.

app.js

Review and Preview

That is the end of our `passport-local` authentication tutorial, but I have a closely related tutorial on the `passport-jwt` strategy that will help you gain a widespread understanding of different user authentication flows.

As we transition from talking about session-based authentication to JWT based authentication, it is important to keep our authentication flows clear. To do a quick review, the basic auth flow of a session-based authentication app is like so:

1. User visits your Express application and signs in using his username and password

2. The username and password are sent via POST request to the `/login` route on the Express application server

3. The Express application server will retrieve the user from the database (a hash and salt are stored on the user profile), take a hash of the password that the user provided a few seconds ago using the salt attached to the database user object, and verify that the hash taken matches the hash stored on the database user object.

4. If the hashes match, we conclude that the user provided the correct credentials, and our `passport-local` middleware will attach the user to the current session.

5. For every new request that the user makes on the front-end, their session Cookie will be attached to the request, which will be subsequently verified by the Passport middleware. If the Passport middleware verifies the session cookie successfully, the server will return the requested route data, and our authentication flow is complete.

What I want you to notice about this flow is the fact that the user only had to type in his username and password **one time**, and for the remainder of the session, he can visit protected routes. The session cookie is **automatically** attached to all of his requests because this is the default behavior of a web browser and how cookies work! In addition, each time a request is made, the Passport middleware and Express Session middleware will be making a query to our database to retrieve session information. In other words, **to authenticate a user, a database is required**.

Now skipping forward, you'll begin to notice that with JWTs, there is absolutely no database required on **each request** to authenticate users. Yes, we will need to make one database request to initially authenticate a user and generate a JWT, but after that, the JWT will be attached in the `Authorization` HTTP header (as opposed to `Cookie` header), and no database is required.

If this doesn't make sense, that is okay. Go to my post on the `passport-jwt` strategy by following this link.

JavaScript Passportjs Expressjs Nodejs User Authentication