Lecture 10: Middleware and Authentication

Middleware

What is middleware?

A middleware is a function that has access to the request and response objects

These functions can:

- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware function in the stack.

You can apply middleware to the entire application, or portions of the application

- You can apply it to a portion of the application by supplying a path as the first parameter to the middleware function

Practical Uses for Middleware

Middlewares are useful for a number of reasons, and have many common uses

- Logging requests
- Authentication
- Access control
- Caching data
- Serialization

Writing a middleware

Writing a middleware is extremely easy

- Register your middleware, optionally providing a path to apply that middleware to
- Have your middleware perform a task and when done:
  - Have your middleware end the response
  - Have your middleware call the next middleware

As an example, see the *server.js* file, which has several middleware:

- One which will count the number of requests made to your website
- One which will count the number of requests that have been made to the current path
- One which will log the last time the user has made a request, and store it in a cookie.
- One which will deny all users access to the */admin* path.

Using cookies

What is a cookie?

An HTTP Cookie is a small piece of data that is shared between the server and the client.

- Can be read or set in client or server
- Ultimately, sent back and forth as string data

Cookies are sent through headers.

HTTP Cookies cannot be deleted, but can be expired

- After their expiration date, they will automatically be removed

Cookies will be sent back to the server on every request automatically; only new or updated cookies will be sent in a response.

Cookies are a browser concept, and they are rarely passed back and forth when you are writing APIs or requesting resources programmatically.

Installing the cookie-parser package

We will be using the cookie-parser middleware in order to easily handle our cookies as an object.

- The other case is to manually parse headers and parse objects as well as we can, which can get redundant.

```
npm install cookie-parser --save
```

We then apply the cookie-parser as a middleware, without a route path so that it applies to the whole application.

Using cookies

CLIENT SIDE

You can set by setting *document.cookies* = *“key=value”*

Even though you are re-assigning, it will simply add it to your list of cookies.

You can get a list of all your cookies and their values using the *document.cookies* and parsing it to find the cookie of your choice.

Deleting cookies requires that you set the cookie with an expiration; ie: *document.cookies=“key=value; expires=Thu, 01 Jan 1970 00:00:00 UTC";*

SERVER SIDE

You can set cookies by calling the *response.cookies(name, value, options)* function.

You can get cookies by referencing the *request.cookies* object, which will have cookies keyed by name.

Deleting cookies requires you to expire the cookies, which we can do by setting the cookie with the *expires* option set to any time in the past, then calling *response.clearCookie(name)*

Example

You can see examples of cookies being manipulated in the *server.js* file, specifically the middleware that keeps track of the last time the user accessed the website. This will be passed back and forth to the user.

Authentication using Express, Middleware, and MongoDB

What is authentication?

Authentication is the act of confirming the identity of a person, group, or entity.

In web technology, this often means creating a **user login system**

- You will use a combination of data in order to identify a user.

There are many other forms of authentication in web technology:

- You can make an authentication system that allows you to limit API Access
    - Force users to have a token
    - Allow users a certain number of access hits a month
- You can selectively allow or dis-allow access to resources based on user login state

Implementing Authentication

In order to implement authentication and create a user login system, we will be breaking down the task into several steps:

- Creating and storing users
- Allowing users to login via a form
- Storing session data in a cookie
- Validating the data stored in the cookie
- Storing the user as part of the request object.

Let’s walk through this process.

Creating and Storing Users

The first step of authentication is very, very easy; you have to create, and store users.

There are some things you'll be storing, and some things you'll be storing in a very specific way

- First off, you will **never** store a plaintext password. You will be using the **bcrypt-nodejs** package in order to create a hash of the password
- For the sake of authentication, you're going to be adding an array for users that will keep track of multiple session identifiers. These session identifiers will allow you to keep track of logged in browser sessions

You will need to create a form to allow users to signup, where you will need to check for:

- Duplicate username / emails / other non-duplicatable data
- Existence of passwords

Allowing users to login via a form

This step is extremely easy!

You will need to provide users with some way to actually perform a login. You will need to setup a form that allows users to POST their username and password to a route.

This route will need to validate the username and password provided against entries in the database.

- You will retrieve the user with that matching username
- You will use **bcrypt** to compare if their **supplied password** is a match to their **hashed password**

I have created a simple file, **bcrypt_example.js** to demonstrate how to use bcrypt to create and compare hashes.

If there is a match, you can proceed; if not you will simply allow the request to continue.

Storing session data in a cookie

If the user logged in with proper credentials, you will then create a session id!

This session id should be some sort of **very** long identifier, such as a Guid.

Rather than storing the user id or username, and password in cookies, we instead are opting to store a session id so that the username or password cannot be intercepted.

This session id will be passed to the user via-cookie and will also be stored as one of many session ids on the user in the database.

Validating the data stored in the cookie

It is now time to make your middleware!

Your middleware should run on each request, and will check for a cookie containing a session ID

If it contains a session id, you will check the database for a single user that has that session id stored in their session id field

- If there is a match, you've found your user!
- If not, your request is coming from an unauthenticated source; expire their cookie.

If not, your request is coming from an unauthenticated source.

Storing the user in the request object

In your middleware, you have access to the request and the response objects, and you can add properties to them easily.

If you are able to associate a session id with a user, you may define a property on the *request* (or response!) object that stores the user, or some representation of them (ie: just storing the user id).

The data you store will be accessible:

- In middleware that are defined after the authentication middleware
- In your routes

If you define middleware after your authentication middleware, you can attach them to particular paths (such as **/user**) and, if a user is not logged in, you can redirect them. You can also do things such as check on other paths (ie: **/admin**) to see if the user has permission to access those paths, and redirect if not.

Logging Out

Logging out is extremely easy, and only has two steps!

- After hitting a logout route, you will expire the cookie for the session id
- You will remove the session id from the user’s session id list
- You will invalidate any other cookies that are relevant to the user.

By doing both of those, you will have successfully invalidated the session and the user will no longer be authenticated.

Passport

What is Passport?

Passport is an authentication middleware for Node.js that works by allowing you to plug different authentication strategies into your server, and having Passport handle access control.

- http://passportjs.org/

Passport requires a *strategy* in order to determine how to associate a particular request to a particular user.

Passport has 3 components to it

- Authentication strategies
- Application middleware
- Sessions (*optional*)

What is a strategy?

A strategy is a way for Passport to associate a request to a particular user. There are many different strategies that you can use:

Strategy	**Description**
passport-facebook	Authenticates via Facebook
passport-http-bearer	Authenticates via the http-bearer strategy.
passport-local	Authenticates via username and password
passport-token	Authenticates via tokens passed on each request; useful for APIS
passport-jwt	Authenticates via JWT Token support

Basic authentication with Passport

We can use Passport to create a basic username and login system setup using the passport-local module. A demonstration can be found in the express-passport example by Passport's creators

- https://github.com/jaredhanson/passport-local
- https://github.com/passport/express-4.x-local-example

To explain the important parts:

- We require Passport, and set passport to use a new Strategy for local users. This is how we tell passport how to take a username and password and turn it into a user.
- We explain to Passport how to serialize the user to be stored in the session; this is what gets stored in the user's cookie.
- We setup a way to deserialize the user, which is going from the information stored in the cookie to the user in the database.
- We add the middlewares for cookie-parser, express-session, passport.initialize, and passport.session
- On routes you want to protect, you apply the passport.authenticate middleware to have the route only load if the criteria is met for authentication.