Introduction to Computer Science I
COMP 2406 – Fall 2019

Authentication, Authorization, and Sessions

Dave McKenney
david.mckenney@carleton.ca

Learning Outcomes

by the End of this Lecture, Students that have Completed the Reading Assignment and Review Questions should be Able to:

Understand authentication/authorization

Understand what cookies are and how they are used

Implement sessions in an Express app

Identify differences between query parameter-based and cookie-based sessions

Before we start…

An important note – we will continue to use HTTP for now

This is important to recognize, as these transactions will not be encrypted

Without the encryption, these examples are insecure. We will discuss HTTPS next lecture.

Authentication vs. Authorization

While often interlinked in a way, authentication and authorization are separate goals…

Authentication vs. Authorization

Authentication involves proving that you are who you say you are. Often using a username/password.

Examples: login forms, HTTP authentication

Authentication vs. Authorization

Authorization involves deciding if you have permission to access/modify a resource

Achieved by specifying access controls to URLs

In our server, we can check the 'permissions' of a user and decide if they can do something or not

Sessions

Session data, in the context of a web application, is data related to one particular clients interactions with the app

A way of adding statefullness into the stateless HTTP protocol

Often used for user settings, authorization, etc.

Authorization in Express

Express allows us to easily add the idea of authorization to our apps

We can create an authorization middleware that is responsible for deciding if a user is authorized or not

Authorization in Express

```
function auth(req, res, next){
        //Check req info, load user info, etc.

        if (user.auth){ //Check if they are authorized
                next( );
        }else{
                res.status(401).send("Unauthorized");
        }
}
```

Note: user.auth could be a more complex check including the URL being accessed

Authorization in Express

We can then register this middleware so it is executed for requests that require authentication

Authorization in Express

For example, to add it to all requests:

```
app.all("/*", auth);
app.get("/users", getUsers);
app.post("/users", newUser);
```

Note that /* matches all request URLs and .all applies to all HTTP methods

Alternatively, this is a good time for app.use:

```
app.use("/", auth);
app.get("/users", getUsers);
app.post("/users", newUser);
```

Any route that starts with / (which is all of them) will call auth middleware function first

Authorization in Express

If we don't want ALL routes to require authorization, we can add the auth middleware to any that need it:

```
app.get("/", homepage);
app.get("/products", getProducts);
app.post("/products", auth, newProduct);
app.put("/products/:pid", auth, updateProd);
```

Authorization not required for home page and GET /products. Required for POST/PUT.

Authenticating in Express

There are many ways to perform authentication and authorization

A basic way would be to have the user include their username and password with the request

The server can then check the validity of the data and decide if the request is allowed or not

Authenticating in Express

This is how basic HTTP authentication works

The server can include the following header in a response with status 401 (unauthorized):

WWW-Authenticate: Basic realm=<realm>

Where <realm> is a description of the area the client is trying to access

Authenticating in Express

This will cause the login dialog to show up on the requesting clients browser

The client can enter their credentials and submit them to the server

The server can verify the credentials and decide how to respond

Authenticating in Express

The browser will automatically include their username/password in the Authorization HTTP header for future requests

The browser automatically handles this by storing the username/password locally

See 19-ex1-basic-auth.js

There are a number of downsides to this approach:

1. **Username and password are sent with every request**
2. **There is no way to log out**
3. **There is no easy way to 'expire' the credentials**

Authenticating in Express

Instead, a commonly used solution is to generate unique 'session IDs' for users when they log in

The server provides this session ID to the client when they log in, and remembers it locally (e.g., in a database or in RAM)

The client includes that session ID when they make requests

Authenticating in Express

The server can look up the provided session ID, see what user it belongs to (if any), and decide what permissions that user has

The session ID is typically only valid for a specific amount of time

This approach has several benefits over including the username/password for each request…

Authenticating in Express

1. **Session IDs are sent with requests instead of usernames and passwords**

If an attacker gains access to the session ID, they can act as that client only until the session expires

The username/password are sent only once to initialize the session

2. We can provide a 'log out' operation that invalidates the session ID

This can be executed when the client makes a request to logout, or if the server detects/suspects the client's session has been compromised

3. The session ID can easily be set to expire after a certain time or when the browser closes

So the session will become invalid automatically after a relatively short period of time

Authentication/Authorization Methods

We will discuss two ways of using session IDs to authorize a user's request:

1. **Using query parameters**
2. **Using cookies**

We will also discuss some of the benefits/dangers

Query Parameter Session ID

The query parameter approach is generally used with non-browser-based apps

For example, JSON APIs that do not present a front-end to interact with

Query Parameter Session ID

To get a session ID in the first place, the client must 'log in' to the app

This can be done by making a request that contains the client's authentication information (e.g., username and password)

Query Parameter Session ID

The server then can:

1. **Validate the username and password**
2. **If they are valid, create a session ID and associate it with the user**
3. **Store the session information (e.g., in database)**
4. **Reply with the session ID so the client knows what it is**

Query Parameter Session ID

The user can then include a query parameter to indicate its session ID for requests it makes

e.g., GET /questions?session_id=183ajf8ek1k34o1O

Query Parameter Session ID

The server can then check:

1. **If the session ID is included**
2. **If the session ID is valid (i.e., belongs to a user)**
3. **If the user is authorized to execute the request**

The server can then handle the request or respond with a 401 status

Query Parameter Session ID

The session ID uniquely identifies one client's session

So the server can also store any information that needs to be remembered about that session (this is similar to the idea from assignment #3)

See 19-ex2-token-session.js for an example

Advantages and Criticisms

We do not need to send the username/password every time with this approach

Also, we can invalidate a session ID by removing it from the database

We could also include an expiry time in the database for the session

Advantages and Criticisms

There are still some drawbacks though…

The session ID is included in the URL for each request

If we are using HTTPS, the URL will be encrypted while it is transferred across the network

Advantages and Criticisms

But the URL may still show up in:

1. **Server logs**
2. **Browser history**

For these reasons, this solution is best used when session IDs have a short lifespan.

Cookie-based Sessions

If the client will be using a browser, a cookie-based solution is likely a better fit

Many HTTP modules/tools also support cookies

Cookie-based Sessions

What is a cookie?

A small piece of data sent by a server to a client and stored by the client's browser

Cookies are sent back to a server, with their data, when requests are made by the client

Cookie-based Sessions

A cookie consists of the following components:

Name

Value

0+ attributes (expiration time, domain, etc.)

Common cookie properties:

Expires – the date the cookie expires (e.g., Wed, 21 Oct 2015 07:28:00 GMT)

Max-Age – the number of seconds the cookie should be valid (e.g., 300 → 5 minutes)

Without an expiry, the cookie is a 'session cookie' and is deleted when the browser closes

Common cookie properties:

Secure – cookie can only be sent over HTTPS

HttpOnly – cookie can only be included by browser in HTTP requests, cannot be accessed by Javascript

Without HttpOnly, Javascript can read cookies, which is a security issue (see XSS attack, etc.)

Cookie-based Sessions

Common cookie properties:

Domain - hosts allowed to receive cookie, if not specified, defaults to current host

Path - URL path that must be contained in requested URL to include cookie

Cookie-based Sessions

How are cookies generally used?

The server 'sets' a cookie (or cookies) when a request is made

The name, value, and attributes of a cookie are included in the Set-Cookie response header

Cookie-based Sessions

The client browser stores the cookie data locally

On future requests to the same server/domain, the browser includes all of the cookies that server has set

This way, information can be 'remembered' between requests

Cookie-based Sessions

Since cookies are sent along with each request, cookie data should generally be small

Current cookie specs require browsers to support:

1. **Cookies up to 4KB**
2. **At least 50 cookies per domain**
3. **At least 3000 cookies in total**

Cookie-based Sessions

How cookies are used for session management:

1. **Client makes an original request**
2. **Server creates session ID, sends cookie to client with that ID**
3. **On further requests, client includes that cookie, so server can look up that session ID**
4. **Server can store data associated with the ID to remember information between requests**

Cookie-based Sessions

An example HTTP response setting a cookie:

HTTP/1.1 200 OK

Content-type: text/html

Set-Cookie: theme=light

Set-Cookie: sessionToken=abc123;
Expires=Wed, 09 Jun 2021 10:18:14 GMT

...

Cookie-based Sessions

An example HTTP request including a cookie:

```
GET /somePage.html HTTP/1.1
Host: www.example.com
Cookie: theme=light; sessionToken=abc123
...
```

Tracking Cookies

Cookies are also used for tracking…

When you go to example.com, the example.com server(s) may set a cookie on your machine

This is an example of a first-party cookie (it came from the server you directly visited)

Tracking Cookies

But, web sites often have content from OTHER domains as well

For example, images and advertisements

Tracking Cookies

When the page you loaded from example.com is processed by your browser, additional requests are made for this other content

Since you are requesting content from these servers, they can also set cookies on your machine

These are called third-party cookies

Tracking Cookies

When you make future requests to those same servers/domains, your browser includes the same cookies

Old cookie standards did not allow third-party cookies

More recent standards have allowed the user agent to decide (three guesses why?)

Tracking Cookies

So when you go to example.com, it loads an advertisement from ads.com

The request to ads.com sets a cookie in your browser (e.g., id=someUniqueID)

Tracking Cookies

Next you go to example2.com, which loads an advertisement from ads.com

The cookie is sent to ads.com, often containing a 'referral URL' listing the page the request originated from (example2.com)

Now ads.com knows you were at example.com and example2.com

Tracking Cookies

This is a simple example, but processing these cookies can produce a LOT of information about your browsing history

(e.g., what URLs you visited, what your query parameters are, what times you visited, etc.)

Cookie-Based Sessions in Express

There are a few modules to facilitate cookie-based sessions in Node.js:

cookie-parser

express-session

connect-mongodb-session

cookie-parser Module

The cookie-parser module is used to automatically parse cookie data from requests:

```
const express = require('express')
const cookieParser = require('cookie-parser')
const app = express()
app.use(cookieParser())
```

Each request object now will have req.cookies

express-session Module

The express-session module provides session handling middleware

Creates a req.session property that can be used to store session data

Used to rely on cookie-parser, but is not necessary anymore

express-session Module

Note that all session data is stored on the server (more on this later)

Only the session ID (managed by the module) is included in the cookie

This means we have much less to keep track of

express-session Module

```
const express = require('express')
const session = require('express-session')
const app = express()
app.use(session({options}))
```

express-session Module

Useful options for session middleware:

cookie – an object including options for the session cookie that is sent to the client

Can set max age, expiration, path/domain, Secure, HttpOnly, etc.

Defaults to: { path: '/', httpOnly: true, secure: false, maxAge: null }

Useful options for session middleware:

rolling – if true, expiration is reset on each request

So cookie will expire in Max-Age time after the most recent request

Defaults to false

Useful options for session middleware:

secret – A required option specifying the key to use for signing cookies

Signing cookies is a way for the server to ensure that the cookie has not been tampered with

express-session Module

Useful options for session middleware:

store – the session storage instance

This defaults to a memory-based store, which is only meant for development purposes

More on using MongoDB as a store later...

express-session Module

The express-session module allows us to easily set up simple sessions

For example, see the user-specific view count page example in 19-ex3-express-session.js

express-session Module

We can also implement a login/logout feature as in the token-based example

This is more straightforward and does not require the user to remember/specify their session ID

See 19-ex4-express-session-login.js

Storing Sessions in MongoDB

As mentioned previously, the default store mechanism of the Express session module is only meant for development

In order to scale the solution and provide persistence, we need another store

Storing Sessions in MongoDB

MongoDB can be used as a store for the session data

There is a connect-mongodb-session module that is designed to easily allow this functionality

```
npm install connect-mongodb-session
```

Storing Sessions in MongoDB

To use connect-mongodb-session:

1. **Require the module**
2. **Create a new instance of MongoDBStore with the proper connection information**
3. **Set the MongoDBStore instance as the value for the store property in the express sessions options**

See 19-ex5-mongo-session-login.js

Sessions in our Store

The store app we have been looking at could make use of sessions as well

The new code provides the same type of login/logout behaviour from the last few examples

Sessions in our Store

Once a user is logged in, our server can decide what information to make available

For example, we could only show the personal information of a user (address, purchase history), if that person is logged in and viewing their own profile

Sessions in our Store

We could also create account types and only allow certain types to perform some actions

For example, only 'admin' types should be able to POST to /products to create a new product

Summary of Sessions

So sessions give us a way of remembering information about the current user

We can authenticate a user with a user/password

We can authorize that user's actions by checking if they have permission to execute the action

Summary of Sessions

Our current solution is still working over HTTP, without any encryption

This is not a secure way of doing things

Summary of Sessions

Next we will talk about HTTPS, encrypting web traffic, and look at how we could add HTTPS support into our own servers

Questions?