Seneca College

Web Programming

WEB322 – Week 10 – Class 1

WEB322 Ho

Managing State Information

Up to this point in the course, we have been building simple applications as examples of how to use the basic building blocks concepts of a website. However, one of the core concepts that almost every serious website will use has been missing. This is concept of **state**. One of the most common uses for state is having users and logins on a website. Adding a login and user acc a website is not that difficult and it will enable you to secure certain routes and make sure information is only available to be by the correct audience.

There are 3 core pieces required in your applications to be able to deliver this:

1. Cookies
2. Sessions
3. Encryption

We are going to cover Cookies and Sessions in this lecture and briefly touch on encryption as it applies to cookies and session week 12, we will go into more detail on encryption for other purposes.

Cookies

Cookies are pieces of data that are passed back and forth from the browser to the server that hold state information about th current audience interacting with your application.

Cookies are not set and maintained automatically you have to code this functionality into your application to support it.

As we have learned; each time a request is made from the browser or a response is sent from the server, there is a set of head attached to the data payload. Request headers contain the 'Cookie' header and response headers contain the 'Set-Cookie' hea These headers are a string of semi colon separated values that you can refer to in server side code through the 'req' object. Th common type of data we want to place in the cookie is a session value.

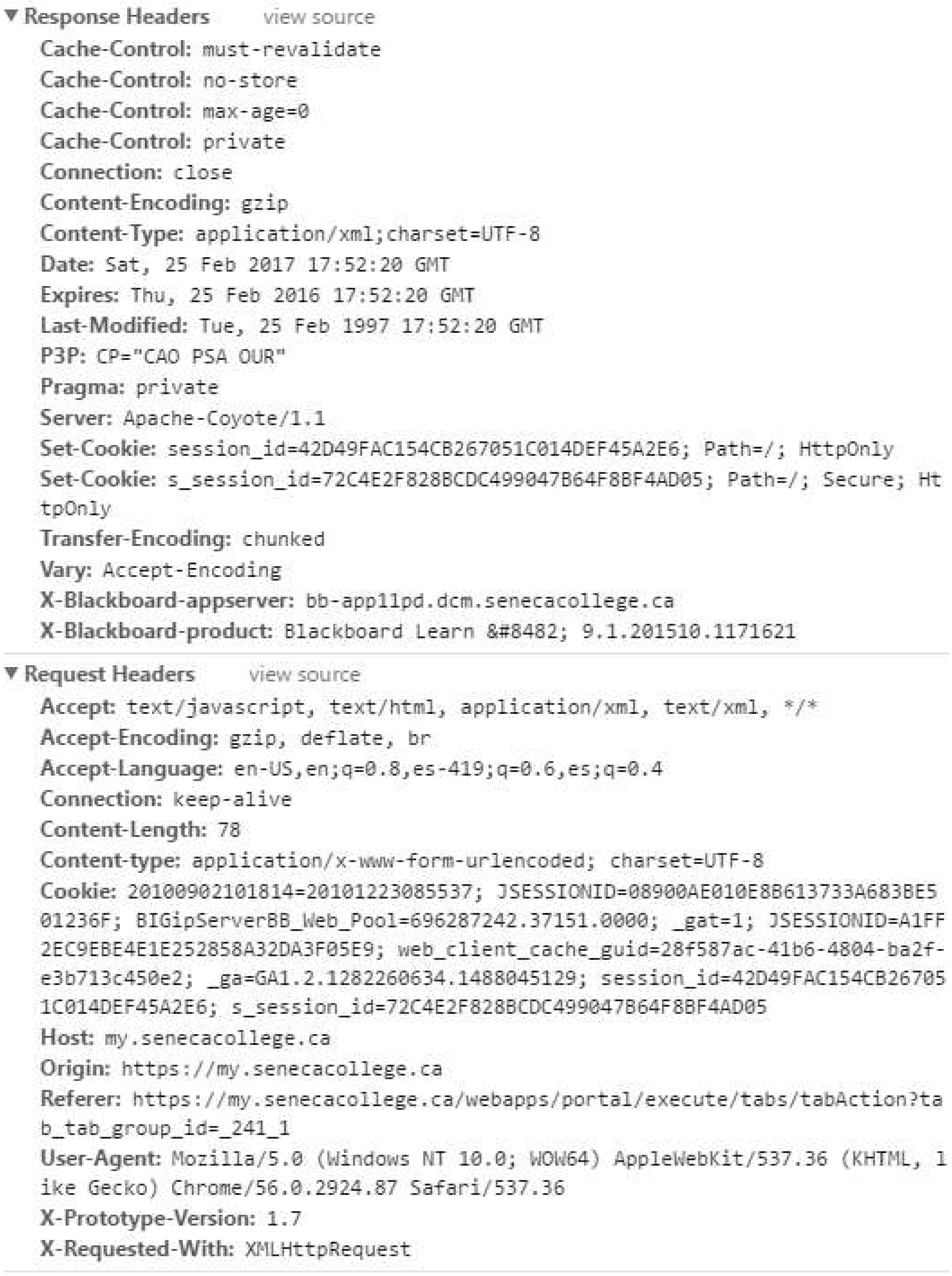
session contains information about the user, and the session is placed as one of many potential key:value pairs of data in one cookies

Here is an example of what a cookie might look like in the request header when inspecting it in the Chrome dev tools

> **Cookie:**_gat=4; _ga=GA1.6.14184624.14881926; session=9HQWRacCvAHDw7KTIw.P_aFNgqKkZrcySj7Jk5HFI8e6TNXBpRemrFczm3YiiVOs7gFG7qZ1GFXJ5Uw-tyOrL6VKd-QoDpYl2INZ1SVjPvt36nq3GOq3kTDh2r9lt580UIl0dNb0Wj_b5IjCOaPnrpqPg-SViSpvQXS7TtRhv6T-FO9UrSRiVWXqt-QBFzLxncbkVgwotroHNRPvjLtdG8ReU34OaD1uQyz5odSpmXlyY_kk.1488044367536.1209600000.M-IJbIdxARUmP-5ummaUOIzmxzWYvbPxawLoBz74

And here is an example of the 'Set-Cookie' header in a response from the server to update it after a request

Set-Cookie:session=z80OxktaQTVma-Eclg.QfJOeQ6sTt2g7HfqjpqHlLsuC5ZFYltXmbs_5XuC5Naw-qVuYEItIYXOQZbEwhaEdcR7R35bmnUPLElJuWirq-AUS_K79TO7k9KhXrgUZvQb6gQxa_bcGBoH6onPMhCotDXagGXS2U-MjdsdCo2gtPXpjh0QkPw-oe3-oQr7kj2JidXlSlYweE8sFYsldfBPG3zDMly0MXbhOI4.14880447805.12090000.SRbFKSnQ1-tbEe8NTNxqQHPjnvOhUdC5B4VYs; path=/; expires=Sat, 11 Mar 2017 17:39:28 GMT; secure; httponly



```
▼ Response Headers      view source
   Cache-Control: must-revalidate
   Cache-Control: no-store
   Cache-Control: max-age=0
   Cache-Control: private
   Connection: close
   Content-Encoding: gzip
   Content-Type: application/xml;charset=UTF-8
   Date: Sat, 25 Feb 2017 17:52:20 GMT
   Expires: Thu, 25 Feb 2016 17:52:20 GMT
   Last-Modified: Tue, 25 Feb 1997 17:52:20 GMT
   P3P: CP="CAO PSA OUR"
   Pragma: private
   Server: Apache-Coyote/1.1
   Set-Cookie: session_id=42D49FAC154CB267051C014DEF45A2E6; Path=/; HttpOnly
   Set-Cookie: s_session_id=72C4E2F828BCDC499047B64F8BF4AD05; Path=/; Secure; Ht
   tpOnly
   Transfer-Encoding: chunked
   Vary: Accept-Encoding
   X-Blackboard-appserver: bb-app11pd.dcm.senecacollege.ca
   X-Blackboard-product: Blackboard Learn &#8482; 9.1.201510.1171621
▼ Request Headers      view source
   Accept: text/javascript, text/html, application/xml, text/xml, */*
   Accept-Encoding: gzip, deflate, br
   Accept-Language: en-US,en;q=0.8,es-419;q=0.6,es;q=0.4
   Connection: keep-alive
   Content-Length: 78
   Content-type: application/x-www-form-urlencoded; charset=UTF-8
   Cookie: 20100902101814=20101223085537; JSESSIONID=08900AE010E8B613733A683BE5
   01236F; BIGipServerBB_Web_Pool=696287242.37151.0000; _gat=1; JSESSIONID=A1FF
   2EC9EBE4E1E252858A32DA3F05E9; web_client_cache_guid=28f587ac-41b6-4804-ba2f-
   e3b713c450e2; _ga=GA1.2.1282260634.1488045129; session_id=42D49FAC154CB26705
   1C014DEF45A2E6; s_session_id=72C4E2F828BCDC499047B64F8BF4AD05
   Host: my.senecacollege.ca
   Origin: https://my.senecacollege.ca
   Referer: https://my.senecacollege.ca/webapps/portal/execute/tabs/tabAction?ta
   b_tab_group_id=_241_1
   User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, l
   ike Gecko) Chrome/56.0.2924.87 Safari/537.36
   X-Prototype-Version: 1.7
   X-Requested-With: XMLHttpRequest
```

An example of Cookie and Set-Cookie headers on blackboard

As you can see, the Cookie header contains the session key and value pair but it can also contain other things as well dependi what the website wants to keep in state between requests from the user. You can also see that its just a jumbled up mess. Tha because this particular example uses an encrypted session. Using encrypted sessions is **EXTREMELY** important in a productio environment. If you do not encrypt your sessions and cookies then a user can look at the request and response headers and potentially pull out very sensitive information about the user those cookies are meant for. As a general rule, all sessions that a persistent between the client and server should always be encrypted and also always sent over HTTPS as well. We will cover H encryption in week 12.

Let’s move on to talk more about sessions and implementing them.

Sessions

Implementing sessions is now a fairly simple task in Express.js. The platform has matured and libraries have been tested with thousands of websites and billions of logins. These libraries are easy to use and integrate into your own projects.

The most popular library for implementing client sessions is Mozilla’s client-sessions node library. This library focuses on keep sessions between the client and server on the client side. This has several advantages over storing the session on the server s keeping track of it on the server. For example, if the server restarts and you do not save session information in a persistent st location, the sessions will be lost and all your users will be logged out anytime the server restarts.

It is also beneficial to keep your sessions on the client side (and have them continuously sent with each request), because this you to host a web site that has multiple web servers and the session will still be maintained regardless of which server is used process the response. In an environment where sessions are kept track of on the server and you have multiple web servers th load balanced, you will need to make sure each users’ requests are always sent to the same web server with each request to p their session. Alternatively, you could attempt to replicate server session information between web servers, however this can complex. For these reasons, storing the session on the client makes scaling and session management a lot easier.

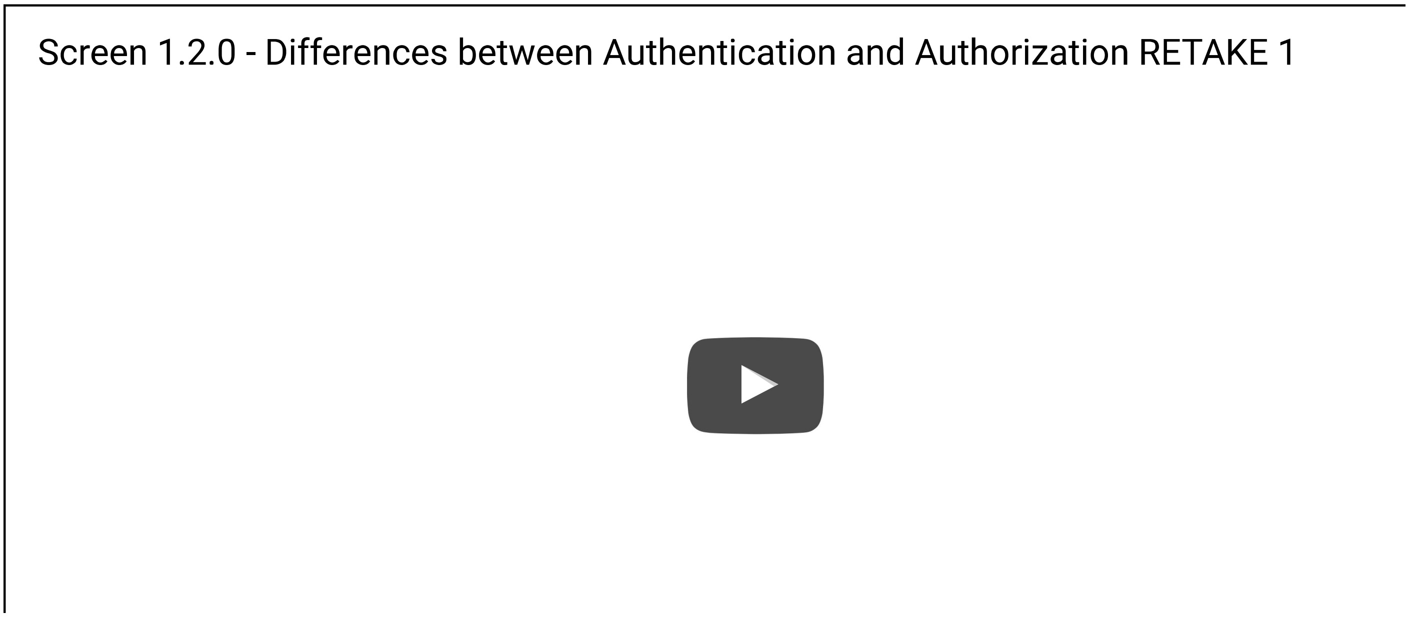
Authentication vs Authorization

Authentication and Authorization mean two entirely different things. It can be easy to confuse them, so let’s discuss them a lit before we begin to implement sessions and secure endpoints. This will enable you to be more comfortable explaining and del the two different concepts.

Authentication is the answer to “Who are you?”. It involves supplying credentials to identify yourself to the server and establis session for your user account.

Authorization is the answer to “What do you have access to?”. It involves checking your permissions to resources you have req and acting accordingly. You may be authenticated with the server and have a user session but you might not be authorized to certain resource. (No permissions!)

Here is a great 1 min video that explains it from MongoDB University. MongoDB also has the concept of authentication and authorization.



Screen 1.2.0 - Differences between Authentication and Authorization RETAKE 1

Status codes

There are a number of standard http response status codes that can be used by your application to inform the browser of wh request was rejected because of an authentication problem or an authorization problem.

- 401 status code (Unauthorized) Authentication error. The resource exists but it requires the user to be authenticated fi view it. It may also require permissions and be checked again after authenticating for proper authorization.
- 403 status code (Forbidden) Authorization error. The resource exists but the user does not have permission to view it.
- 404 status code Not Found. The resource that was requested was not found on the server. This is commonly used whe requested that just doesn't exist.

Adding sessions to an Express.js application

Now that we've explained what cookies, sessions, authentication, authorization, and status codes are, we can get to the codin the lecture and see how it all comes together.

We will break down the project to add client sessions handling into 4 parts:

1. Add the client-sessions library to the project.
2. Create a middleware function to setup the client-sessions configuration.
3. Write a login route handler to process a POST to /login and establish **authentication** of a user session.
4. Add a middleware function that handles checking for **authorization** of a user for a specific resource.

Step 1: Add the client-sessions library to the project.

We will need to npm install the client-sessions library and –save it to the package.json file

```
npm install client-sessions --save
```

Then we just have to "require" it in our server file.

```
const express = require("express");
const app = express();
const bodyParser = require("body-parser");
const exphbs = require("express-handlebars");
const clientSessions = require("client-sessions");

const HTTP_PORT = process.env.PORT || 8080;
```

Step 2: Create a middleware function to setup client-sessions.

Now we need to register clientSessions as a middleware and configure it, as well as register handlebars as a view rendering er and bodyParser to handle our application/x-www-form-urlencoded POST data.

```
// Register handlebars as the rendering engine for views
// Register handlebars as the rendering engine for views
app.engine(".hbs", exphbs({ extname: ".hbs" }));
app.set("view engine", ".hbs");

// Setup the static folder that static resources can load from
// like images, css files, etc.
app.use(express.static("static"));

// Setup client-sessions
app.use(clientSessions({
  cookieName: "session", // this is the object name that will be added to 'req'
  secret: "week10example_web322", // this should be a long un-guessable string.
  duration: 2 * 60 * 1000, // duration of the session in milliseconds (2 minutes)
  activeDuration: 1000 * 60 // the session will be extended by this many ms each request (1 mi
}));

// Parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: false }));
```

Step 3: Write a login route handler

Now we need to setup a login route that will check that the username and password match the user who tried to login. We wil login.hbs, and dashboard.hbs templates for the login and dashboard pages.

```
// call this function after the http server starts listening for requests
function onHttpStart() {
  console.log("Express http server listening on: " + HTTP_PORT);
}

// A simple user object, hardcoded for this example
const user = {
  username: "sampleuser",
  password: "samplepassword",
  email: "sampleuser@example.com"
};

// Setup a route on the 'root' of the url to redirect to /login
app.get("/", (req, res) => {
  res.redirect("/login");
});

// Display the login html page
app.get("/login", function(req, res) {
  res.render("login", { });
});

// The login route that adds the user to the session
```

```
app.post("/login", (req, res) => {
  const username = req.body.username;
  const password = req.body.password;

  if(username === "" || password === "") {
    // Render 'missing credentials'
    return res.render("login", { errorMsg: "Missing credentials." });
  }

  // use sample "user" (declared above)
  if(username === user.username && password === user.password){

    // Add the user on the session and redirect them to the dashboard page.
    req.session.user = {
      username: user.username,
      email: user.email
    };

    res.redirect("/dashboard");
  } else {
    // render 'invalid username or password'
    res.render("login", { errorMsg: "invalid username or password!"});
  }
});

// Log a user out by destroying their session
// and redirecting them to /login
app.get("/logout", function(req, res) {
  req.session.reset();
  res.redirect("/login");
});

app.listen(HTTP_PORT, onHttpStart);
```

Step 4: Add a middleware function that checks for authorization

The last thing we need to do is add the dashboard route and add a middleware function to check for authentication before re
the dashboard.

```
// An authenticated route that requires the user to be logged in.
// Notice the middleware 'ensureLogin' that comes before the function
// that renders the dashboard page
app.get("/dashboard", ensureLogin, (req, res) => {
  res.render("dashboard", {user: req.session.user});
});
```

At the top of the file just after we declared the static hardcoded user, we can add a ensureLogin function that is called to checl user is logged in and redirect them to login if they are not, otherwise continue on to the next middleware in the route handler

```
// This is a helper middleware function that checks if a user is logged in
// we can use it in any route that we want to protect against unauthenticated access.
// A more advanced version of this would include checks for authorization as well after
// checking if the user is authenticated
function ensureLogin(req, res, next) {
  if (!req.session.user) {
    res.redirect("/login");
  } else {
    next();
  }
}
```

Now we can create the login.hbs and dashboard.hbs files in the /views folder

login.hbs

```
<!doctype html>
<html>

<head>
  <title>Login</title>
</head>

<body>
  {{#if errorMsg}}
    <h5>{{errorMsg}} </h5>
  {{else}}
    <h5>Please login below</h5>
  {{/if}}
  <form method="post" action="/login">
    <label>Username</label><input name="username" type="text">
    <label>Password</label><input name="password" type="password">
    <button type="submit">Login</button>
  </form>
</body>

</html>
```

dashboard.hbs

```
<!doctype html>
<html>

<head>
  <title>Dashboard</title>
  <title>Dashboard</title>
</head>

<body>
  <h3>Hello {{user.username}}</h3>
  <p>Welcome to your dashboard</p>
  <p>Here is the information we have on file for you:</p>
  <h4>username: {{user.username}}</h4>
  <h4>email: {{user.email}}</h4>
  <a href="/logout">Logout</a>
</body>

</html>
```