Chapter 13

Authentication, Authorization & Sessions

What is in This Chapter ?

This chapter discusses how we can add **authentication** and **authorization** to our websites by means of **sessions** and **cookies**. We look briefly at how the **cookie-parser** and **express-session** middleware can make our lives easier. We conclude with a simple example of how to do authentication via username/password **HTTP** forms and how to authenticate users before they access particular routes.

13.1 Authentication and Authorization

When building web apps, we often need to know who is using our site and what they are allowed to do because most web apps handle things that are personal, private, or restricted. Here are some common examples:

- **Personalization** – a shopping site needs to know who we are so it can show our cart, order history, recommendations, etc..
- **Security** – a banking app must ensure that only we can see our account balance and transfer money.
- **Access control** – a school portal might let students view their own grades, but only teachers can update them.
- **Trust** – users expect that their data and actions are safe, private, and inaccessible to strangers.

To make this possible, websites rely on three core ideas:

- ***authentication*** - proving our identity (e.g., logging in with a username and password)
- ***authorization*** - deciding what that user is allowed to access (e.g., admin vs. guest privileges)
- ***sessions*** - remembering the user as they move from page to page (e.g., cookies)

Together, these form the foundation for secure, personalized web applications. Let's look a little more at the first two of these.

> ***Authentication is the process of verifying the identity of a user or process.***

In other words, it's about proving that someone really is who they claim to be. The most common authentication method is that of supplying a username and password, but many sites add extra steps such as security questions, codes sent to our phone, security images, etc.. Here is a list showing examples of the various ways to perform authentication:

- **Login form** - requiring the user to submit a username and password on a webpage.
- **AJAX-based login** - sending login information in the background (often used in single-page applications).
- **HTTP authentication** - the browser itself prompts for a username and password.
- **Token-based login** - the server issues a token that the browser/app sends with each request.
- **Third-party login** - using providers like Google, Facebook, or GitHub to sign in.
- **Multi-factor authentication (MFA)** - adding an extra step, such as a code sent to our phone, on top of a password.

***Authorization* is the process of deciding what an authenticated user is allowed to do.**

For example:

- A **regular user** might be allowed to view their own profile, but not edit someone else's.
- An **admin user** might have permission to add or delete users, while others cannot.
- A **guest** (not logged in) may only be able to browse public pages.

Authorization is usually achieved through access controls, which can be applied at different levels:

- **URLs or routes** - certain paths on the server (like **/admin**) are restricted to certain roles.
- **Database queries** - filtering what data a user can see (e.g., only their own orders).
- **Features and actions** - deciding whether buttons, forms, or actions are visible/usable based on role.

Let's consider an example of using a **URL** endpoint. Assume that a client sends the request **PUT /users/dave**. The server will use the **URL** to figure out which resource is being targeted. In this case, the server sees **/users/dave** and interprets it as "the profile of the user named **dave**." This is part of how **REST**ful **API**s work … the **URL** path identifies the specific resource 😉.

Next, the server needs to know who is making the request. This is done through **authentication**, usually via a session, cookie, or token sent with the request (we will talk more about this later). For example, if Dave is logged in, the browser might include a session cookie that proves the request is coming from Dave.

At this point, the server knows the resource being requested as well as the identity of the requester so it can perform the authorization:

- If this is indeed Dave trying to update his own profile, the server allows it.
- If it is not Dave (and not **admin**) trying to update this profile, the server denies it.

In practice, the server always checks a user's permissions or roles before performing sensitive actions, ensuring only authorized users can access or modify resources.

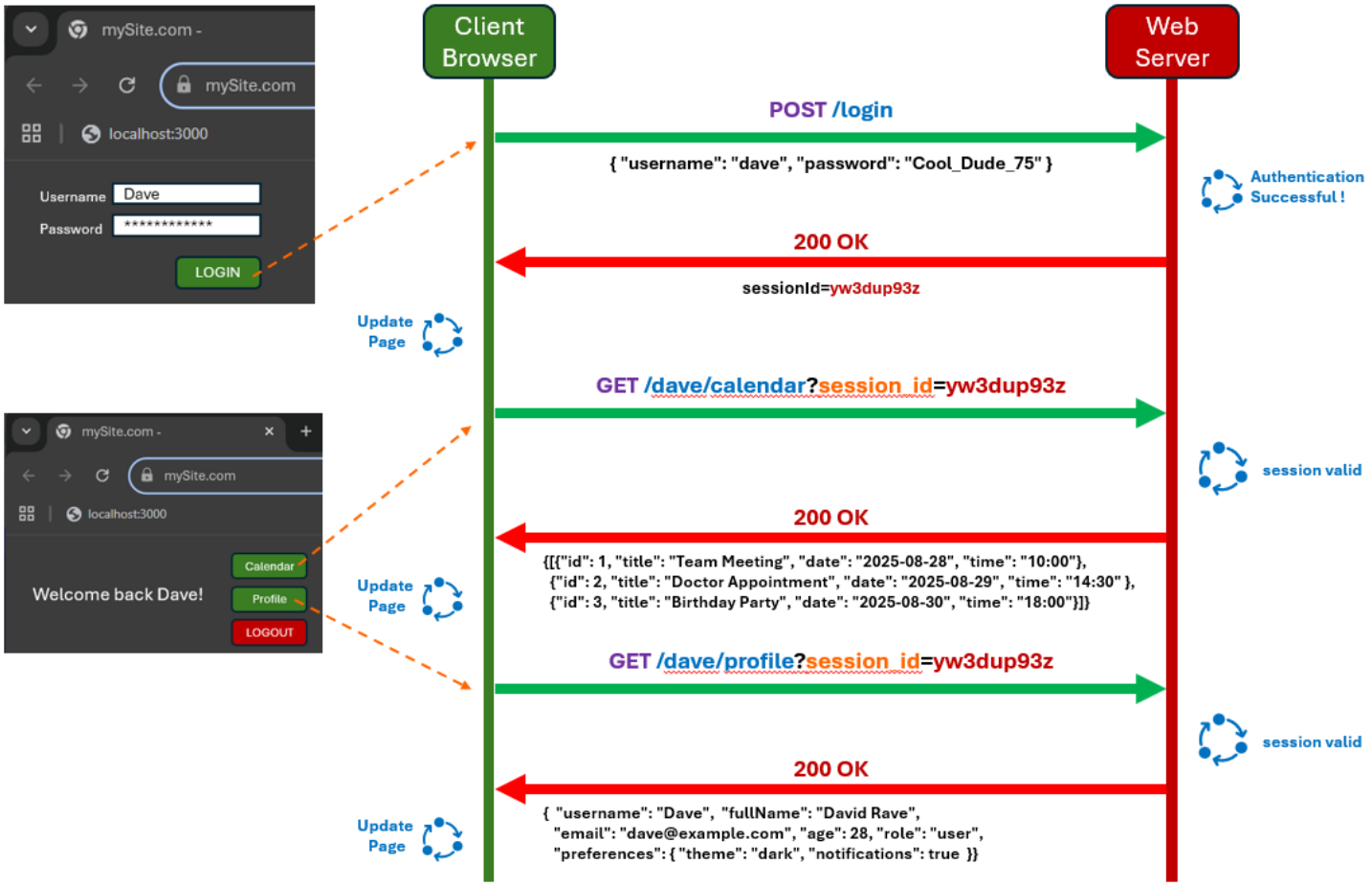
When a user logs in and then makes multiple requests to a server, how does the server know that its is the same user each time? After all, **HTTP** is stateless (i.e., each request is independent and doesn't automatically remember previous interactions).

The simplest approach would be to have the client include their username and password each time that they make a request. The server can then easily authenticate each request and make the appropriate decisions regarding authorization. But there are downsides to this, as we will discuss in a moment.

13.2 Sessions and Cookies

A more common approach is to have the server send a **session ID** to the client after they login. There are two main ways to send a **session ID** with a user's request.

(1) The first approach is to use query parameters. In this case, the user logs in, perhaps with a form **POST**. The server authenticates the credentials and decides if the person is authorized to continue using the site. If they are, the server generates a unique **session ID** which will be associated with this authenticated user on the server. The server stores that **session ID** (e.g., locally or in a database), and then replies to the client with the **session ID**. The client then passes that **session ID** with each request as a query parameter. The server can then check each time to see if the **session ID** parameter is there, if it is valid and if the user associated with that **session ID** is authorized for the given request. If not authorized (e.g., the session timed out), a **401 Not Authorized** response can be returned. Here is a visual of what happens:



The advantage of sending **session ID**s as query parameters is that the username and password do not need to be sent with every request. The server can invalidate a **session ID** at any time by removing it from its database, and sessions can also have an expiry time to limit their lifespan. As an analogy, the **session ID** acts like a "ticket" in that it proves the user has already logged in, so they don't have to re-enter credentials every time.

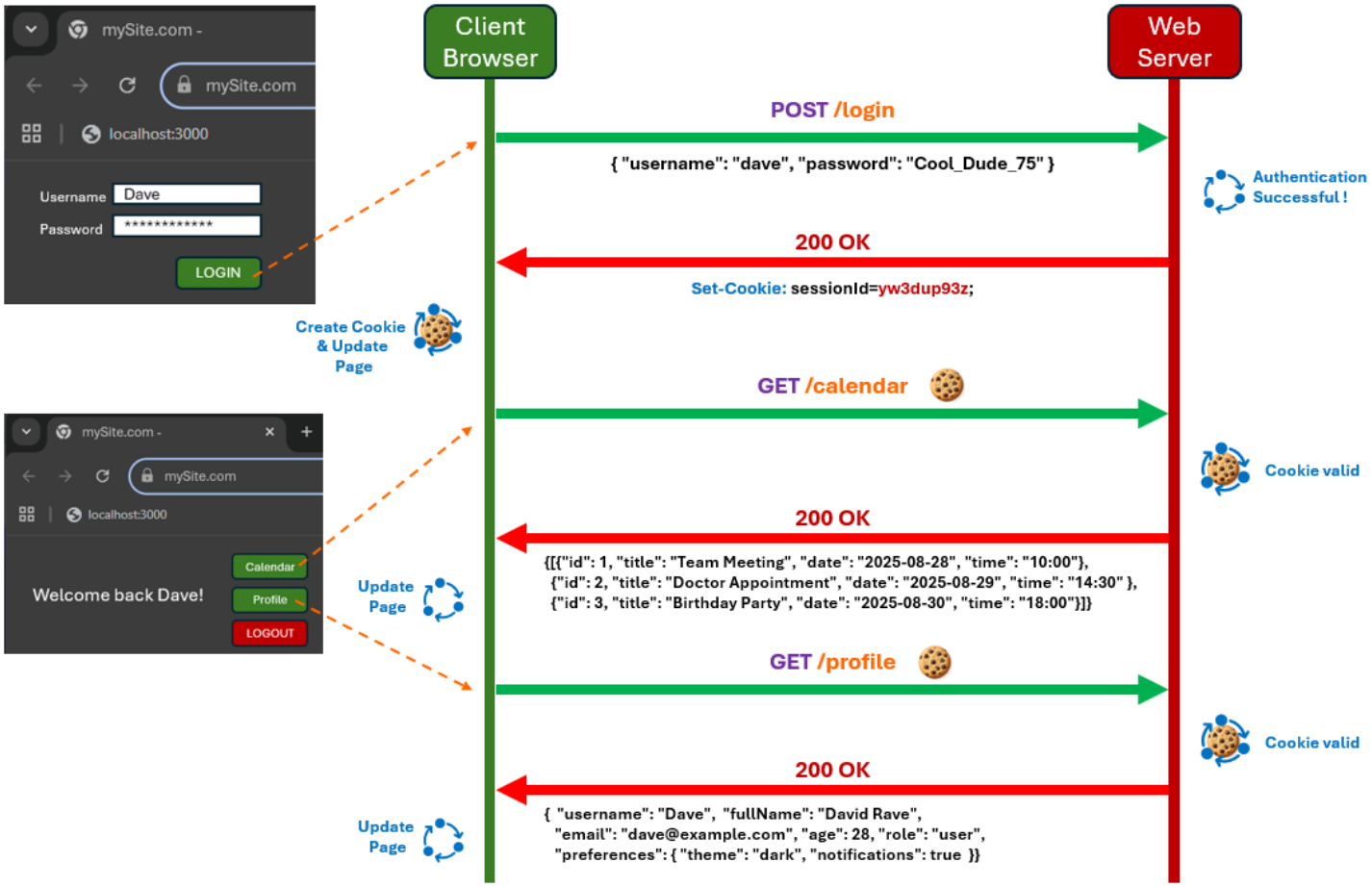
Session IDs are usually long, random, and hard to guess. This is important for security, so that attackers cannot predict or "guess" someone else's **session ID** and hijack their session. They are typically random strings with enough characters to make brute-force attacks infeasible. Sometimes they include letters, numbers, and symbols to increase complexity.

So, this approach works fine. However, because the **session ID** is included in the **URL**, it is transmitted with every request. Even if **HTTPS** encrypts the transmission, **URL**s may still appear in server logs, browser history, or shared links … creating a potential security risk. For this reason, using query parameters is generally only recommended for short-lived sessions or low-risk scenarios.

(2) The second approach (which is a better fit for browsers), is to use cookies:

A cookie is a small piece of data sent by the server that stores a session ID to identify the user on future requests.

The "query parameter" approach and the "cookie" approach are similar in that they both send a **session ID** with each request, allowing the server to identify a logged-in user. The difference lies in where and how the **session ID** is stored and transmitted. The "query parameter" approach includes the **session ID** directly in the **URL**, which can create potential security risks. Cookies, on the other hand, are stored by the browser and sent automatically in the **HTTP** headers, keeping the session ID out of the **URL**. The overall approach is similar to the "query parameter" approach:



As with the "query parameter" approach, the **session ID** is sent back to the client with a request for the client to store it in a cookie. This cookie is then sent with every subsequent request. Each time, the server will verify that the **session ID** is valid and still active (e.g., not timed out). Once this is confirmed each time, the server can identify the user and apply authorization rules to decide what resources/actions are allowed.

Although this looks like it has almost the same kind of interaction as the "query parameter" approach, the "cookie" approach is more powerful and secure. A cookie is much more than a simple **session ID** because it includes additional attributes that control how the data is used. The server defines the rules for these cookies according to various attributes and the browser enforces them by deciding *when* and *whether* to include the cookie in requests (as well as when to delete it).

- **`Name`** & **`Value`** - the actual data stored (i.e., often it is the **session ID** and its value).
- **`Expires`** - the date and time that the cookie expires. Without an expiry, the cookie is a *session cookie* and is deleted when the browser closes. The browser automatically deletes the cookie once the set time is reached. The user can't use it after that, even if it's still stored.
- **`Max-Age`** - number of seconds until the cookie expires. A zero or negative number will expire the cookie immediately. This attribute takes precedence over "Expires".
- **`Domain`** - the browser will only send the cookie to servers matching this domain. For example, a cookie for **oneSite.com** won't be sent to **anotherSite.com**.
- **`Path`** - the browser only sends the cookie for requests that match the given path. For example, a cookie with path **/products** won't be sent to **/users**.
- **`Secure`** - the browser only sends the cookie over **HTTPS**, never over plain **HTTP**.
- **`HttpOnly`** - the browser blocks **JavaScript** from accessing the cookie (i.e., **`document.cookie`** can't read it). Only the server can see it.
- **`SameSite`** - the browser decides whether the cookie is sent with cross-site requests.

These attributes make cookies flexible and secure, allowing the server to control when, where, and how the **session ID** (or other data) is transmitted, rather than just storing a single value.

In summary, the use of cookies and **session ID**s is more secure and practical than sending the username and password with every request because …

- ✓ **Reduced exposure of credentials** - the username/password is sent only once during login, and the **session ID** is used for subsequent requests.
- ✓ **Automatic expiration** - sessions can expire after a set time or when the browser closes. Even if an attacker steals a **session ID**, they can only use it until the session ends.
- ✓ **Logout support** - the server can invalidate a **session ID** at any time, for example, when the user logs out or if suspicious activity is detected.

So, we will use **session ID**s since they are more secure and practical.

Let's talk a little more about cookie-based sessions. Cookies are mainly used for three purposes:

1. **Session management** - keeping users logged in, storing shopping carts, game scores, or any information the server should remember across requests.
2. **Personalization** - remembering user preferences, themes, language settings, or other customizations.
3. **Tracking** - recording and analyzing user behavior, such as page visits, clicks, or usage patterns.

We can have multiple cookies being passed on each request. Here is an example of an **HTTP** response header asking the browser to create three new cookies (or update them):

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: theme=light; Path=/; Max-Age=86400; SameSite=Lax
Set-Cookie: sessionToken=abc123; Path=/; HttpOnly; Secure;
            Expires=Wed, 22 Nov 2023 11:18:14 GMT; SameSite=Strict
Set-Cookie: language=en-US; Path=/; Max-Age=604800; SameSite=Lax
```

This example demonstrates how a server can manage sessions, user preferences, and other settings simultaneously using cookies. The browser stores all three cookies and automatically includes them in future requests to the appropriate domain and path, respecting each cookie's attributes such as expiration, security flags, and scope.

Since cookies are sent with every request, they should generally be small to avoid performance issues. Current cookie specifications require browsers to support at least **4KB** per cookie, at least **50** cookies per domain, and at least **3000** cookies in total, though actual browser limits may be higher.

On future requests, the client browser may send headers that look like this:

```
GET /dashboard HTTP/1.1
Host: example.com
Cookie: theme=light; sessionToken=abc123; language=en-US
```

Notice that only the cookie **name** and **value** are sent. The other attributes (i.e., `Path`, `Expires`, `HttpOnly`, `Secure`, `SameSite`) are not included in the request. They only affect how and when the browser sends the cookie, not the content of the header itself.

Tracking Cookies

When we visit a website (e.g., **example.com**), the server may store a cookie on our device — this is called a ***first-party* cookie, since it comes from the site we are directly visiting**.

However, many websites also load content from other domains (i.e., ads, images, or social media widgets). When our browser requests these resources, those external servers can also store cookies on our device. These are known as ***third-party* cookies, because they come from domains other than the one we're currently visiting**.

When our browser makes requests to a server, it automatically includes any cookies previously set by that server. For example:

1. We visit **example.com**, which loads an advertisement from **ads.com**.
2. The request to **ads.com** sets a cookie in our browser (e.g., **id=someID**).
3. Later, we visit **example2.com**, which also loads an ad from **ads.com**.
4. Our browser will send the *previously set cookie* to **ads.com** often including information such as the referring page (i.e., **example2.com**).

Step 4 happens because that "previously-set" cookie … assuming it has not expired … is associated with the **ads.com** domain and browsers automatically include **all cookies for a domain** whenever a request is made to that domain, regardless of which site is actually loading the content).

As a result, **ads.com** can track that the same user visited both **example.com** and **example2.com**. This is how third-party cookies enable ***cross-site tracking***.

This is a simple example, but processing cookies can reveal a lot of information about our browsing history, such as (a) **Which URLs we visited**, (b) **What our query parameters were**, (c) **The times we visited each site.** Even seemingly harmless cookies can be combined to build detailed profiles of user behavior.

Many companies take advantage of long-lived cookies to track users across multiple websites, often without the user realizing it. These cookies can store unique IDs that let advertisers and analytics services **reconstruct browsing histories, preferences, and behaviors over months or even years**, creating detailed user profiles that are largely invisible to the person being tracked.

There are several practical ways to protect ourselves from unwanted tracking via cookies, especially third-party cookies:

- Block third-party cookies
- Use private/incognito mode
- Clear cookies regularly
- Use browser extensions or privacy-focused browsers
- Pay attention to cookie consent dialogs
- Disable cross-site tracking in browser settings

Cookies themselves aren't inherently bad, but being mindful of third-party cookies and controlling them helps protect our privacy.

13.3 Middleware: cookie-parser & express-session

Let's make a server that will use a cookie to track how many times we have visited the webpage at **http://localhost:3000/visit**. We will do this without any helpful modules by creating our own `sessionId` values and storing them locally on the server in an object called `sessions`:

```
const sessions = {};
```

We will store the `sessionId` values from the cookies that will be passed back and forth.

One way to create a `sessionId`, is to use a random number generator. This code, for example represents one way to do this:

```
Math.random().toString(36).slice(2);
```

It first generates a random number from 0 to 1 (e.g., **0.3749285147**). Then it converts it to a base-36 string which uses the digits from 0-0 and characters a-z (e.g., `"0.drk4h8v7z"`). Then it removes the first two characters so that we have a nice random `sessionId` (e.g., `"drk4h8v7z"`). However, this is not cryptographically secure because an attacker could potentially guess or brute-force the ID due to the amount of randomness that it provides being limited.

Instead, it is better to make use of the crypto module, which is more secure and readily available. Here is a better way to produce a `sessionId`:

```
const crypto = require('crypto');
const sessionId = crypto.randomBytes(16).toString('hex'); // 32 hex characters (128 bits)
```

This creates a securely random session ID like this: `"a7b3c92e7f1046c8a3d7e8cbd52a4e61"`.

Now, what will our cookie look like? It is simple … it will just be this `sessionId` and will look like this:

```
sessionId=a7b3c92e7f1046c8a3d7e8cbd52a4e61
```

To create a cookie, our server will need to send an **HTTP** response with a **Set-Cookie** header that contains the `sessionId` as follows:

```
Set-Cookie: sessionId=a7b3c92e7f1046c8a3d7e8cbd52a4e61; HttpOnly; Path=/
```

The `HttpOnly` option keeps it safe from **JavaScript** and `Path=/` makes it valid for the whole site.

Later, the browser will automatically send this cookie with each request. This allows the server to recognize the user (distinguishing them from other clients) and track how many times they have visited the page.

So, in our server code, each time a request comes in, we will check if the **sessionId** matches one that we have stored. If it does, increase the number of views for that session and increase it.
If it does not match, then this is must be a new client request and we will give it a new **sessionId** and then start counting that client's views as well.

Now, since the browser only sends cookies as a plain string in the request header, we need a way to parse that string into key-value pairs we can check in code. Since we only have one key/value pair (i.e., the **sessionId**), this is not too difficult. However, in general, the browser may send multiple cookies with each request (e.g., if we enhance our code later) like this:

```
"sessionId=abc123xyz; theme=dark; lastVisit=2025-10-17"
```

So, we need a way to convert this string into this object:

```
{
    sessionId: "abc123xyz",
    theme: "dark",
    lastVisit: "2025-10-17"
}
```

Here is a function to do this:

```
// Helper function to parse the cookies
function parseCookies(cookieHeader) {
    const cookies = {};
    if (!cookieHeader)
        return cookies;
    cookieHeader.split(';').forEach(pair => {
        const [name, value] = pair.trim().split('=');
        cookies[name] = value;
    });
    return cookies;
}
```

The code first separates each cookie into substrings by splitting using the ";" separator, then finds the key/values by splitting the substrings using the "=" separator.

To get the cookie string that we need to pass into this parsing function, we simply ask the request for its list of headers and then for the cookie header as follows:

```
const cookies = parseCookies(req.headers.cookie);
let sessionId = cookies.sessionId;
```

Now it is time to look at our server code. It is simple, since we will just handle a **GET** for the **/visit** page.

```
const crypto = require('crypto');
const express = require('express');
const app = express();
const PORT = 3000;
```

```
// This is where we will store all session information
const sessions = {};

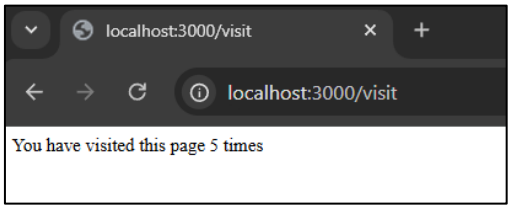
// Helper function to parse the cookies
function parseCookies(cookieHeader) {
    // … code already described …
}

// Sample route to /visit URL
app.get('/visit', (req, res) => {
    const cookies = parseCookies(req.headers.cookie);
    let sessionId = cookies.sessionId;

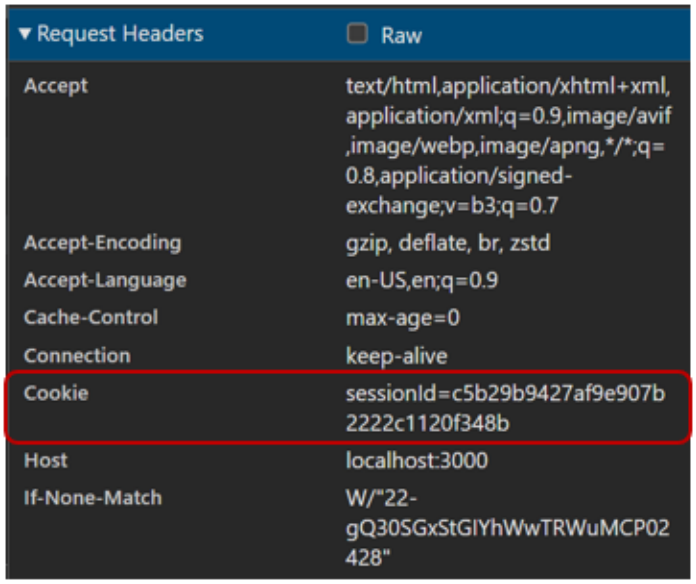
    // If there is no session or this will be a new one, create one
    if (!sessionId || !sessions[sessionId]) {
        const sessionId = crypto.randomBytes(16).toString('hex'); // 32 hex characters (128 bits)
        sessions[sessionId] = { views: 1 };
        res.setHeader('Set-Cookie', `sessionId=${sessionId}; HttpOnly; Path=/`);
        res.send('You have visited this page 1 time');
    } else {
        // Increment existing session views
        sessions[sessionId].views++;
        res.send(`You have visited this page ${sessions[sessionId].views} times`);
    }
});

app.listen(PORT);
console.log("Server listening on port " + PORT);
```

If we run this code, we see the counter increase each time that we visit the page:



We can also see (in Chrome DevTools) that our cookie is being passed each time in the request header →



Notice how much "tedious" work was involved:

(a) we need to parse the cookies,

(b) we need to generate our own `sessionId` and

(c) we need to manually set up the cookie attributes.

Thankfully, there are a few modules that will help to reduce the tedious coding involved. The first is a cookie-parser module:

https://expressjs.com/en/resources/middleware/cookie-parser.html

To uses it, we should install it: **npm install cookie-parser**

Let's look at how this middleware module simplifies cookie handling, so we don't have to manually parse the cookie header:

```
const crypto = require('crypto');
const express = require('express');
const app = express();
const PORT = 3000;
const cookieParser = require('cookie-parser');

// This is where we will store all session information
const sessions = {};

// Use cookie-parser middleware
app.use(cookieParser());

// Sample route to /visit URL
app.get('/visit', (req, res) => {
    let sessionId = req.cookies.sessionId;

    // If there is no session or this will be a new one, create one
    if (!sessionId || !sessions[sessionId]) {
        const sessionId = crypto.randomBytes(16).toString('hex'); // 32 hex characters (128 bits)
        sessions[sessionId] = { views: 1 };

        // Set the cookie with HttpOnly and Secure flags
        res.cookie('sessionId', sessionId, {
            httpOnly: true,
            secure: false,  // allow cookies over HTTP (needed for localhost dev)
            path: '/'
        });
        res.send('You have visited this page 1 time');
    } else {
        // Increment existing session views
        sessions[sessionId].views++;
        res.send(`You have visited this page ${sessions[sessionId].views} times`);
    }
});

app.listen(PORT);
console.log("Server listening on port " + PORT);
```

Notice that things are simpler now. We install the middleware and then the cookies are automatically parsed so that we no longer need the helper function 😊. Also, we can use `res.cookie()` now instead of `res.setHeader()` to set the cookie attributes in a simpler way.

Now, let's simplify things even further by using the **express-session** middleware module:

https://expressjs.com/en/resources/middleware/session.html

We should install it as well: **npm install express-session**

This module will handle the sessions for us, so that we do not have to manually store them ourselves … nor do we have to come up with a new `sessionId` each time 😊. So, it will make it easy to manage user sessions. It creates a `req.session` object for each client, where we can store session-specific data.

Here is the revised code that makes use of this. We notice that there is no need for the **cookie-parser** module anymore since all of that is handled for us when the sessions are managed.

```
const express = require('express');
const session = require('express-session');
const app = express();
const PORT = 3000;

// Use express-session middleware
app.use(session({
    secret: 'mySecretKey',       // used to sign the session ID cookie
    resave: false,               // don't save session if unmodified
    saveUninitialized: true,     // create session for new clients
    cookie: {
        httpOnly: true,
        secure: false,  // allow cookies over HTTP (needed for localhost dev)
        path: '/'
    }
}));

// Sample route to /visit
app.get('/visit', (req, res) => {
    // If 'views' does not exist in session, initialize it
    if (!req.session.views) {
        req.session.views = 1;
        res.send('You have visited this page 1 time');
    } else {
        // Increment existing session views
        req.session.views++;
        res.send(`You have visited this page ${req.session.views} times`);
    }
});

app.listen(PORT);
console.log("Server listening on port " + PORT);
```

The code requires a bit of explanation. The parameter passed to **session()** is an *options* object that configures how **express-session** behaves. In this particular case, the *options* object tells **Express** how to handle session storage, when to create/save sessions, and how to configure the **session ID** cookie. Each property shown has a specific role:

- `secret: 'mySecretKey'`

  A string used to *sign* the **session ID** cookie. Signing the cookie ensures that the browser cannot tamper with the **session ID**. We should make it long and random to prevent attackers from guessing it.

- `resave: false`

  Normally, **express-session** saves the session back to storage on every request, even if nothing changed. Setting this to `false` prevents unnecessary saves, which improves performance.

- `saveUninitialized: true`

  This option controls whether the server should create a session for a client that hasn't stored any data yet (i.e., `true` = a session is created immediately when a user connects, even if no data is stored yet. `false` = the session is only created when we assign something to `req.session`).

- `cookie: { … }`

  This configures the cookie that stores the **session ID** in the browser.

Here are a few additional properties that we can set:

- `keys: ['key1', 'key2', 'key3']`

  This is used when we want cookie signing with multiple secrets. It is mostly relevant if we pass an array of secrets instead of a single secret. The server can rotate secrets over time: the first key signs new cookies, older keys are used to verify old cookies.

- `rolling: true`

  This controls whether the session cookie is reset (refreshed) on every response. If `true`, every response resets the cookie expiration, so the session lives longer as long as the user keeps interacting. If `false` (the default) the cookie expiration is only set once at creation, so inactivity can cause the session to expire.

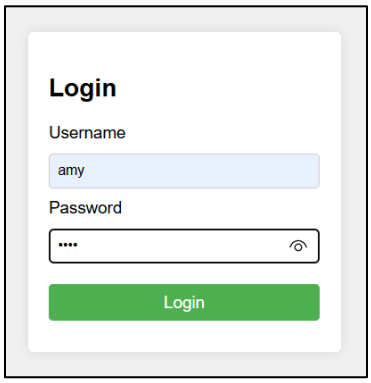
- ```
  store: MongoStore.create({
           mongoUrl: 'mongodb://localhost:27017/myDatabase',
           collectionName: 'sessions', //name of collection to store sessions
           ttl: 14 * 24 * 60 * 60   //session expiration in seconds (14 days)
         })
  ```

  This determines where session data is stored on the server. By default, sessions are stored in memory (i.e., `new session.MemoryStore()`), which is fine for testing but not for production. Common stores include: **MongoDB**, MySQL, etc.

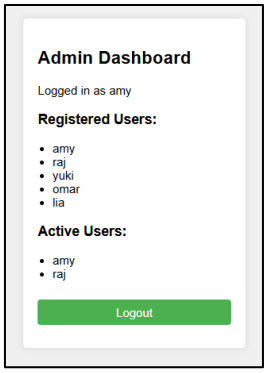
13.4 Example: Handling Login Sessions

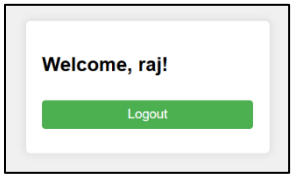
Now let's look at a more complete **Express**/**Pug** example that handles authentication, authorization and sessions. There will be **5** registered users (hard-coded) … two with admin privileges and three without. A user will be able to login from a main homepage which will simply be an **HTTP** form as shown on the next page.
  ```

When the user presses the **Login** button, the server will attempt to authenticate them to make sure that the username and password matches someone in the list of registered users. If there is no match, then a **"401 Not authorized. Invalid password"** error message will be returned.



Otherwise, one of two pages will be shown … either the **Admin Dashboard** page (for admin users) or a simpler **Welcome** page for non-admin users →





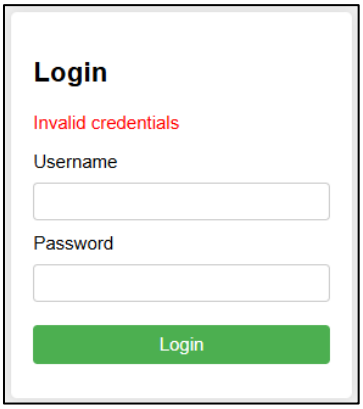
Both pages have a **Logout** button. The bonus for the admin users is that they get to see the list of all registered/active users.

The server will allow specific routes as described in the following table:

URL route http://localhost:3000	Any User (if not logged in)	Admin User (if logged in)	Non-Admin User (if logged in)
/	goes to login page	goes to login page	goes to login page
/logout	goes to login page	logs user out, goes to login page	logs user out, goes to login page
/admin	"Unauthorized"	goes to admin dashboard page	"Unauthorized"
/welcome	goes to login page	goes to welcome page	goes to welcome page

There are 3 **PUG** pages, one for each of the three views shown in the images above.

The **login.pug** page is basically just an **HTTP** form with a **Login** submit button. However, it does allow one incoming variable that represents a potential error message (i.e., "Invalid credentials") which allows this page to be re-rendered with an error message above the form as shown in the image here →




```
doctype html
html
    head
        title Login
        link(rel="stylesheet", href="/style.css")
    body
        .container
            h2 Login
            if error
                    p.error #{error}
            form(method="POST", action="/login")
                label(for="username") Username
                input(type="text", name="username", id="username", required)

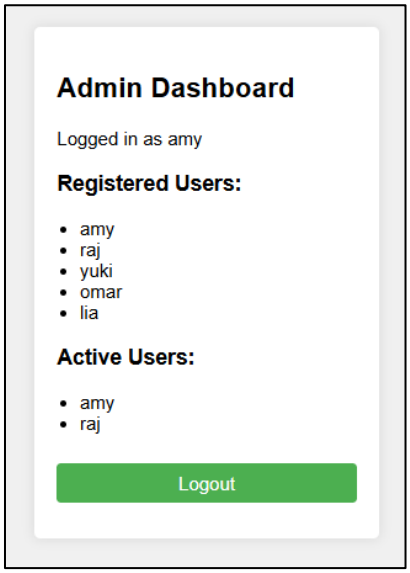
                label(for="password") Password
                input(type="password", name="password", id="password", required)

                button(type="submit") Login
```

The **admin.pug** is a little more interesting in that it takes a **username** as well as a list of **activeUsers** and **registeredUsers** as incoming parameters. It then displays the active/registered users. In the code, it handles the case where there are no active users … but this should never be the case in our code, since the admin user has to be logged on to see this page. The form at the end is a simple **Logout** button:

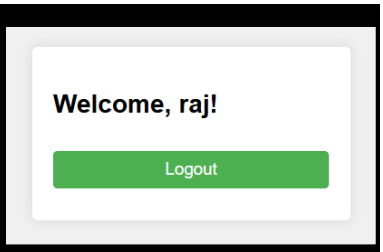
```
doctype html
html
    head
        title Admin Dashboard
        link(rel="stylesheet", href="/style.css")
    body
        .container
            h2 Admin Dashboard
            p Logged in as #{username}

            h3 Registered Users:
            ul
                each u in registeredUsers
                    li= u
            h3 Active Users:
            ul
                each u in activeUsers
                    li= u
            form(method="GET", action="/logout")
                button(type="submit") Logout
```



The **user.pug** is a simple welcome page for non-admin users which only takes a **username** as its sole incoming parameter. It also has a simple form at the end with **Logout** button:

```
doctype html
html
    head
        title Welcome
        link(rel="stylesheet", href="/style.css")
    body
        .container
            h2 Welcome, #{username}!
            form(method="GET", action="/logout")
                button(type="submit") Logout
```



Now, let's look at our server code a little bit at a time. Here is what we have at the top of our code … much of the usual stuff. But notice that we hardcode the users here:

```
const express = require('express');
const session = require('express-session'); // We will need npm install express-session
const app = express();
const PORT = 3000;
app.set('view engine', 'pug');
let users = {
    "amy":   { password: "1234",   admin: true },
    "raj":   { password: "india",  admin: false },
    "yuki":  { password: "sakura", admin: false },
    "omar":  { password: "desert", admin: true },
    "lia":   { password: "flor",   admin: false }
};
```

Next, we can set up some middleware before we deal with our routes:

```
// Use the express-session middleware
app.use(session({
    secret: 'f93^A7z!9uBv#2QpX0kLm@4sWrT8&dEe', // This can be anything
    //cookie: {maxAge:50000},                   // Expire in 50 seconds
    resave: false,                              // Do not save session every time
    saveUninitialized: true                     // Create session upon connection
}));

// Serve the static files in the public directory
app.use(express.static("public"));

// Middleware that lets us use req.body for incoming form query data
app.use(express.urlencoded({ extended: true }));
```

The first one lets us set options for our **express-session** middleware to put in our secret key (arbitrary) as well as a couple of others. Notice the commented-out cookie setting that sets the session timeout to be **50** seconds. We can put this code back in later when we are testing and see how it affects things 😉.

After that, we indicate that we want all static pages in the public directory to be served (only our stylesheet is in there in this example). Then we set up the middleware to parse the **req.body** for incoming query data … this makes our life easier by doing the tedious parsing work for us.

Now we are ready to discuss the routes. The root **/** route is our login page, which renders the **login.pug** page with no error parameter:

```
app.get("/", (req, res) => { res.render("login"); });
```

Next, we will handle a login **POST /login** request from the submitted form:

```
// Send POST request to /login route to our login function
app.post("/login", (req, res) => {
    const { username, password } = req.body;

    // Perform authentication
    if (!users[username] || users[username].password !== password) {
        res.render("login", { error: "Invalid credentials" });
        return;
    }
    req.session.username = username;

    // Decide whether to show the Admin Dashboard page or the Welcome page
    if (users[username].admin) {
        // Redirect to the admin page instead of just rendering, since we need to build
        // up the lists of active and registered users first.
        res.redirect("/admin");
    } else {
        // Redirect to welcome page
        res.redirect("/welcome");
    }
});
```

It first extracts the `username` and `password` from the form and then performs the authentication by checking if they match a user in the array. If not, it re-renders the same login page but this time with an error message 😉. If successfully authenticated, our code stores `username` for that session, which we will use to be able to detect who is currently logged in. Then the code checks if the user is an authorized admin user or not, then directs to either the **/admin** route or the **/welcome** route.

Now we can consider the **/logout** route. In this case, we simply destroy the session and then re-direct back to the root login page **/** as follows:

```
// Send GET request to /logout route to our logout function
app.get("/logout", (req, res) => {
    req.session.destroy(err => {
        res.redirect("/");
    });
});
```

Next, let's handle the **/welcome** route. If the user is not logged in, we re-direct to the login page, otherwise we render the **user.pug** page with the `username` passed in as a parameter:

```
// Handle /welcome route
app.get("/welcome", (req, res) => {
    // If not logged in, go back to the home page
    if (!req.session.username) {
        res.redirect("/");
        return;
    }
    // Otherwise render the user.pug page
    res.render("user", { username: req.session.username });
});
```

Finally, we will handle the **/admin** route. It will first call our own `auth` helper function to make sure that the user is logged in and has admin privileges, which looks like this:

```
// Our Authorization function
function auth(req, res, next) {
    // Check if there is a username property set for the session, and if they have admin rights
    if (!req.session.username || !users[req.session.username].admin) {
        res.status(401).send("Unauthorized");
        return;
    }
    next();
}
```

Notice the use of `next()` to continue the middleware chain.

Here is the code to handle the **/admin** route. Notice that it calls our **auth** function first. The code has a bit of work to do because it needs to gather the lists of registered and active users, which it passes as parameters to render the **admin.pug** page:

```
// Handle /admin route
app.get("/admin", auth, (req, res) => {
    // Get all registered users
    const registeredUsers = Object.keys(users);

    // Get all active users (i.e., anyone with a session + username)
    const activeUsers = [];
    const sessions = req.sessionStore.sessions;
    const sessionKeys = Object.keys(sessions);

    for (let i=0; i<sessionKeys.length; i++) {
        const sessionId = sessionKeys[i];
        const sessionData = JSON.parse(sessions[sessionId]);

        if (sessionData.username && !activeUsers.includes(sessionData.username)) {
            activeUsers.push(sessionData.username);
        }
    }

    res.render("admin", {
        username: req.session.username,
        registeredUsers,
        activeUsers
    });
});
```

Registered Users:

- amy
- raj
- yuki
- omar
- lia

Active Users:

- amy
- raj

Notice in the code that the `registeredUsers` array is just the names (i.e., keys) of the `users` that we hardcoded in our server code. The `activeUsers` array takes some more work because we first need to get the array of sessions from the one stored with the request. Then we grab the keys (which are the **session ID**s). We then loop through the sessions, parse the **JSON** data there and grab the `username` from that parsed session data. As long as that user is not already in the list, we add them to the list of `activeUsers`.

Of course, our code completes with the usual code to start up the server:

```
app.listen(PORT);
console.log("Server listening on port " + PORT);
```

At this point in the course, we have seen how authentication, authorization, and sessions work together to make web applications secure and personalized.

- **Authentication** lets the server know who a user is, usually by checking credentials like a username and password.
- **Authorization** then decides what that user is allowed to do, such as viewing an admin dashboard or a regular user page.
- **Sessions** and **cookies** are the glue that hold it all together. When a user logs in, the server creates a session and uses a cookie to remember that session across multiple requests, so users don't have to log in every time they click a link.

The result is that our websites can now have a kind of "memory," remembering who a user is across multiple requests, while clearly distinguishing between authorized and unauthorized parts of the site.