1. **Cookie mechanism**

**Step 1: Client > Signing up (HTTP POST)**

Before anything else, the user has to sign up. The client **posts a HTTP request** to the server containing his/her username and password.

**Step 2: Server > Handling sign up**

The server receives this request and **hashes** the password before storing the username and password in your database. This way, if someone gains access to your database they won't see your users' actual passwords.

**Step 3: Client > User login (HTTP POST)**

Now your user **logs in**. He/she provides their username/password and again, **HTTP POST** request to log in the server.

**Step 4: Server > Validating login**

The server looks up the username in the database, hashes the supplied login password, and compares it to the previously hashed password in the database. If it doesn't match, we may deny them access by [sending a **401 (Not authorized)** status code and ending the request](https://stackoverflow.com/questions/1959947/whats-an-appropriate-http-status-code-to-return-by-a-rest-api-service-for-a-val).

**Step 5: Server > Generating access token**

If everything checks out, we're going to create an access token, which uniquely identifies the user's session. Still in the server, we do two things with the **access token**:

1. **Store it in the database associated with that user**
2. Attach it to a response cookie to be returned to the client. Be sure to set an expiration date/time to limit the user's session

Henceforth, the cookies will be attached to every request (and response) made between the client and server.

**Another explanation**: server creates a session ID as user identifier, and store it in database

1. After the request is made, the server validate the user on the backend by querying in the database. If the request is valid, it will create a session by using the user information fetched from the database and store them, for each session a unique id called **session Id** is created ,by default session Id will be given to client through the Browser.
2. **Browser will submit this session Id** **on each subsequent requests**, the session ID is verified against the database, based on this session id website will identify the session belonging to which client and then give access the request.

**Step 6: Client > Making page requests**

Back on the client side, we are now logged in. Every time the client makes a request for a page that requires authorization (i.e. they need to be logged in), the server obtains the access token from the cookie and checks it against the one in the database associated with that user. If it checks out, access is granted.

This should get you started. Be sure to clear the cookies upon logout!

**SSL** secures interception during requests/responses, but an attacker might access your cookies at the client side (e.g. your browser). Theoretically, they could then pose as a logged in user until the cookie expires. I say “theoretically” because the implementation above doesn’t handle that. In the above implementation, the attacker will have access until the access token in your database is updated (i.e. next login).

you could along with the token/session key also **save the user's ip address** along with other identifying parameters such as user-agent, etc. if the request then comes with a valid cookie but from the wrong ip, browser, etc then you deny the request and redirect the user to the login page to authenticate again.

1. **Cookie size limitation**

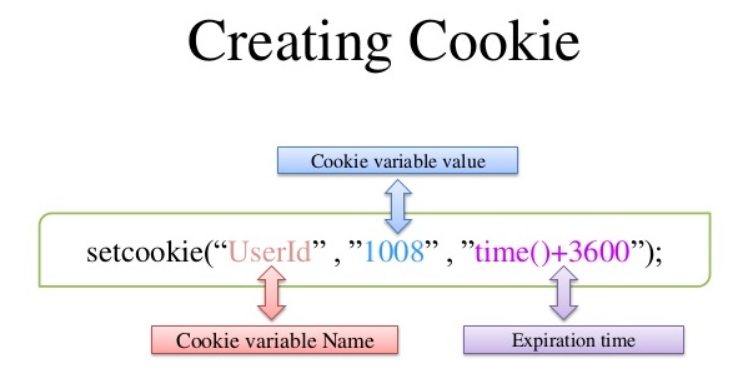
* **For each Domain, up to 4 KB /cookie, and 20 cookies**
* **For whole browser, up to 300 cookies.**

1. **Cookie Generation**

* *By server:* The Set-cookie response header from the server directs the client to set a cookie on that particular domain. **The implementation to actually create and store the cookie lies in the browser**. For subsequent requests to the same domain, the browser automatically sets the Cookie request header for each request, thereby letting the server have some state to an otherwise stateless HTTP protocol. The Domain and Path cookie attributes are used by the browser to determine which cookies are to be sent to a server. The server only receives name=value pairs, and nothing more.
* *By Client:* One can create a cookie on the browser using **javascript** document.cookie = cookiename=cookievalue. However, **if the server does not intend to respond to any random cookie a user creates, then such a cookie serves no purpose**.
* Cookies always belong to the client. There is no such thing as server side cookie.
* On the client side, by setting the HttpOnly attribute of the cookie, it is possible to prevent scripts ( mostly Javscript ) from reading your cookies , thereby acting as a defence mechanism against Cookie theft through XSS, but sends the cookie to the intended server only.
* Therefore, in most of the cases since cookies are used to **bring 'state'** ( memory of past user events ), **creating cookies on client side does not add much value**, unless one is aware of the cookies the server uses / responds to.

1. **Cookies contents**

When a developer creates a cookie, with the function **setcookie**, he must specify at least three arguments. These arguments are setcookie (**name**, **value**, **expiration**);



1. **Cookie Attribute**
2. **Name:** Specifies the name of the cookie
3. **Value:** Specifies the value of the cookie
4. **Secure:** Specifies whether or not the cookie should only be transmitted over a **secure HTTPS** connection. TRUE indicates that the cookie will only be set if a secure connection exists. Default is FALSE
5. **Domain:** Specifies the domain name of the cookie. To make the cookie available on all subdomains of example.com, set the domain to “example.com”. Setting it to www.example.com will make the cookie only available in the www subdomain
6. **Path:** Specifies the server path of the cookie. If set to “/”, the cookie will be available within the entire domain. If set to “/php/”, the cookie will only be available within the php directory and all sub-directories of php. The default value is the current directory that the cookie is being set in

* Explanation: Domain and Path

**The domain option**

* The next option is domain, which indicates the domain(s) for which the cookie should be sent. By default, domain is set to the host name of the page setting the cookie, so the cookie value is sent whenever a request is made to the same host name.Â  For example, the default domain for a cookie set on this site would be [www.nczonline.net](http://www.nczonline.net/). The domain option is used to widen the number of domains for which the cookie value will be sent. Sample:
* Set-Cookie: name=Nicholas; domain=nczonline.net
* Consider the case of a large network such as [Yahoo!](http://www.yahoo.com/) that has many sites in the form of *name*.[yahoo.com](http://yahoo.com/) (e.g., [my.yahoo.com](http://my.yahoo.com/), [finance.yahoo.com](http://finance.yahoo.com/), etc.). A single cookie value can be set for all of these sites by setting the domain option to simply [yahoo.com](http://yahoo.com/). The browser performs a tail comparison of this value and the host name to which a request is sent (meaning it starts the comparison from the end of the string) and sends the corresponding Cookie header when there’s a match.
* The value set for the domain option must be part of the host name that is sending the Set-Cookie header. I couldn’t, for example, set a cookie on [google.com](http://www.google.com/) because that would introduce a security issue. Invalid domain options are simply ignored.

**The path option**

* Another way to control when the Cookie header will be sent is to specify the path option. Similar to the domain option, path indicates a URL path that must exist in the requested resource before sending the Cookie header. This comparison is done by comparing the option value character-by-character against the start of the request URL. If the characters match, then the Cookie header is sent. Sample:
* Set-Cookie: name=Nicholas; path=/blog
* In this example, the path option would match /blog, /blogroll, etc.; anything that begins with /blog is valid. Note that this comparison is only done once the domain option has been verified. The default value for the path option is the path of the URL that sent the Set-Cookie header.

1. **HTTPOnly:** If set to TRUE the cookie will be accessible only through the HTTP protocol (the cookie will not be accessible by scripting languages). This setting can help to reduce identity theft through XSS attacks. Default is FALSE
2. **Expires:** Specifies when the cookie expires. The value: time ()+86400\*30, will set the cookie to expire in 30 days. If **this parameter is omitted** or **set to 0**, the cookie will expire at the end of the session (when the browser closes). Default is 0

This HTTP header supports different parameters:

* Expires
* **Max-Age ( seems not to be used any more, due to browser compatibility.)**
* Domain
* Path
* Secure
* HttpOnly

The specification says:

If a cookie has both the Max-Age and the Expires attribute, the Max-  
Age attribute has precedence and controls the expiration date of the cookie. **If a cookie has neither the Max-Age nor the Expires attribute, the user agent will retain the cookie until "the current session is over"** (as defined by the user agent).

To change the "expiry" of a cookie from the server, a Set-Header must be sent again with:

* the same cookie name
* the same Domain parameter value
* the same Path parameter value
* a different Expires or Max-Age attribute

But note that if the HttpOnly parameter is not set (or not supported by the browser), the cookie value and parameters can be modified by JavaScript.  
see: [JavaScript - Cookies](http://www.quirksmode.org/js/cookies.html)

So if some JS code change a cookie value this way:

1. document.cookie = "myCookie=anotherValue";

instead of

1. document.cookie = "myCookie=anotherValue; Expires=Wed, 13 Jan 2021 22:23:01 GMT";

The Expires being omitted, the cookie should switch from its "persistent" status to the "session" one.

1. **Cookie Types**

There are 2 types of cookies:

1. **session cookies**: one without expires property being set.
2. **persistent cookies**: one with expires property set

**If you do not set the cookie's expiration**, the cookie is created but it is not stored on the user's hard disk. Instead, the cookie is created in browser’s memory and is maintained as part of the user's session information. When the user closes the browser, the cookie is discarded.

In case of **persistent cookies**, even expired cookie will not be deleted immediately on expiration but only when a user visits the site that wrote the cookies. So, if you use persistent cookies and **user closed his browser without logoff**, the cookie will exist until user visits the site again. If at the time of the next visit, the expiration will still not happen then user will be logged on automatically.

1. **Session Cookie**

This type of cookies dies when the browser is closed because they are stored in the browser’s memory. They’re used for e-commerce websites so the user can continue browsing without losing what he put in his cart. If the user visits the website again after closing the browser these cookies will not be available. It is safer because no developer other than the browser can access them.

1. **Persistent Cookie**

These cookies do not depend on the browser session because they are stored in a file of browser computer. If the user closes the browser and then access the website again then these cookies will still be available. The lifetime of these cookies are specified in cookies itself (as expiration time). They are less secure.

1. **Third Party Cookie**

A cookie set by a domain name that is not the domain name that appears in the browser address bar these cookies is mainly used for tracking user browsing patterns and/or finding the Advertisement recommendations for the user.

1. **Secure Cookie**

A secure cookie can only be transmitted over an encrypted connection.  A cookie is made secure by adding the secure flag to the cookie. Browsers which support the secure flag will only send cookies with the secure flag when the request is going to an HTTPS page.

1. **HTTP Only Cookie**

It informs the browser that this particular cookie should only be accessed by the server. Any attempt to access the cookie from the client script is strictly prohibited. This is important security protection for session cookies.

1. **Session ID**
2. PHP code generates a unique identification in the form of hash for that specific session which is a random string of 32 hexadecimal numbers such as 5f7dok65iif989fwrmn88er47gk834 is known as **PHPsessionID**.
3. A **session ID** **or token** is a **unique number** which is used to **identify a user** that has logged into a website. **The session ID is stored inside the server**, it is assigned to a specific user for the duration of that user’s visit (**session**). The **session ID** can be stored as a cookie, form field, or URL.

* In the given picture we can clearly see there are three components inside it: **HTTP Client**, **HTTP server** and **Database** (holding session ID).
* **Step1:** the client sends a request to the server via POST or GET.
* **Step2:** session Id created on the web server. **Server saves session ID into the database** and using set-cookie function send session ID to the client browser as a response.
* **Step3:**a cookie with session ID stored on client browser is sent back to the server where server matches it from the database and sends a response as HTTP 200 OK.

1. **Login/logout**

* Can I just log out by wiping cookies instead of hitting logout?

Yes, since the web application uses cookies to uniquely identify you, deleting cookies will log you out.

* What are the issues of just wiping cookies versus clicking the logout button?

The **logout button** serves a special purpose in that it sends a request to delete the session and **returns a response to delete the cookie in your browser** as well. If you don't send a request to delete the session, **then the session still remains alive server side**. Sessions do have a maximum lifetime and will be deleted by the garbage collector eventually.

If the logout endpoint is executed in the context of a browser session that previously had an authenticated user session, then **that session is invalidated** and **the user will no longer be considered authenticated**. However, achieving this requirement (logging out a user) does not strictly require that all cookies be removed.

In other words, **calling logout will invalidate the user session if one existed before**, but how that is accomplished is mostly an implementation detail. Can you clarify if you strictly require that no cookies exists after the logout is performed or are you simply experiencing issues with calling logout?

1. **Session Cookie vs Persistent Cookie**

**session cookies**: expires property is NOT set.

**persistent cookies**: expires property is set

 A typical web authentication workflow looks like this:

1. User provides their credentials.
2. Server validates credentials.
3. If credentials are valid
   * Server generates a token.
   * Server keeps this token.
   * Server responds to the login with this token.
4. Browser stores token.
5. Browser makes requests with token.
6. Server validates token and responds accordingly.

Normally, this token is stored in a cookie. The presence and validity of the token in a request lets the server know if the client making the request is authenticated. No token, no entry, which is effectively the same as not being logged in.

1. **Remove a cookie**

**Setting Expires to be in the past** is the standard, spec-compliant way of **deleting a cookie**, and user agents are required by spec to respect it.

Just set the cookie on exactly the same name, path and domain, but with an Expires value in the past.

1. Example 1

* For authentication, we are using AWS Cognito. Our project contains an API server and a web server. On the client-side, when the user login to the application, we send the username & password to the cognito instance, which returns a **JWT access token**. We set the **access token** in the **cookies** and redirect the user to the homepage. **On the http server side**, on each request for a private page, we check the cookie and validate the JWT access token. Similarly on the API side, for each request, we check the cookie and validate the access token.
* When a user log out from the application, we remove the cookie and signout from cognito.
* The problem occurs when a user presses the back button of the browser after signing out. It seems like when the user presses the back button, the page is restored from the cache along with the cookies. And since the JWT access token is still valid, after user presses the back button they can still make the api calls and navigate the application
* Should make the JWT Access Token expire on the server-side on logout.
* A browser cached page doesn't restore cookies. Check to see if your cookie is marked httponly. If so, you won't be able to clear it with this client-side js. In this case, you'll need the log out HTTP request to respond with a Set-Cookie header that clears the cookie.

1. **Cookie flows**

* A server can send a cookie using the **Set-Cookie** header:

|  |  |
| --- | --- |
| 1  2  3 | HTTP/1.1 200 Ok  **Set-Cookie: access\_token=123** |

* A client will then store this data and send it in subsequent requests through the **Cookie** header:

|  |  |
| --- | --- |
| 1  2  3  4 | GET / HTTP/1.1  Host: example.com  **Cookie: access\_token=1234** |

* Note that servers can send multiple cookies at once:

|  |  |
| --- | --- |
| 1  2  3  4 | HTTP/1.1 200 Ok  **Set-Cookie:** access\_token=1234  **Set-Cookie:** user\_id=10 |

* and clients can do the same in their request:

|  |  |
| --- | --- |
| 1  2  3  4 | GET / HTTP/1.1  Host: example.com  **Cookie:** **access\_token=1234; user\_id=10** |

in addition to the plain key and value, cookies can carry additional directives that limit their time-to-live and scope:

### **Expires**

Specifies when a cookie should expire, so that browsers do not store and transmit it indefinitely. A clear example is a session ID, which usually expires after some time. This directive is expressed as a date in the form of Date: <day-name>, <day> <month> <year> <hour>:<minute>:<second> GMT, like Date: Fri, 24 Aug 2018 04:33:00 GMT. Here’s a full example of a cookie that expires on the 1st of January 2018:

|  |  |
| --- | --- |
| 1 | access\_token=1234;Expires=Mon, 1st Jan 2018 00:00:00 GMT |

### Max-Age

Similar to the Expires directive, Max-Age specifies the number of seconds until the cookie should expire. A cookie that should last 1 hour would look like the following:

|  |  |
| --- | --- |
| 1 | access\_token=1234;Max-Age=3600 |

### Domain

This directive defines which hosts the cookie should be sent to. Remember, cookies generally contain sensitive data, so it’s important for browsers not to leak them to untrusted hosts. A cookie with the directive Domain=trusted.example.com will not be sent along with requests to any domain other than trusted.example.com, not even the root domain (example.com). Here’s a valid example of a cookie limited to a particular subdomain:

|  |  |
| --- | --- |
| 1 | access\_token=1234;Domain=trusted.example.com |

### Path

Similar to the Domain directive, but applies to the URL path (/some/path). This directive prevents a cookie from being shared with untrusted paths, such as in the following example:

|  |  |
| --- | --- |
| 1 | access\_token=1234;Path=/trusted/path |

## Cookie flow 2

* Note that servers can set multiple cookies at once:

HTTP/1.1 200 Ok

Set-Cookie: access\_token=1234

Set-Cookie: user\_id=10...

* and clients can store multiple cookies and send them in their request:

GET / HTTP/1.1

Host: example.com

Cookie: access\_token=1234; user\_id=10...

1. **Cookie in Balance-Loader**

**Cookies in load-balanced web applications**

As we saw earlier, server issuing cookie maintains a list of sessionID on server to identify its corresponding users. This list/map is typically maintained in **server’s memory/RAM**. This is a simple setup. **Most of our production web servers are load balanced**.

Now imagine **if we have an application with two web servers**:

1. User initially connects to first server and after valid credentials, server 1 issues the cookie.
2. But, on next request, user is connected to server 2. The browser would send the cookie issued by server 1 as the domain name is still same.
3. However, **server 2 has no idea about the received cookie as it has not issued this cookie.**



Handling cookies using Load Balancer

There are **multiple ways to address this issue**:

1. **Two servers always talk to each other**. **Whenever server 1 issues a cookie**, it also tells server 2 about this new cookie. **Server 2 would then store that in its memory**. Vice is versa is also applicable. When user logs out, similar thing happens. Cookie on other server is destroyed. This is known as **session trickling**.
2. First approach is rudimentary. Better approach to share cookie is via **shared Database** like MySQL. Concept is same. Only thing is, instead of communicating directly, servers co-ordinate via database.
3. Second approach is better but slower as every request would involve database call. To tackle this, high speed **in-memory databases like Redis** are used. Redis is NoSQL, Key-Value database which is perfect fit for this scenario.

Handling cookies using Redis like high speed in-memory database



1. In a different scenario, where we have micro-services, our API servers or web servers are hidden behind a firewall and there is a separate service that is responsible for issuing and validating cookies. Sometimes application **gateways/load balancers are responsible for cookies** and web servers are hidden from direct access.
2. **HTTP remember-me feature in log in**

**Generally,**

The **"save password**" part comes from the browser's password manager whenever it sees an <input type="password"> that looks like it really is asking for a password. You can use the autocomplete attribute to suppress this in most browsers:

<input type="password" name="password" autocomplete="off">

This won't validate but that usually doesn't matter.

The "**remember me**" part is completely separate from the browser's password manager. The "remember me" flag is the **server's business** and all it does is fiddle with the expiry date on the cookie that it sends back. The server will always send a cookie back (unless they're not using cookies for tracking sessions but that's rare and wouldn't need a "remember me" anyway) with something inside it to identify the client user.

If you **check "remember me"** then you're telling the server that you want a **persistent session**. To achieve this, the server will include an **expiry date** with the cookie and that expiry date will be some time in the future. When the date arrives, the browser will expire and delete the cookie; without the cookie, the server won't know who you are anymore and you'll have to login again.

If you **don't check "remember me"** then you'll get a **session cookie**. **Session cookies don't have expiry dates** on them so automatically expire when the browser exits. Session cookies are useful for shared machines.

Executive summary:

* "Save password" is from the browser's password manager.
* "Remember me" is about the login cookie's expiry time.

The session id is stored in the cookie. **while the password, or the hash of it is not stored**. A session is created on the server side whenever you log in. If you logged in with "Remember Me" checked**, the server passes a cookie with the session id (or encrypted session id, or something that uniquely identifies the user session) and this cookie is saved on the client side**.  
When you login for the next time, the server checks whether there is a cookie with the session, if it is there (and the session has not been killed/expired - see point 2 below) then the server identifies you as "Veera" and lets you in the site.

# How "Remember me" works in Duo multi-factor authentication

The Remember me feature of Duo Mobile works for most people most of the time. However, it is not 100% reliable because the feature relies on web cookies of a particular type, and the web browser you are using must accept this type of cookie. These are common situations in which you may need to authenticate with Duo MFA each time you log in.

### Web browser issues

* Allowing cookies  
  Most web browsers have security and privacy settings that allow you to accept or reject cookies entirely. You must allow cookies to be set by \*.northwestern.edu and \*.duosecurity.com.
* Tracking or 3rd party cookies  
  Cookies are often sent back only to the site that initially gave you the cookie. But many web sites also use what are called *3rd party* cookies. These are also known as *tracking* or *cross-site* cookies, because they allow one web site (e.g., www.cnn.com), to track your visits to another site (e.g., www.facebook.com). Duo's Remember me cookie, although it is not used for tracking or advertising purposes, is indistinguishable from these cookies. More and more browsers are not allowing these cookies by default, though most let you adjust these settings.
* Multiple browsers  
  If you authenticate with Duo MFA using one browser (e.g., Chrome), then later login via another browser (e.g., Firefox), you will be prompted to authenticate with Duo MFA again. Web browsers do not share cookies with each other.
* Clearing cache/cookies  
  If you delete Duo's Remember me cookie by manually clearing your browser's cookies/cache, you will have to authenticate with Duo MFA again the next time you log in. Some web browsers can also be configured to clear cookies and/or cache each time they are launched.
* Private or Incognito mode  
  Most browsers have a private or incognito mode. When you use this, cookies are not saved, website addresses aren’t cached, and several other features are modified. You will always have to authenticate with Duo MFA when using incognito/private mode.

# Secure Remember Me for Login using PHP Session and Cookies

Last modified on September 11th, 2020.

Login Script with ‘Remember Me’ feature will allow the user to preserve their logged in status. When the user checks the **Remember Me** option, then the logged in status is serialized in the [PHP session or cookies](https://phppot.com/php/session-vs-cookies/) like storages. While writing user login data in the session or cookie we need to be aware of the security breaches which might compromise the application’s authentication system. Plain passwords should not be stored in the user’s cookie, this will allow hacking the application.

This example will help you [to build a persistent](https://phppot.com/php/how-to-build-a-persistent-shopping-cart-in-php/) authentication system for your PHP web application. When the user attempts to log in with the application, the entered login credentials are verified with the database. If a match is found, the PHP session and the cookies are used to preserve user logged-in state before redirecting the user to the dashboard. On successful login, the unique member id from the member database is stored in a session. Then, the cookies are set to keep the login name and the password for a specified expiration period.  Instead of storing the users’ plain password, random password and token are generated and stored in the cookie to avoid hacking.

When the user accessing the application pages, the [existing logged in session](https://phppot.com/php/php-login-script-with-session/) is checked to redirect the user to access the requested page. If the session is empty, then the code will check the logged-in with the cookies. If both the session and the cookies are not having any data about the remembered login, then the user will be redirected back to the login page. The authentication cookies are set with the expiration time of 1 month. The random password and tokens will be stored in the database with the expiration date and time. The cookie-based logged in state validation is done by testing cookie availability and expiration stored in the database.

1. **Cookie-based Authentication**

**What is in the cookie, format of auth. Cookie.**

## Examples of cookies

Cookies are most commonly used to track website activity. When you visit some sites, the server gives you a cookie that acts as your identification card. Upon each return visit to that site, your browser passes that cookie back to the server. In this way, a web server can gather information about which web pages are used the most, and which pages are gathering the most repeat hits.

Cookies are also used for online shopping. Online stores often use cookies that record any personal information you enter, as well as any items in your electronic shopping cart, so that you don't need to re-enter this information each time you visit the site.

Servers can use cookies to provide personalized web pages. When you select preferences at a site that uses this option, the server places the information in a cookie. When you return, the server uses the information in the cookie to create a customized page for you.

## Cookies are scoped by path: the Path attribute

Consider this backend which sets a new cookie for its frontend when visiting <http://127.0.0.1:5000/>. On the other two routes instead we print the request's cookies:

from flask import Flask, make\_response, request

app = Flask(\_\_name\_\_)

@app.route("/", methods=["GET"])

def index():

response = make\_response("Here, take some cookie!")

response.set\_cookie(key="id", value="3db4adj3d", path="/about/")

return response

@app.route("/about/", methods=["GET"])

def about():

print(request.cookies)

return "Hello world!"

@app.route("/contact/", methods=["GET"])

def contact():

print(request.cookies)

return "Hello world!"

To run the app:

FLASK\_ENV=development FLASK\_APP=flask\_app.py flask run

In another terminal, if we make connection with the root route we can see the cookie in Set-Cookie:

curl -I http://127.0.0.1:5000/ --cookie-jar cookies

HTTP/1.0 200 OK

Content-Type: text/html; charset=utf-8

Content-Length: 23

Set-Cookie: id=3db4adj3d; Path=/about/

Server: Werkzeug/1.0.1 Python/3.8.3

Date: Wed, 27 May 2020 09:21:37 GMT

Notice how the cookies has a Path attribute:

Set-Cookie: id=3db4adj3d; Path=/about/

Let's now visit the /about/ route by sending the cookie we saved in the first visit:

curl -I http://127.0.0.1:5000/about/ --cookie cookies

In the terminal where the Flask app is running you should see:

ImmutableMultiDict([('id', '3db4adj3d')])

127.0.0.1 - - [27/May/2020 11:27:55] "HEAD /about/ HTTP/1.1" 200 -

As expected the cookie goes back to the backend. Now try to visit the /contact/ route:

curl -I http://127.0.0.1:5000/contact/ --cookie cookies

This time in the terminal where the Flask app is running you should see:

ImmutableMultiDict([])

127.0.0.1 - - [27/May/2020 11:29:00] "HEAD /contact/ HTTP/1.1" 200 -

What that means? Cookies are scoped by path. **A cookie with a given** Path **attribute cannot be sent to another, unrelated path, even if both path live on the same domain**.

This is the first layer of **permissions** for cookies.

When Path is omitted during cookie creation, the browsers defaults to **/**.

## Cookies are scoped by domain: the Domain attribute

The value for the Domain attribute of a cookie controls **whether the browser should accept it or not** and **where the cookie goes back**.

Let's see some examples.

**NOTE**: the following URL are on free Heroku instances. Give it a second to spin up. Open up a browser's console before opening the links to see the result in the network tab.

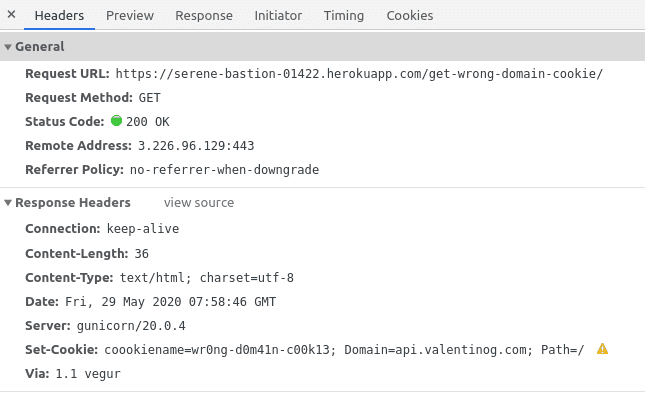
### Non matching host (wrong host)

Consider the following cookie set by <https://serene-bastion-01422.herokuapp.com/get-wrong-domain-cookie/>:

Set-Cookie: coookiename=wr0ng-d0m41n-c00k13; Domain=api.valentinog.com

Here the cookie originates from **serene-bastion-01422.herokuapp.com**, but the Domain attribute has **api.valentinog.com**.

There's no other choice for the browser to **reject this cookie**. Chrome for example gives a warning (Firefox does not):

[](https://www.valentinog.com/blog/static/0f0923a1ac35891f603506fa6ebb5a2f/af192/chrome-reject-wrong-domain-cookie.png)

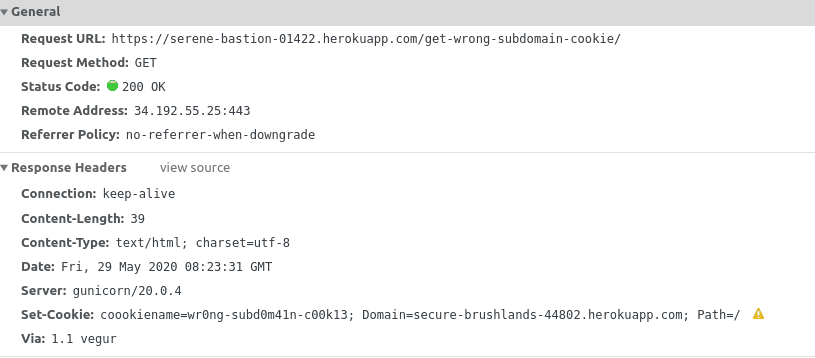
### Non matching host (subdomain)

Consider the following cookie set by <https://serene-bastion-01422.herokuapp.com/get-wrong-subdomain-cookie/>:

Set-Cookie: coookiename=wr0ng-subd0m41n-c00k13; Domain=secure-brushlands-44802.herokuapp.com

Here the cookie originates from **serene-bastion-01422.herokuapp.com**, but the Domain attribute is **secure-brushlands-44802.herokuapp.com**.

They are on the same domain, but the subdomain is different. Again, the browser rejects this cookie as well:

[](https://www.valentinog.com/blog/static/e0417b91d463c255f06fbbc9a39f47b7/ef916/chrome-reject-wrong-subdomain-cookie.png)

### Matching host (whole domain)

Consider now the following cookie set by visiting <https://www.valentinog.com/get-domain-cookie.html>:

set-cookie: cookiename=d0m41n-c00k13; Domain=valentinog.com

This cookie is set at the web server level with [Nginx add\_header](http://nginx.org/en/docs/http/ngx_http_headers_module.html):

add\_header Set-Cookie "cookiename=d0m41n-c00k13; Domain=valentinog.com";

I **used Nginx here to show you there are various ways to set a cookie**. The fact that a cookie is set by a web server or by the application's code **doesn't matter much for the browser**.

What matters is the domain the cookie is coming from.

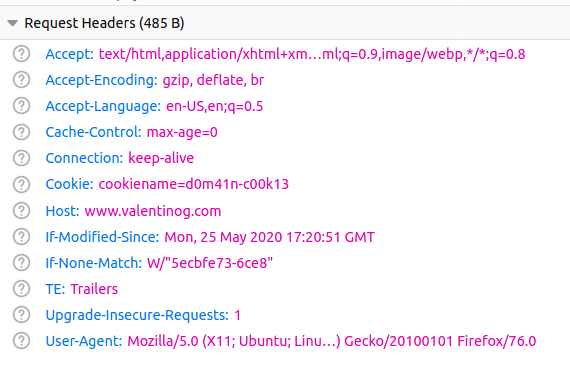
Here the browser will **happily accept the cookie** because the host in Domain **includes the host from which the cookie came**.

In other words, **valentinog.com** includes the **subdomain www.valentinog.com**.

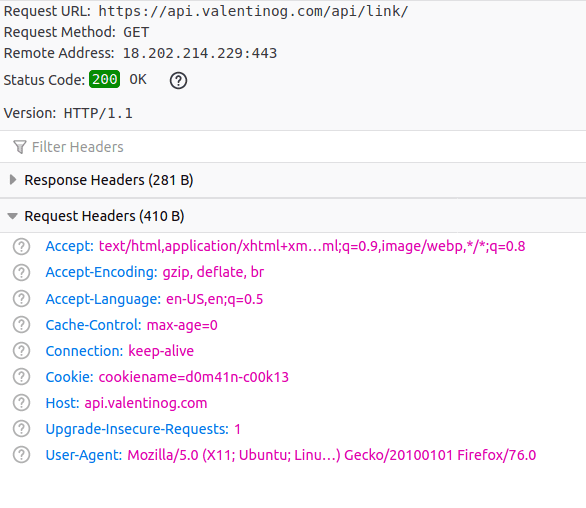
Also, the **cookie travels back with any new request against valentinog.com**, as well as **any request to subdomains on valentinog.com**.

Here's a request to the www subdomain with the cookie attached:

Here's a request to the **subdomain:** www with the cookie attached:

[](https://www.valentinog.com/blog/static/04ed9576c4bdf1207e531eaa6c76618e/432e7/cookie-sent-back-domain-matches.png)

Here's a request to another **subdomain**: **api** with the cookie automatically attached:

[](https://www.valentinog.com/blog/static/759672f4eb6e4517cf8dc091521f889d/a76f4/cookie-subdomain-sent-back-domain-matches.png)

### Cookies and the Public Suffix List

Now consider the following cookie set by <https://serene-bastion-01422.herokuapp.com/get-domain-cookie/>:

Set-Cookie: coookiename=d0m41n-c00k13; Domain=herokuapp.com

Here the cookie comes from **serene-bastion-01422.herokuapp.com**, and the Domain attribute is **herokuapp.com**. What should the browser do here?

You **might think that serene-bastion-01422.herokuapp.com is included in the domain herokuapp.com**, so the browser should accept the cookie.

Instead, **it rejects the cookie** because it comes from a domain included in the **Public Suffix List**.

The **Public Suffix List** is a list maintained by Mozilla, used by all browsers to restrict who can set cookies on behalf of other domains.

Resources:

* [Public suffix list](https://publicsuffix.org/)
* [Cookies and the Public Suffix List](https://devcenter.heroku.com/articles/cookies-and-herokuapp-com)

### **Matching host (subdomain)**

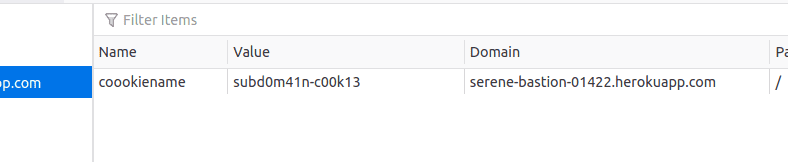
Consider now the following cookie set by <https://serene-bastion-01422.herokuapp.com/get-subdomain-cookie/>:

Set-Cookie: coookiename=subd0m41n-c00k13

When Domain**is omitted during cookie creation**, the browsers defaults to the originating host in the address bar, in this case my code does:

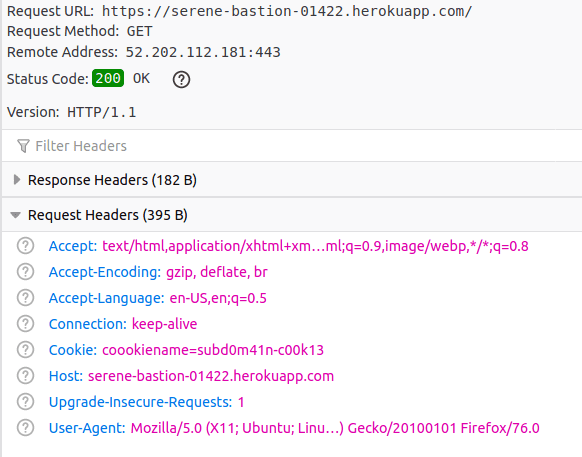
response.set\_cookie(key="coookiename", value="subd0m41n-c00k13")

When the cookie lands in the browser's cookie storage we see the Domain applied:

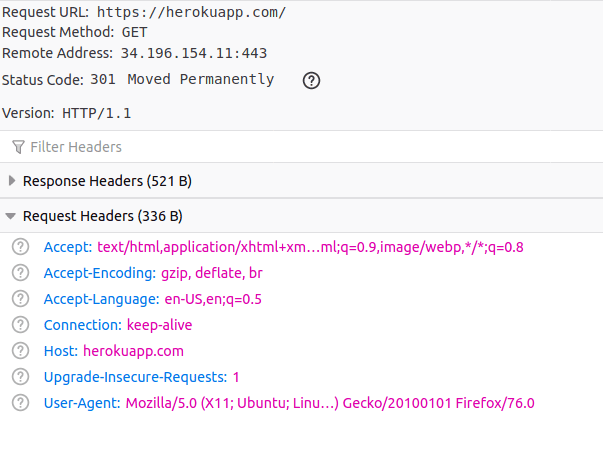
[](https://www.valentinog.com/blog/static/7545527e090e86cf1793a5cef2daeae9/ea7fb/default-domain-cookie.png)

So we have this cookie from serene-bastion-01422.herokuapp.com. **Where this cookie should be sent now?**.

If you visit <https://serene-bastion-01422.herokuapp.com/> the cookie goes with the request:

[](https://www.valentinog.com/blog/static/12375a46d81a8f8384786c7ac302d393/7c1cd/cookie-sent-subdomain.png)

**But, if you visit herokuapp.com**, **the cookie does not leave the browser at all**:

[](https://www.valentinog.com/blog/static/d9cb09c285639425f6d5a8e2434c5b40/9128f/cookie-subdomain-to-domain-not-sent.png)

(It doesn't matter that herokuapp.com later redirects to heroku.com).

To recap, **the browser uses the following heuristics to decide what to do with cookies** (by sender host here I mean the actual URL you visit):

* **Reject the cookie** altogether if either the domain or the subdomain in Domain don't match the sender host
* **Reject the cookie** if the value of Domain is included in the Public suffix list
* **Accept the cookie** if the domain or the subdomain in Domain matches the sender host

Once the browsers accepts the cookie, and it's about to **make a request** it says:

* **Send it back the cookie** if the request host matches exactly the value I saw in Domain
* **Send it back the cookie** if the request host is a subdomain matching exactly the value I saw in Domain
* **Send it back the cookie** if the request host is a subdomain like sub.example.dev included in a Domain like example.dev
* **Don't send it back the cookie** if the request host is a main domain like example.dev and Domain was sub.example.dev

**Takeaway**: Domain is the second layer of **permissions** for cookies, alongside with the Path attribute.

## Cookies and authentication

Authentication is one of the most challenging tasks in web development. There seems to be so much confusion around this topic, as token based authentication with JWT seems to supersede "old", solid patterns like **session based authentication**.

Let's see what role cookies play here.

### Session based authentication

**Authentication** is one of the most common use case for cookies.

When you visit a website that requests authentication, on credential submit (through a form for example) the backend sends under the hood a Set-Cookie header to the frontend.

A typical session cookie looks like the following:

Set-Cookie: sessionid=sty1z3kz11mpqxjv648mqwlx4ginpt6c; expires=Tue, 09 Jun 2020 15:46:52 GMT; HttpOnly; Max-Age=1209600; Path=/; SameSite=Lax

In this Set-Cookie header the server may include a cookie named **session, session id, or similar**.

This is the only identifier that the browser can see in the clear. Any time the **authenticated user requests a new page to the backend, the browser sends back the session cookie**.

At this point the backend pairs the session id with the session stored on a storage behind the scenes to properly identify the user.

Session based authentication is know as **stateful** because the backend has to keep track of sessions for each user. The storage for these sessions might be:

* a database
* a key/value store like Redis
* the filesystem

Of these three session storages, Redis or the like should be preferred over database or filesystem.

Note that session based authentication has **nothing to do with the browser's Session Storage**.

It's called **session based** only because the relevant data for user identification lives in the backend's session storage, which is not the same thing as a browser's Session Storage.

### When to use session based authentication?

Use it **whenever you can**. **Session based authentication is one of the simplest, secure, and straightforward form of authentication for websites**. It's available by default on all the most popular web frameworks like Django.

But, its **stateful** nature is also its main drawback, especially when a website is served by a load balancer. In this case, techniques like **sticky sessions**, or **storing sessions on a centralized Redis storage** can help.

1. **Cookie Domain and Path**

Back in the days of the good old web 1.0 cookies were implemented to preserve and persist state information on the side of the User Agent (mostly a web browser). The nitty gritty details are all written down in [RFC 6265](https://tools.ietf.org/html/rfc6265/), but it all boils down to cookies being key-value pairs that can be written to the User Agent using the Set-Cookie http header in the response. Besides the cookie name (the key) and its value, other options can additionally be set to control when a cookie should expire (expires or the newer max-age), if it may be sent via http or only via https (**Secure**), and if JavaScript in the browser should be able to read and write the cookie value or if the cookie should **be hidden from JavaScript** and only be sent to the server (**HttpOnly**).

If a cookie is set and not yet expired, the User Agent sends the cookie along to the server with **every request that matches the domain and path values** that can also be set on a cookie. What do they do?

Let's assume there is a website on www.example.com and it allows you to log in to customize your experience. Your log in session is stored in a session cookie. The website also offers a store at shop.example.com which uses the same login mechanism. With the default behavior, the cookie would be set to www.example.com and not be sent to shop.example.com, so you would have to log in at both subdomains explicitly. When the website sets the cookie using the **domain=example.com** parameter, the cookie will be sent to all **subdomains of example.com**, and so the user only needs to **login once**. You can only use the domain option to **narrow or broaden** the scope of the cookie on your own domain: The value set of the domain option must be part of the host name that is sending the Set-Cookie header. So you can't set a cookie for thinktecture.com as that would introduce a security issue.

The path allows you to further **narrow down** the places where a cookie will be sent to. If, for example, the website sets the cookie and states domain=example.com; path=/blog, then the cookie will only be sent to the server **if the request starts with /blog**, so it would be sent to www.example.com/blog and shop.example.com/blogroll, but not to example.com/blo.

* Cookies can only be set for the current domain or its subdomains as you have already found out (otherwise, anyone could replace anyone else's cookie; chaos would ensue). This is enforced by the browser: if you attempt to set cookies for another domain from the server side (using an HTTP header) the browser will ignore the cookie. If you attempt to do the same from the client side (using Javascript), the [same origin policy](http://en.wikipedia.org/wiki/Same_origin_policy) will prevent you from doing so.
* Therefore, www.evilsite.com can set a cookie for its own domain with Javascript, but that's not an issue: it could already do that using an HTTP header. There's no new attack vector here.
* Cookie Domain Match
* Cookie with Domain=.example.com **will** be available for *www.example.com*
* Cookie with Domain=.example.com **will** be available for *example.com*
* Cookie with Domain=**example.com** will be converted to **.example.com** and thus **will** also be available for *www.example.com*
* Cookie with Domain=example.com will **not** be available for *anotherexample.com*
* *www.example.com* **will** be able to set cookie for *example.com*
* *www.example.com* will **not** be able to set cookie for *www2.example.com*
* *www.example.com* will **not** be able to set cookie for *.com*

.www.example.com will only be accessible for other domains below that domain (e.g. **foo.**www.example.com or **bar.**www.example.com)

**.**example.com can also be accessed by any other domain below example.com (e.g. foo.example.com or bar.example.com, and example.com)

1. [**How client know which cookies should be send to the server**](https://stackoverflow.com/questions/49707593/how-client-know-which-cookies-should-be-send-to-the-server)

* Each domain has its own cookies in a cookie jar / cookie store. Whenever a request is made by the browser to the server, **all cookies in the store for that domain or subdomain** will be **sent to the server**.
* The cookies are saved in the browser, the browser maintains a **cookie file somewhere for each domain**. Once a request to that domain is made the entire content of the cookie file is sent as cookies to the server. These cookies may have been generated by the server or by a **javascript call**. The cookies are not necessarily created by the previous response from the server.

To directly answer the question: the client sends back **all** cookie data it has for that **domain** and **path**.

If you do not see any Set-Cookie header in any HTTP response, it may be because the cookie has been issued by server and stored on your computer **before you started looking** at Dev Tools or Fiddler. It could have been up to a few days, weeks, even months ago.

In Firefox, if you navigate to about:preferences#privacy and click Manage Data, you can see which domains already have cookies issued and stored on your computer. The Storage tab in Firefox Dev Tools can show you details of all cookies. The expiry of the cookie is determined by the server, using the Expires or Max-Age directive in the Set-Cookie header.

**How a cookie first ended up on the client computer:**

1. Client makes its first ever HTTP request to server, e.g. GET www.example.com
2. Server creates a cookie and sends it back in the HTTP response, e.g. the response headers contains a line: Set-Cookie: sessionID=1234567; path=/; Max-Age=31536000
3. The client receives HTTP response and stores the cookie in the "jar" for domain www.example.com.

**How server uses cookies to identify the client**

In subsequent HTTP requests to domain www.example.com, the client sends back all cookies in the jar that matches the path or sub-path. For example, the client wishes to issue a request GET www.example.com/about, sees that the URL is a sub-path of / in domain www.example.com, so it sends the cookie as a line in the HTTP request header, i.e. Cookie: sessionID=1234567.

The server sees the cookie and knows exactly which client made this request.

* **secure cookies vs insecure cookies**
* Secure cookies will be sent only on connections that are made over SSL(https protocol). Normal cookies will be sent on both http and https protocols.
* **session cookies vs. persistent cookies**
* session cookies - These cookies persist as long as the browser session is open. This means that Once you have cleared cache or closed the browser they get lost.
* persistent cookies - These will persist even if the browser is closed and opened again unless you have set the browser to clear cookies on exit in which case they will behave just like session cookies.
* **First party cookies vs. Third party cookies**.
* First party cookies - generated by the domain currently open as main document - this means they have same domain as the one displayed in your browser.
* Third party cookies - generated by a different domain then currently opened by the browser(in the addressbar) but which are **managed inside an iframe or various resource calls** like css, **script**, media(images, videos or other embedded media)

**Cookie Domain**

## Understanding Cookie Domain and sub-domain

As discussed earlier, cookies have a Domain directive which indicates one or more domains for which the cookie should be sent. By default, domain is set to the host name of the page setting the cookie.

Imagine a website [https://google.com](https://google.com/) setting a following header:

Set-Cookie: id=1234;

So, browser will send the cookie with every subsequent request to [https://google.com](https://google.com/) domain. Since, **default value of the domain is used, browser will not send this cookie to any sub-domain of google.com.** Thus cookie will not be sent by browser for requests to following domains:

* [https://mail.google.com](https://mail.google.com/)
* [https://drive.google.com](https://drive.google.com/)
* [https://files.drive.google.com](https://files.drive.google.com/)

However if [https://google.com](https://google.com/) sends following header:

Set-Cookie: id=1234; Domain=google.com

Since, server has explicitly specified domain value, browser would send cookie for any sub-domain of [https://\*.google.com](https://blog.webf.zone/*.google.com). As explained by [Nicolas Zakas](https://www.nczonline.net/blog/2009/05/05/http-cookies-explained/), **browser performs a tail comparison of this value and the host name to which a request is sent** (meaning it starts the comparison from the end of the string) and sends the corresponding Cookie header when there’s a match. We can also conclude that —

Parent domain can set cookies for Sub-domain and Sub-domain can also set cookies for Parent domain.

The 2 domains mydomain.com and subdomain.mydomain.com can only share cookies if the domain is explicitly named in the Set-Cookie header. Otherwise, the scope of the cookie is restricted to the request host. (This is referred to as a "host-only cookie". See [What is a host only cookie?](https://stackoverflow.com/questions/12387338/what-is-a-host-only-cookie))

For instance, if you sent the following header from subdomain.mydomain.com, then the cookie won't be sent for requests to mydomain.com:

Set-Cookie: name=value

However if you use the following, it will be usable on both domains:

Set-Cookie: name=value; domain=mydomain.com

This cookie will be sent for any subdomain of mydomain.com, including nested subdomains like subsub.subdomain.mydomain.com.

In [RFC 2109](http://tools.ietf.org/html/rfc2109), a domain without a leading dot meant that it could not be used on subdomains, and only a leading dot (.mydomain.com) would allow it to be used across multiple subdomains (but not the top-level domain, so what you ask was not possible in the older spec).

However, all modern browsers respect the newer specification [RFC 6265](http://tools.ietf.org/html/rfc6265), and will ignore any leading dot, meaning you can use the cookie on subdomains as well as the top-level domain.

In summary, if you set a cookie like the second example above from mydomain.com, it would be accessible by subdomain.mydomain.com, and vice versa. This can also be used to allow sub1.mydomain.com and sub2.mydomain.com to share cookies.

**CORS** - cross domain calls via xhttp ajax calls - this case arises when you create a domain requests resources from another domain via xhttp(ajax calls). In this case the browser makes a preflight check to see if the receiving domain accepts queries from the origin domain(origin headers are sent to the domain to check the cross domain policy). The server must necessarily respond with a valid options header and the server may allow identity data which is short for cookie data. If the remote domain has answered correctly with an "Access-Control-Allow-Origin" header that allows your domain or "\*" then you are allowed to send cookies via this request. And these will behave just like normal calls.

## what data do Cookies collect

The most common attributes include:

* A randomly generated and unique number that is used to recognize your computer. This makes web applications such as online shops and online banking possible: The website "remembers" you accordingly, so that, for example, the previously selected goods do not disappear from your shopping basket every time you call up a new page.
* The [domain name](https://www.ionos.com/digitalguide/domains/domain-tips/what-is-a-domain/), i.e. the website to which the cookie refers. A website visit can also generate several cookies if, for example, image files are hosted on another server.
* **User settings**such as language and special preferences. The website operator does not want to force the user to adjust the website according to their needs each time they visit it. Therefore, such settings are stored by cookies.
* **Time spent**on the website or individual sub-pages. This data is collected for statistical evaluation.
* **Data entered by the user via web forms –**such as e-mail address, name, or telephone number. This also includes search terms entered in search masks.
* **Visited sub-pages** such as product pages in online shops. This data is highly relevant for online marketing.
* **Meta data** such as the expiry date or time of a cookie, the path, and the security specifications (e.g. “HTTPS only”). While some cookies are deleted after leaving the website (which is particularly common in online banking), other cookies remain for years.

1. **Cookie Security**

Imagine that after login, our web server at **www.example.com** sets a session cookie like this:

Set-Cookie: **session=89bea4bb-85d1-4f8b-b4fa-a9b8db015e2b**; Max-Age=2600000; **Domain=example.com**; **Path=/**; Secure; HttpOnly.

Whenever the browser sends a request using https, our server can nosh the fine cookie and from its taste it knows that there's a session with the GUID 89bea4bb-85d1-4f8b-b4fa-a9b8db015e2b.It can then answer the request and amend it with information from the user that is currently logged in with this session id. The web shop at **shop.example.com** is a single-page web application and it communicates with **api.example.com**, which also uses the same session cookie information

An attacker now sets up a new websites. Let's call it example-com-product-reviews.**attacker.com**. The attacker lures users onto his website by providing additional information for products only found on shop.example.com. **When a user visits his site**, he returns a website with some content and **additional JavaScript**. The JavaScript can't read our session cookie. **First of all**, example-com-product-reviews.attacker.com is not matched by the example.com domain option, and **secondly** the cookie is set to HttpOnly, so JavaScript can't touch it anyways. This way the **attacking JavaScript** can't do anything with our cookie directly. But **what the JavaScript can do is use the Fetch API (the modern replacement of the old XMLHttpRequest object) to send a request to the shops API**. It could, for example send a malicious request to POST https://api.example.com/addItemToCart?itemId=123.

While the user reads the information presented on example-com-product-reviews.attacker.com, the **JavaScript executes and the browser sends the request**. Since this **request** is sent to api.example.com, which matches the domain option from our session cookie, and the path also matches, **the browser indeed attaches the cookie** to the request. The API server receives the request, finds the attached session cookie, determines the user and adds a very expensive item to the shopping cart - without the user even noticing.

Another forged request like POST https://api.example.com/checkoutUsingAlreadyProvidedPaymentMethod could then cause real damage.

This attack method is known as **Cross Site Request Forgery**, or short CSRF. If you want you can read more about that in the [CRSF Page on OWASP](https://owasp.org/www-community/attacks/csrf).

Now, there is another technology in place that could help prevent these kinds of attacks. It is called CORS (**Cross Origin Resource Sharing**) and involves a bit of knowledge on how to apply it correctly. For now, we assume that somehow the common development setting Allow any origin made it on the production server on api.example.com instead of a specific setting to only allow requests from shop.example.com. And while this might sound a bit constructed, configuration mistakes can happen - and sadly they also do happen all the time. It just takes a project setup at a time where the final domains aren't known yet, so AnyOrigin is used for development purposes. Later when the shop needs to be rolled out everybody is so stressed out that nobody thinks about checking and reviewing all these // Todo: Configure this to the actual shop domain, OR ELSE! comments left in the codebase - in good faith that this will be done before release.

But what is the 'same' domain as per the SameSite rules? Usually, the combination of second (**example**.com) and top level domain (example.**com**) is the common denominator. For example when the web application served from www.example.com sends a request to shop.example.com, this would be considered as a same-site request.

However, there are exceptions from that. One prominent example would be co.uk or co.jp, where commercial domain registrants get a third level domain all under the co second level domain. Requests between mysite.co.uk and attacker.co.uk should not be same-site, although they are both under co.uk. Another example are subdomains that can be used by different parties, for example mysite.azurewebsites.net and attacker.azurewebsites.net for Microsoft Azure or \*.compute.amazonaws.com for Amazons cloud services. Here you also want stronger isolation, as the sibling subdomains are absolutely not under your control.

Since the browser needs to decide whether a request is same-site or not, and technically anybody can register a domain and offer its subdomains for such purposes, there needs to be a way for browser vendors to figure out if a domain should be treated as same-site or not. This is where the [Public Suffix List](https://publicsuffix.org/) comes in handy. This list provides the domains that are a "public suffix" (aka under which different parties can register their own subdomains / prefixes).

Based on the information provided by this list, the Browsers are able to distinguish between first and third party cookies based on the current domain.

**Now, what do these SameSite Cookie options do?**

Let's go with the easiest settings first, None: This is the same as the original behavior of cookies where the SameSite option did not exist: Every cookie will be sent along with every request, as long as the values of domain and path match.

The second is Strict: This defines that cookies are only sent to the server when the request is *initiated* by a first-party context. What that actually means is that when you enter https://shop.example.com into your Browsers address bar, or you click on a link to that URL, this very first request to the server is *not* yet a first-party context (as it was initiated by an unknown party). Only subsequent requests that originate from that loaded page are considered being in the first-party context and as such provide the cookie to the server.

Now, for our session cookie this is a bit too harsh: The first request to shop.example.com would not provide the session cookie, so the initial response of the shop would be the same as for a not-logged in user, but the next navigation on the application would show the user as logged in.

This is where Lax provides its services: This specifies that the cookie will be sent to the server also on top-level navigations to our site, i.e. when the user follows a link to our shop. So even on the first navigation to our site the cookie will be sent and the user immediately appears as logged in.

The most important thing is that Strict and Lax both will prevent sending of the cookies for all other requests from a 3rd party context. This means that in our example the web site served from example-com-product-reviews.attacker.com both described attacks won't work anymore. The image-tag requesting the user profile picture from our shop will not send the cookie along to shop.example.com (so the fake page won't be able to show the real users image). Nor will the browser send the cookie along with the POST requests to api.example.com.

[Ask Question](https://stackoverflow.com/questions/ask)