



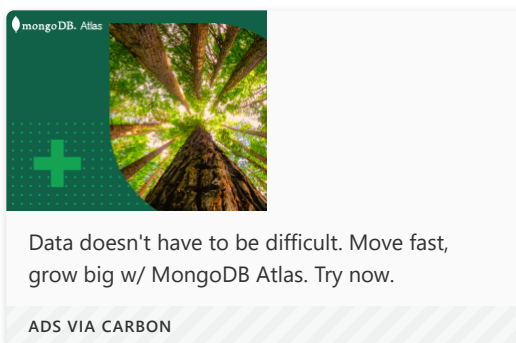
VALENTINO GAGLIARDI

/ HIRE

[LEARN](#)

Last updated: June 3, 2020 by Valentino Gagliardi - 25 minutes read

A practical, Complete Tutorial on HTTP cookies



Learn how HTTP cookies work: simple, practical examples with JavaScript and Python.

A re-introduction to

HTTP cookies



What are cookies in web development?

Cookies are tiny pieces of data that the backend can store in the user's browsers. User tracking, personalization, and most important, **authentication**, are the most common use cases for cookies.

Cookies have a lot of privacy concerns, and have been subject to strict regulation over the years.

In this post I'll focus mainly on the technical side: you'll learn how to create, use, and work with HTTP cookies, on the frontend, and on the backend.

What you will learn

In the following guide you'll learn:

- how to work with cookies, backend and frontend
- cookie **security and permissions**
- interaction between **cookies, AJAX, and CORS**

TABLE OF CONTENTS

- [What are cookies in web development?](#)
- [What you will learn](#)
- [Setting up the backend](#)
- [Who creates cookies?](#)
- [How to see cookies?](#)
- [I've got a cookie, what now?](#)
- [Cookies can expire: Max-Age and expires](#)
- [Cookies are scoped by path: the Path attribute](#)
- [Cookies are scoped by domain: the Domain attribute](#)
 - [Non matching host \(wrong host\)](#)
 - [Non matching host \(subdomain\)](#)
 - [Matching host \(whole domain\)](#)

- [Cookies and the Public Suffix List](#)
- [Matching host \(subdomain\)](#)
- [Cookies can travel over AJAX requests](#)
- [Cookies cannot always travel over AJAX requests](#)
- [Dealing with CORS](#)
- [A concrete example](#)
- [Cookies can be kind of secret: the Secure attribute](#)
- [Don't touch my cookie: the HttpOnly attribute](#)
- [The dreaded SameSite attribute](#)
 - [First and third-party cookie](#)
 - [Working with SameSite](#)
- [Cookies and authentication](#)
 - [Session based authentication](#)
 - [When to use session based authentication?](#)
 - [A note on JWT](#)
- [Wrapping up](#)
- [Further resources](#)

Setting up the backend

The examples for the **backend** are in **Python with Flask**. If you want to follow along, create a new Python virtual environment, move into it, and install Flask:

```
mkdir cookies && cd $_  
  
python3 -m venv venv  
source venv/bin/activate  
  
pip install Flask
```

In the project folder create a new file named `flask_app.py`, and use my examples to experiment locally.

Who creates cookies?

First things first, **where does cookies come from? Who creates cookies?**

While it's possible to create cookies in the browser with `document.cookie`, most of the times it's responsibility of the **backend to set cookies in the response before sending it to the client**.

By **backend here we mean** that cookies can be created by:

- the actual application's code on the backend (Python, JavaScript, PHP, Java)
- a webserver responding to requests (Nginx, Apache)

For doing so the backend sets in the response an HTTP header named **Set-Cookie** with a corresponding string made of a key/value pair, plus optional attributes:

```
Set-Cookie: myfirstcookie=somecookievalue
```

When and where to create these cookies depends on the requirements.

So, **cookies** are simple strings. Consider this example in Python with Flask. Create a Python file named **flask_app.py** in the project folder with the following code:

```
from flask import Flask, make_response

app = Flask(__name__)

@app.route("/index/", methods=["GET"])
def index():
    response = make_response("Here, take some cookie!")
    response.headers["Set-Cookie"] = "myfirstcookie=somecookievalue"
    return response
```

Then run the app:

```
FLASK_ENV=development FLASK_APP=flask_app.py flask run
```

When this application is running, and the user visits <http://127.0.0.1:5000/index/> the backend sets a **response header** named **Set-Cookie** with a key/value pair.

(127.0.0.1:5000 is the default listening address/port for Flask applications in development).

The **Set-Cookie** header is the key to understand how to create cookies:

```
response.headers["Set-Cookie"] = "myfirstcookie=somecookievalue"
```

On the right side you can see the actual cookie **"myfirstcookie=somecookievalue"**.

Most frameworks have their own utility functions for setting cookies programmatically, like Flask's **set_cookie()**.

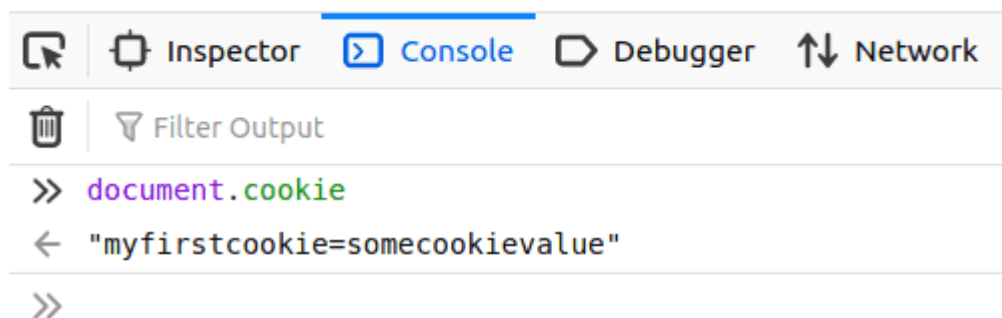
Under the hood they simply set a header in the response with **Set-Cookie**.

How to see cookies?

Consider again the previous example with Flask. Once you visit <http://127.0.0.1:5000/index/>, the backend sets a cookie in the browser. To see this cookie you can either call `document.cookie` from the browser's console:

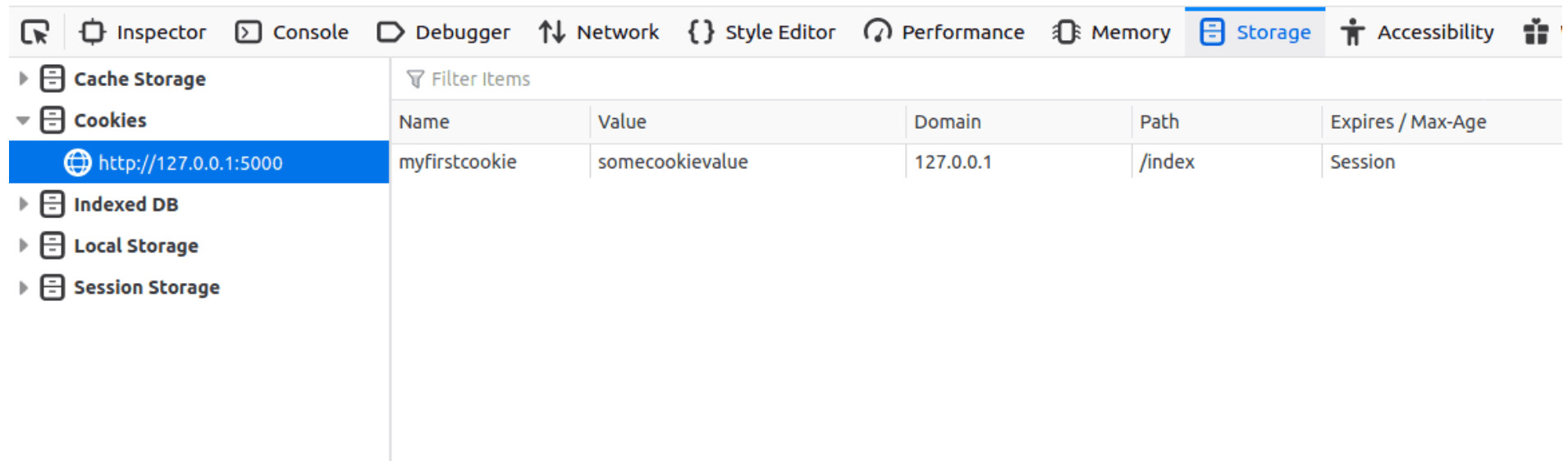


Here, take some cookie!



Or you can check the **Storage** tab in the developer tools. Click on **Cookies**, and you should see the cookie there:

Here, take some cookie!



On a command line you can use also **curl** to see what cookies the backend sets:

```
curl -I http://127.0.0.1:5000/index/
```

To save cookies to a file for later use:

```
curl -I http://127.0.0.1:5000/index/ --cookie-jar mycookies
```

To display cookies on stdout:

```
curl -I http://127.0.0.1:5000/index/ --cookie-jar -
```

Note that cookies without the `HttpOnly` attribute are accessible on `document.cookie` from JavaScript in the browser. On the other hand a cookie marked as `HttpOnly` cannot be accessed from JavaScript.

To mark a cookie as `HttpOnly` pass the attribute in the cookie:

```
Set-Cookie: myfirstcookie=somecookievalue; HttpOnly
```

Now the cookie will still appear in the Cookie Storage tab, but `document.cookie` will return an empty string.

From **this point on for convenience I'll use Flask's `response.set_cookie()` to create cookies on the backend.**

To inspect cookies along the way in this guide we'll use alternatively:

- curl
- Firefox developer tools
- Chrome developer tools

I've got a cookie, what now?

Your browser gets a cookie. Now what? Once you have a cookie, the browser **can send back the cookie to the backend**.

This could have a number of applications: user tracking, personalization, and most important, **authentication**.

For example, once you log in in a website the backend can give you a cookie:

```
Set-Cookie: userid=sup3r4n0m-us3r-1d3nt1f13r
```

To **properly identify you on each subsequent request, the backend checks the cookie coming from the browser in the request**.

To send the cookie, the browser appends a **Cookie** header in the request:

```
Cookie: userid=sup3r4n0m-us3r-1d3nt1f13r
```

How, when, and why the browser sends back cookies is the topic for the next sections.

Cookies can expire: Max-Age and expires

By default, **cookies expire when the user closes the session, that is, when she closes the browser**. To persist a cookie we can pass `expires` or `Max-Age` attributes:

```
Set-Cookie: myfirstcookie=somecookievalue; expires=Tue, 09 Jun 2020 15:46:52 GMT; Max
```

When both attributes are present, `Max-Age` has precedence over `expires`.

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: cookienamewrong-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):

× Headers Preview Response Initiator Timing Cookies

▼ General

Request URL: https://serene-bastion-01422.herokuapp.com/get-wrong-domain-cookie/
Request Method: GET
Status Code: 200 OK
Remote Address: 3.226.96.129:443
Referrer Policy: no-referrer-when-downgrade

▼ Response Headers view source

Connection: keep-alive
Content-Length: 36
Content-Type: text/html; charset=utf-8
Date: Fri, 29 May 2020 07:58:46 GMT
Server: gunicorn/20.0.4
Set-Cookie: cookiename=wr0ng-d0m41n-c00k13; Domain=api.valentinog.com; Path=/ ⚠
Via: 1.1 vegur

Non matching host (subdomain)

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

```
Set-Cookie: cookiename=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:

▼ General

Request URL: https://serene-bastion-01422.herokuapp.com/get-wrong-subdomain-cookie/
Request Method: GET
Status Code:  200 OK
Remote Address: 34.192.55.25:443
Referrer Policy: no-referrer-when-downgrade

▼ Response Headers [view source](#)

Connection: keep-alive
Content-Length: 39
Content-Type: text/html; charset=utf-8
Date: Fri, 29 May 2020 08:23:31 GMT
Server: gunicorn/20.0.4
Set-Cookie: coookieName=wr0ng-subd0m41n-c00k13; Domain=secure-brushlands-44802.herokuapp.com; Path=/ 
Via: 1.1 vegur

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:

```
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**.

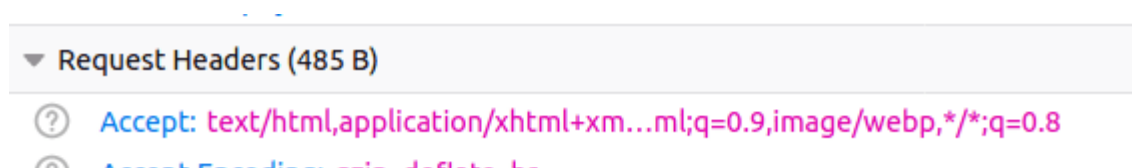
What matters is the domain the cookie is coming from.

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:



ⓘ Accept-Encoding: gzip, deflate, br
ⓘ Accept-Language: en-US,en;q=0.5
ⓘ Cache-Control: max-age=0
ⓘ Connection: keep-alive
ⓘ Cookie: cookiename=d0m41n-c00k13
ⓘ Host: www.valentinog.com
ⓘ If-Modified-Since: Mon, 25 May 2020 17:20:51 GMT
ⓘ If-None-Match: W/"5ecbfe73-6ce8"
ⓘ TE: Trailers
ⓘ Upgrade-Insecure-Requests: 1
ⓘ User-Agent: Mozilla/5.0 (X11; Ubuntu; Linu...) Gecko/20100101 Firefox/76.0

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

Request URL: <https://api.valentinog.com/api/link/>
Request Method: GET
Remote Address: 18.202.214.229:443
Status Code: 200 OK ⓘ
Version: HTTP/1.1

🔍 Filter Headers

▶ Response Headers (281 B)

▼ Request Headers (410 B)

ⓘ Accept: text/html,application/xhtml+xml;q=0.9,image/webp,*/*;q=0.8
ⓘ Accept-Encoding: gzip, deflate, br
ⓘ Accept-Language: en-US,en;q=0.5
ⓘ Cache-Control: max-age=0
ⓘ Connection: keep-alive

Cookie: cookiename=d0m41n-c00k13
Host: api.valentinog.com
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linu...) Gecko/20100101 Firefox/76.0

Cookies and the Public Suffix List

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

```
Set-Cookie: cookiename=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](#)
- [Cookies and the Public Suffix List](#)

Matching host (subdomain)

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

```
Set-Cookie: cookiename=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="cookiename", value="subd0m41n-c00k13")
```

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

Filter Items			
Name	Value	Domain	Pa

p.com

cookiename

subd0m41n-c00k13

serene-bastion-01422.herokuapp.com

/

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:

```
Request URL: https://serene-bastion-01422.herokuapp.com/
Request Method: GET
Remote Address: 52.202.112.181:443
Status Code: 200 OK ⓘ
Version: HTTP/1.1
```

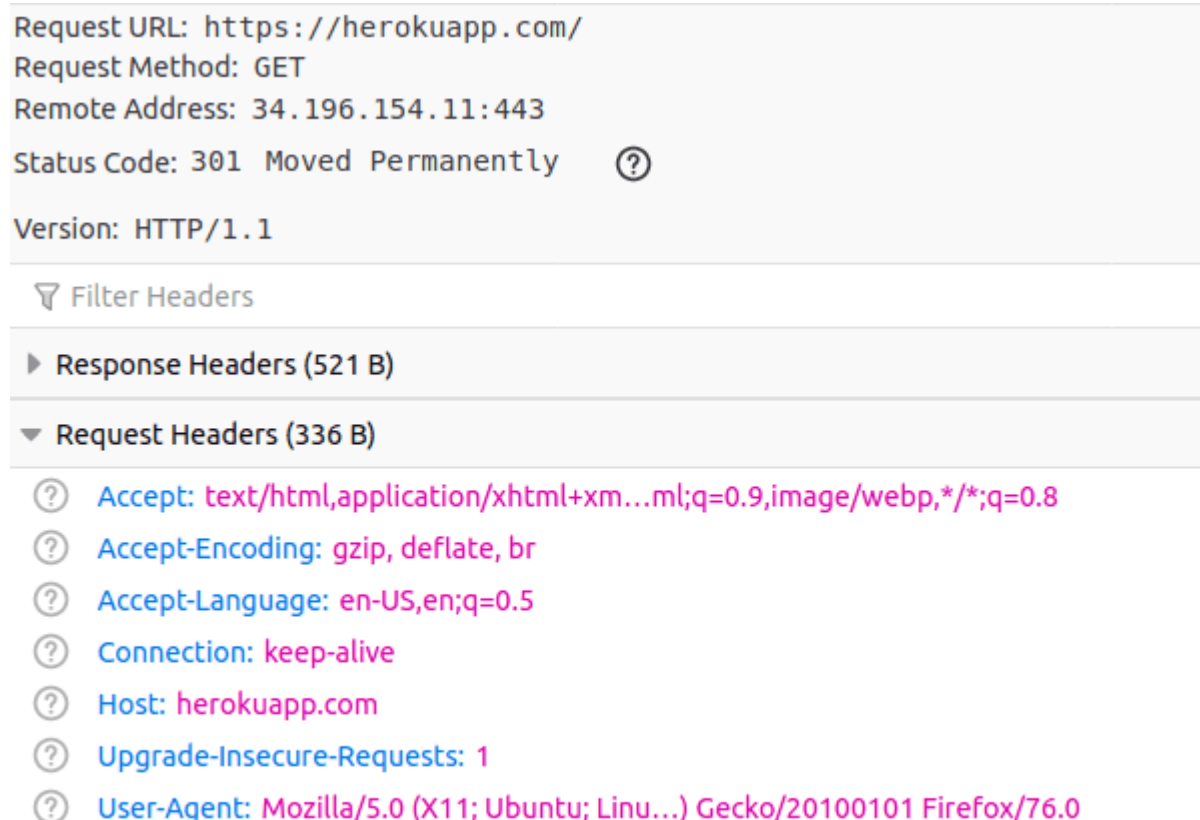
Filter Headers

▶ Response Headers (182 B)

▼ Request Headers (395 B)

- ⓘ Accept: text/html,application/xhtml+xml;q=0.9,image/webp,*/*;q=0.8
- ⓘ Accept-Encoding: gzip, deflate, br
- ⓘ Accept-Language: en-US,en;q=0.5
- ⓘ Connection: keep-alive
- ⓘ Cookie: cookiename=subd0m41n-c00k13
- ⓘ Host: serene-bastion-01422.herokuapp.com
- ⓘ Upgrade-Insecure-Requests: 1
- ⓘ User-Agent: Mozilla/5.0 (X11; Ubuntu; Linu...) Gecko/20100101 Firefox/76.0

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



(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 can travel over AJAX requests

Cookies can travel over AJAX requests. **AJAX requests** are asynchronous HTTP requests made with JavaScript (XMLHttpRequest or Fetch) to get and send back data to a backend.

Consider another example with Flask where we have a template, which in turn loads a JavaScript file. Here's the Flask app:

```
from flask import Flask, make_response, render_template

app = Flask(__name__)

@app.route("/", methods=["GET"])
def index():
    return render_template("index.html")

@app.route("/get-cookie/", methods=["GET"])
def get_cookie():
    response = make_response("Here, take some cookie!")
    response.set_cookie(key="id", value="3db4adj3d")
    return response
```

Here's the template in `templates/index.html`:

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
<button>FETCH</button>
</body>
<script src="{ { url_for('static', filename='index.js') } }"></script>
</html>
```

Here's the JavaScript code in `static/index.js`:

```
const button = document.getElementsByTagName("button")[0];

button.addEventListener("click", function() {
  getACookie();
});

function getACookie() {
  fetch("/get-cookie/")
    .then(response => {
      // make sure to check response.ok in the real world!
      return response.text();
    });
}
```

```
    })  
    .then(text => console.log(text));  
}
```

When visiting <http://127.0.0.1:5000/> we see a button. By clicking the button we make a Fetch request to /get-cookie/ to obtain a cookie back. As expected the cookie lands in the browser's Cookie storage.

Now let's change a bit our Flask app to expose another endpoint:

```
from flask import Flask, make_response, request, render_template, jsonify  
  
app = Flask(__name__)  
  
@app.route("/", methods=["GET"])  
def index():  
    return render_template("index.html")  
  
@app.route("/get-cookie/", methods=["GET"])  
def get_cookie():  
    response = make_response("Here, take some cookie!")  
    response.set_cookie(key="id", value="3db4adj3d")
```

```
return response
```

```
@app.route("/api/cities/", methods=["GET"])
def cities():
    if request.cookies["id"] == "3db4adj3d":
        cities = [{"name": "Rome", "id": 1}, {"name": "Siena", "id": 2}]
        return jsonify(cities)
    return jsonify(msg="Ops!")
```

Also, let's tweak our JavaScript code so that we make another Fetch request after getting the cookie:

```
const button = document.getElementsByTagName("button")[0];

button.addEventListener("click", function() {
    getACookie().then(() => getData());
});

function getACookie() {
    return fetch("/get-cookie/").then(response => {
        // make sure to check response.ok in the real world!
        return Promise.resolve("All good, fetch the data");
    });
};
```

```
}

function getData() {
  fetch("/api/cities/")
    .then(response => {
      // make sure to check response.ok in the real world!
      return response.json();
    })
    .then(json => console.log(json));
}
```

When visiting <http://127.0.0.1:5000/> we see a button. By clicking the button we make a Fetch request to `/get-cookie/` to obtain a cookie back. As soon as the cookie comes, we make another Fetch request to `/api/cities/`.

In the browser's console you should see an array of cities. Also, in the Network tab of the developer tool you should see a header named **Cookie**, transmitted to the backend over the AJAX request:



```
? Accept-Encoding: gzip, deflate
? Accept-Language: en-US,en;q=0.5
? Connection: keep-alive
? Cookie: id=3db4adj3d
? Host: 127.0.0.1:5000
? Referer: http://127.0.0.1:5000/
? User-Agent: Mozilla/5.0 (X11; Ubuntu; Linu...) Gecko/20100101 Firefox/76.0
```

This **cookie exchange back and forth between frontend and backend works fine as long as the frontend is in the same context of the backend**: we say that they're on the same origin.

That's because by default, Fetch sends **credentials, i.e. cookies** only when the request hits the same origin from which the request fires.

Here, JavaScript is served by a Flask template on <http://127.0.0.1:5000/>.

Let's see instead what happens for different origins.

Cookies cannot always travel over AJAX requests

Consider a different situation where the backend runs stand-alone, so you have this Flask app running:

```
FLASK_ENV=development FLASK_APP=flask_app.py flask run
```

Now in a different folder, outside of the Flask app, create an `index.html`:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
</head>
<body>
<button>FETCH</button>
</body>
<script src="index.js"></script>
</html>
```

Create in the same folder a JavaScript file named `index.js` with the following code:

```
const button = document.getElementsByTagName("button")[0];

button.addEventListener("click", function() {
  getACookie().then(() => getData());
});
```



```
function getACookie() {  
  return fetch("http://localhost:5000/get-cookie/").then(response => {  
    // make sure to check response.ok in the real world!  
    return Promise.resolve("All good, fetch the data");  
  });  
}  
  
function getData() {  
  fetch("http://localhost:5000/api/cities/")  
    .then(response => {  
      // make sure to check response.ok in the real world!  
      return response.json();  
    })  
    .then(json => console.log(json));  
}
```

In the same folder, from the terminal run:

```
npx serve
```

This command gives you a local address/port to connect to, like `http://localhost:42091/`. Visit the page and try to click the button with the browser's console open. In the console you should see:

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote resource.
```

Now, `http://localhost:5000/` is not the same as `http://localhost:42091/`. They're different **origins**, hence **CORS** kick ins.

An **origin consists of a scheme, domain, and port number**. That means `http://localhost:5000/` is a different origin from `http://localhost:42091/`.

Dealing with CORS

CORS, acronym for Cross-Origin Resource Sharing, is a way for servers to control access to resources on a given origin, when JavaScript code running on a different origin requests these resources.

By default, browsers block AJAX requests to remote resources which are not on the same origin, unless a specific HTTP header named `Access-Control-Allow-Origin` is exposed by the server.

To fix this first error we need to configure CORS for Flask:

```
pip install flask-cors
```

Then apply CORS to Flask:

```
from flask import Flask, make_response, request, render_template, jsonify
from flask_cors import CORS

app = Flask(__name__)
CORS(app=app)

@app.route("/", methods=["GET"])
def index():
    return render_template("index.html")

@app.route("/get-cookie/", methods=["GET"])
def get_cookie():
    response = make_response("Here, take some cookie!")
    response.set_cookie(key="id", value="3db4adj3d")
    return response
```

```
@app.route("/api/cities/", methods=["GET"])
def cities():
    if request.cookies["id"] == "3db4adj3d":
        cities = [{"name": "Rome", "id": 1}, {"name": "Siena", "id": 2}]
        return jsonify(cities)
    return jsonify(msg="Ops!")
```

Now try to click again the button with the browser's console open. In the console you should see:

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote res
```

Despite we got the same error, this time the culprit lies in the second route.

There's no such cookie named "id" attached to the request, so Flask crashes and no **Access-Control-Allow-Origin** gets set.

You can confirm this by looking at the request in the Network tab. No such **Cookie** is sent:

```
Request URL: http://localhost:5000/api/cities/
Request Method: GET
Remote Address: 127.0.0.1:5000
Status Code: 500 INTERNAL SERVER ERROR ⓘ
Version: HTTP/1.0
```

Referrer Policy: no-referrer-when-downgrade

Filter Headers

▼ Response Headers (192 B)

- ? Connection: close
- ? Content-Type: text/html; charset=utf-8
- ? Date: Wed, 27 May 2020 07:03:48 GMT
- ? Server: Werkzeug/1.0.1 Python/3.8.3
- ? X-XSS-Protection: 0

▼ Request Headers (309 B)

- ? Accept: */*
- ? Accept-Encoding: gzip, deflate
- ? Accept-Language: en-US,en;q=0.5
- ? Connection: keep-alive
- ? Host: localhost:5000
- ? Origin: http://localhost:35235
- ? Referer: http://localhost:35235/
- ? User-Agent: Mozilla/5.0 (X11; Ubuntu; Linu...) Gecko/20100101 Firefox/76.0

To include cookies in a Fetch requests across different origins we must provide the **credentials** flag (by default it's same origin).

Without this flag Fetch simply ignores cookies. To fix our example:

```
const button = document.getElementsByTagName("button")[0];

button.addEventListener("click", function() {
```

```
getACookie().then(() => getData());
});

function getACookie() {
  return fetch("http://localhost:5000/get-cookie/", {
    credentials: "include"
  }).then(response => {
    // make sure to check response.ok in the real world!
    return Promise.resolve("All good, fetch the data");
  });
}

function getData() {
  fetch("http://localhost:5000/api/cities/", {
    credentials: "include"
  })
  .then(response => {
    // make sure to check response.ok in the real world!
    return response.json();
  })
  .then(json => console.log(json));
}
```

`credentials: "include"` has to be present on the first Fetch request, to save the cookie in the browser's Cookie storage:

```
fetch("http://localhost:5000/get-cookie/", {  
  credentials: "include"  
})
```

It has also to be present on the second request to allow transmitting cookies back to the backend:

```
fetch("http://localhost:5000/api/cities/", {  
  credentials: "include"  
})
```

Try again, and you'll see we need to fix another error on the backend:

```
Cross-Origin Request Blocked: The Same Origin Policy disallows reading the remote res
```

To **allow cookie transmission** in CORS requests, the backend needs to expose the `Access-Control-Allow-Credentials` header as well. Easy fix:

```
CORS(app=app, supports_credentials=True)
```

Now you should see the expected array of cities in the browser's console.

Takeaways: to make cookies travel over AJAX requests between different origins provide:

- `credentials: "include"` on the frontend for Fetch
- `Access-Control-Allow-Credentials` and `Access-Control-Allow-Origin` on the backend.

Cookies can travel over AJAX requests, but they have to respect the domain rules we described earlier.

Resources:

- [Fetch request credentials](#)
- [XMLHttpRequest.withCredentials](#)
- [Cross-origin fetches](#)

A concrete example

Our previous example uses localhost to keep things simple and replicable on your local machine.

To **imagine cookie exchange over AJAX requests in the real world** you can think of the following scenario:

1. a user visits <https://www.a-example.dev>
2. she clicks a button or makes some action which triggers a Fetch request to <https://api.b-example.dev>
3. <https://api.b-example.dev> sets a cookie with `Domain=api.b-example.dev`
4. on subsequent Fetch requests to <https://api.b-example.dev> the cookie is sent back

Cookies can be kind of secret: the Secure attribute

But not so secret after all.

The `Secure` attribute for a cookie ensures that the cookie **is never accepted over HTTP**, that is, **the browser rejects secure cookies unless the connection happens over HTTPS**.

To mark a cookie as `Secure` pass the attribute in the cookie:

```
Set-Cookie: "id=3db4adj3d; Secure"
```

In Flask:

```
response.set_cookie(key="id", value="3db4adj3d", secure=True)
```

If you want to try against a live environment, run the following command on the console and note how curl here **does not save the cookie over HTTP**:

```
curl -I http://serene-bastion-01422.herokuapp.com/get-secure-cookie/ --cookie-jar -
```

Note: this will work only in curl 7.64.0 >= which implements rfc6265bis. Older versions of curl implement RCF6265. [See](#)

Over HTTPS instead, the cookie appears in the cookie jar:

```
curl -I https://serene-bastion-01422.herokuapp.com/get-secure-cookie/ --cookie-jar -
```

Here's the jar:

```
serene-bastion-01422.herokuapp.com    FALSE    /    TRUE    0    id    3db4
```

To try the cookie in a browser visit both versions of the url above and check out the Cookie storage in the developer tool.

Don't get fooled by **Secure**: browsers accept the cookie over **HTTPS**, but there's no protection for the cookie once it lands in the browser.

For this reason a **Secure** cookie, like any cookie, is not intended for transmission of sensitive data, even if the name would suggest the opposite.

Don't touch my cookie: the HttpOnly attribute

The **HttpOnly** attribute for a cookie ensures that the cookie **is not accessible by JavaScript code**. This is the most **important form of protection against XSS attacks**

However, it is **sent on each subsequent HTTP request**, with respect of any permission enforced by **Domain** and **Path**.

To mark a cookie as **HttpOnly** pass the attribute in the cookie:

```
Set-Cookie: "id=3db4adj3d; HttpOnly"
```

In Flask:

```
response.set_cookie(key="id", value="3db4adj3d", httponly=True)
```

A cookie marked as `HttpOnly` cannot be accessed from JavaScript: if inspected in the console, `document.cookie` returns an empty string.

However, **Fetch can get, and send back** `HttpOnly` cookies when `credentials` is set to `include`, again, with respect of any permission enforced by `Domain` and `Path`:

```
fetch(/* url */, {  
  credentials: "include"  
})
```

When to use `HttpOnly`? **Whenever you can**. Cookies should always be `HttpOnly`, unless there's a specific requirement for exposing them to runtime JavaScript.

Resources:

- [What is XSS](#)
- [Protecting Your Cookies: HttpOnly](#)

The dreaded SameSite attribute

First and third-party cookie

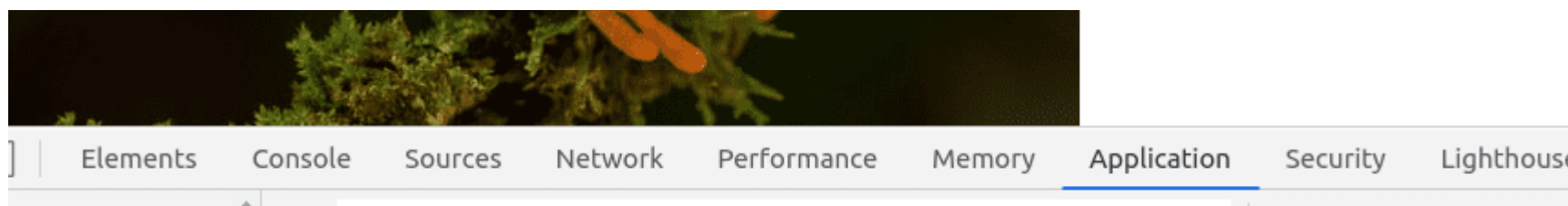
Consider a cookie acquired by visiting <https://serene-bastion-01422.herokuapp.com/get-cookie/>:

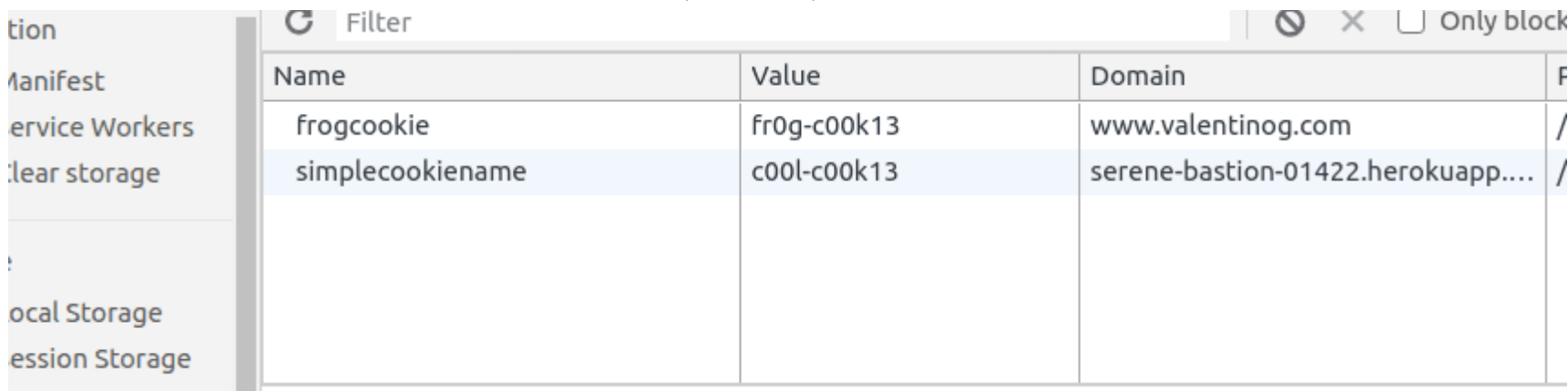
```
Set-Cookie: simplecookie=c00l-c00k13; Path=/
```

We refer to this kind of cookies as **first-party**. That is, I visit that URL in the browser, and if I visit the same URL, or another path of that site (provided that **Path** is `/`) the browser sends the cookie back to the website. Normal cookie stuff.

Now consider another web page at <https://serene-bastion-01422.herokuapp.com/get-frog/>. This page sets a cookie as well, and in addition it **loads an image from a remote resource** hosted at <https://www.valentinog.com/cookie-frog.jpg>.

This **remote resource in turns sets a cookie on its own**. You can see the actual scenario in this picture:





Name	Value	Domain	Path
frogcookie	fr0g-c00k13	www.valentinog.com	/
simplecookie	c00l-c00k13	serene-bastion-01422.herokuapp.com	/

Note: If you're on Chrome 85 you won't see this cookie. Starting from this version Chrome rejects it.

We refer to this kind of cookies as **third-party**. Another example of third-party cookie:

1. a user visits <https://www.a-example.dev>
2. she clicks a button or makes some action which triggers a Fetch request to <https://api.b-example.dev>
3. <https://api.b-example.dev> sets a cookie with `Domain=api.b-example.dev`
4. now the page at <https://www.a-example.dev> holds a **third-party** cookie from <https://api.b-example.dev>

Working with SameSite

At the time of writing, third-party cookies causes a warning to pop up in the **Chrome** console:

"A cookie associated with a cross-site resource at <http://www.valentinog.com/> was set without the SameSite attribute. A future release of Chrome will only deliver cookies with cross-site requests if they are set with SameSite=None and Secure.

What the browser is trying to say is that **third-party cookies** must have the new **SameSite** attribute. But why?

The **SameSite** attribute is a **new feature aimed at improving cookie security** to: prevent **Cross Site Request Forgery attacks**, avoid privacy leaks.

SameSite can be assigned one of these three values:

- **Strict**
- **Lax**
- **None**

If we are a service providing embeddable widgets (iframes), or we need to put cookies in remote websites (for a good reason and not for wild tracking), these cookies must be marked as **SameSite=None**, and **Secure**:

```
Set-Cookie: frogcookie=fr0g-c00k13; SameSite=None; Secure
```

Failing to do so will make the **browser reject the third-party cookie**. Here's what browsers are going to do in the near future:

A cookie associated with a cross-site resource at <http://www.valentinog.com/> was set without the SameSite attribute. It has been blocked, as Chrome now only delivers cookies with cross-site requests if they are set with SameSite=None and Secure.

In other words `SameSite=None; Secure` will make third-party cookies work as they work today, the only difference being that they must be transmitted only over HTTPS.

A cookie configured this way is sent alongside each request if domain and path matches. This is the normal behaviour.

Worth noting, `SameSite` does not concern only third-party cookies.

By default, **browsers will enforce `SameSite=Lax` on all cookies, both first-party and third-party, if the attribute is missing**. Here's Firefox Nightly on a first-party cookie:

Cookie "get_frog_simplecookienam" has "sameSite" policy set to "lax" because it is missing a "sameSite" attribute, and "sameSite=lax" is the default value for this attribute.

A `SameSite=Lax` cookie is sent back with **safe HTTP methods**, namely GET, HEAD, OPTIONS, and TRACE. **POST requests instead won't carry the cookie**.

Third-party cookies with `SameSite=Strict` instead will be rejected altogether by the browser.

To recap, **here's the browser's behaviour for the different values of SameSite:**

VALUE	INCOMING COOKIE	OUTGOING COOKIE
Strict	Reject	-
Lax	Accept	Send with safe HTTP methods
None + Secure	Accept	Send

To learn more about **SameSite** and to understand in detail all the use cases for this attribute, go read these fantastic resources:

- [Prepare for SameSite Cookie Updates](#)
- [SameSite cookies explained](#)
- [SameSite cookie recipes](#)
- [Tough Cookies](#)
- [Cross-Site Request Forgery is dead!](#)
- [CSRF is \(really\) dead](#)

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 15:4
```

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 known 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.

A note on JWT

JWT, short for JSON Web Tokens, is an authentication mechanism, rising in popularity in recent years.

JWT is well suited for single page and mobile applications, but it presents a new set of challenges. The typical flow for a frontend application wanting to authenticate against an API is the following:

1. Frontend sends credentials to the backend
2. Backend checks credentials and sends back a token
3. Frontend sends the token on each subsequent request

The main question which comes up with this approach is: **where do I store this token in the frontend for keeping the user logged in?**

The most natural thing to do for someone who writes JavaScript is to save the token in **localStorage**. This **is bad for so many reasons**.

localStorage is easily accessible from JavaScript code, and it's an **easy target for XSS attacks**.

To overcome this issue, most developers resort to save the **JWT token in a cookie** thinking that **HttpOnly** and **Secure** can protect the cookie, at least from XSS attacks.

The new **SameSite** attribute, set to **SameSite=Strict** would also protect your "cookified " JWT from CSRF attacks. But, is **also completely invalidates the use case for JWT in first instance** because **SameSite=Strict** does not sends cookies on cross-origin requests!

How about **SameSite=Lax** then? This mode allows sending cookies back with **safe HTTP methods**, namely GET, HEAD, OPTIONS, and TRACE. POST requests won't transmit the cookie either way.

Really, storing a JWT token in a cookie or in **localStorage** are both **bad ideas**.

If you really want to use JWT instead of sticking with session based auth, and scaling your session storage, **you might want to use JWT with refresh tokens** to keep the user logged in.

Resources:

- [The Ultimate Guide to handling JWTs on frontend clients \(GraphQL\)](#)
- [Stop using JWT for sessions](#)
- [Please, stop using localStorage](#)

Wrapping up

HTTP cookies have been there since 1994. They're everywhere.

Cookies are simple text strings, but they can be fine tuned for permissions, with **Domain** and **Path**, transmitted only over HTTPS with **Secure**, hide from JavaScript with **HttpOnly**.

A cookie might be used for personalization of the user's experience, user authentication, or shady purposes like tracking.

But, for all the intended uses, **cookies can expose users to attacks and vulnerabilities**.

Browser's vendors and the Internet Engineering Task Force have worked year after year to improve cookie security, the last recent step being **SameSite**.

So what makes a secure cookie? There isn't such a thing. We could consider relatively secure a cookie that:

- travels only over HTTPS, that is, has **Secure**
- has **HttpOnly** whenever possible
- has the proper **SameSite** configuration
- does not carry sensitive data

Thanks for reading!

Further resources

- [IETF on cookies](#)
- [Set-Cookie on MDN](#)
- [HTTP Cookies on MDN](#)
- [Your Cookie Questions Answered](#)

Icon in the featured picture by [freepik](#).

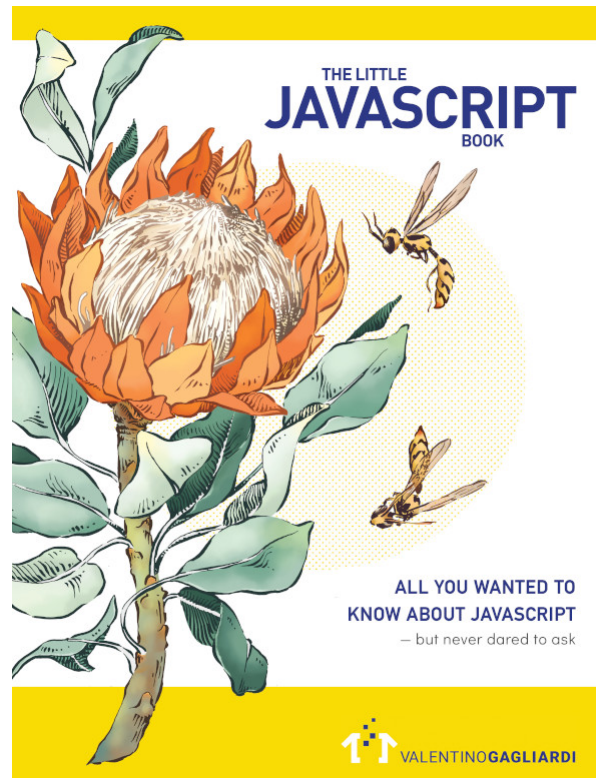
STAY UPDATED

Be the first to know when I publish new stuff.



BOOKS

The Little JavaScript Book



Hi! I'm Valentino! Educator and consultant, I help people learning to code with on-site and remote workshops. Looking for JavaScript and Python training? [Let's get in touch!](#)



:: All rights reserved 2020, Valentino Gagliardi - [Privacy_policy](#) - [Cookie_policy](#) ::