Web Development and API Design

Lesson 08: Authentication, CORS and CSRF

Prof. Andrea Arcuri

Goals

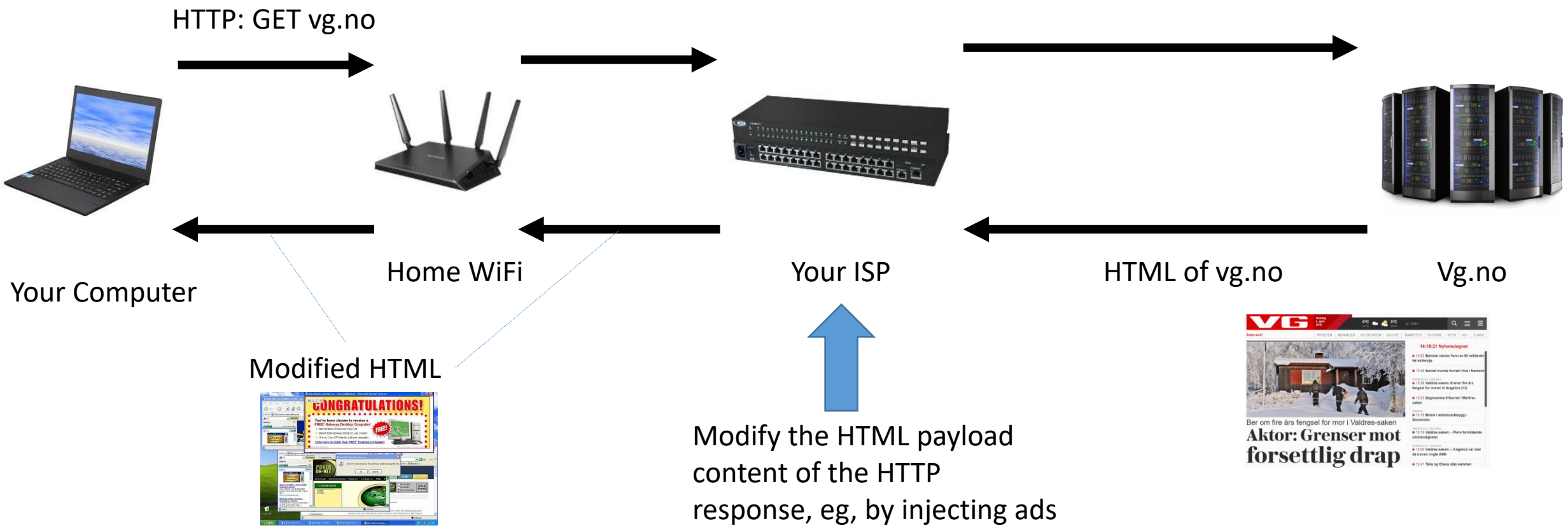
- Brief revision of HTTPS
- Revision/Introduction on how session-based authentication with cookies work
- Understand what *Cross-Origin Resource Sharing* (CORS) is
- Understand the risks of *Cross-Site Request Forgery* (CSRF)
- Learn how to add auth to a *NodeJS/React* application

HTTPS

HTTP is Not Secure

- Messages in HTTP are not secure, because not encrypted
- HTTPS extends HTTP by using encryption on all messages
- Important to do for *ALL* kinds of communications, even if not critical
  - Not just for authentication (login) in banks or other internet services
- Example, ISPs (eg Comcast) can inject ads in web pages
  - ISP -> Internet Service Provider

- What would be the point of encrypting pages of a newspaper?
- If not encrypted (ie HTTP instead of HTTPS) anyone between you and the target server can alter the payload of the messages…

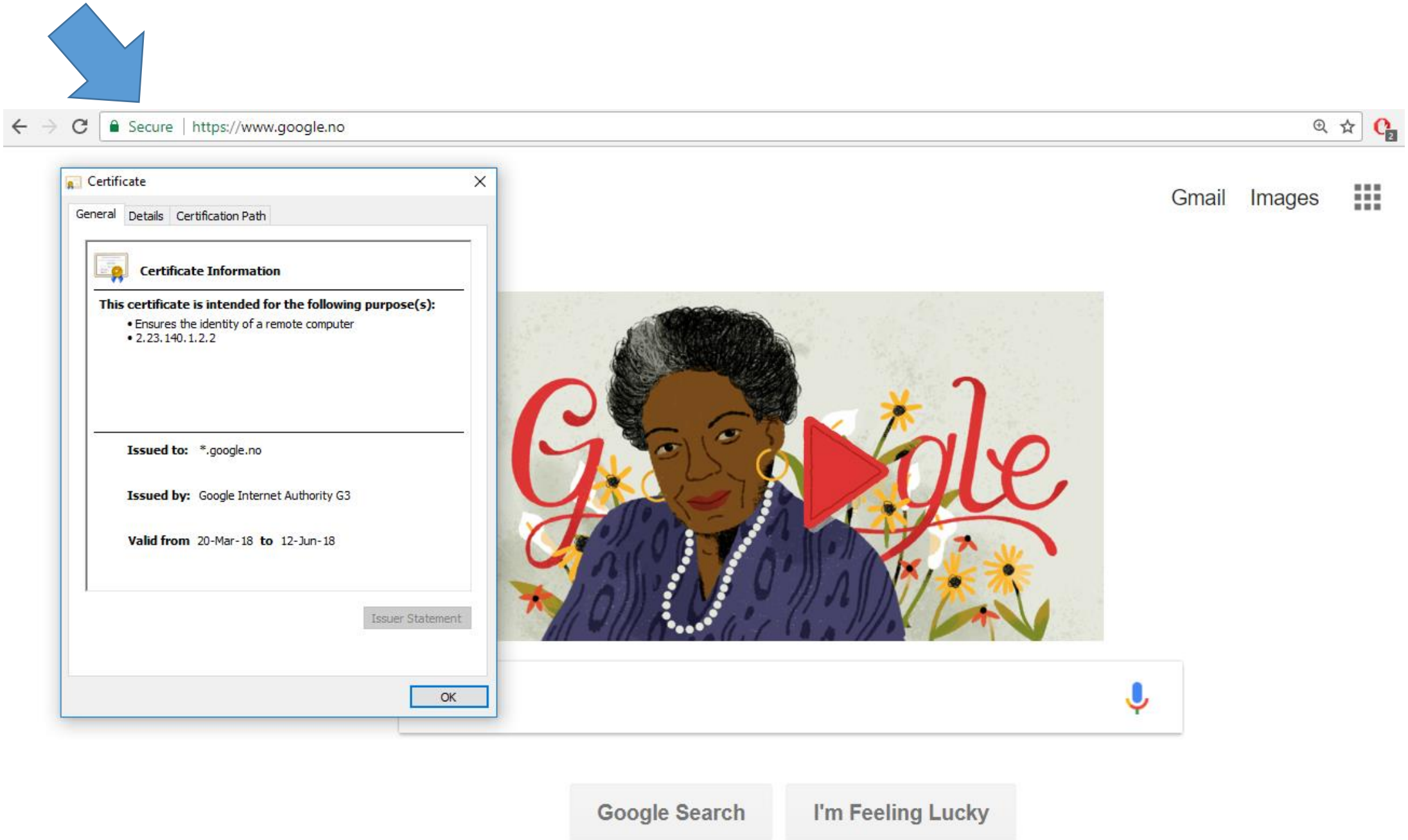


TLS/SSL Digital Certificates

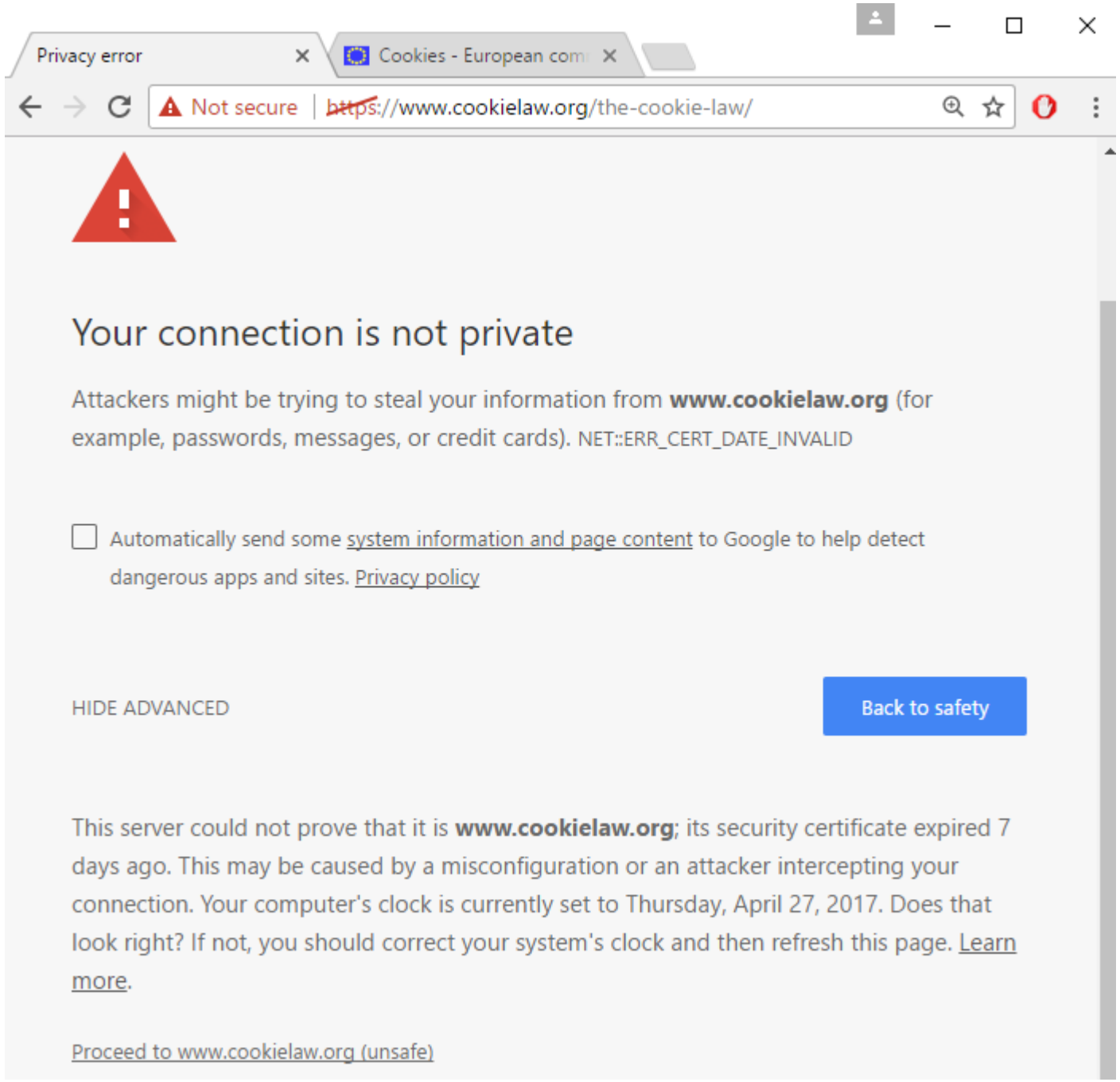
- When using HTTPS to a server, first download a digital certificate
- Digital Certificate (DC) contains:
  1. domain name (eg, *www.facebook.com*)
  2. trusted certificate authority (CA)
  3. server's *public encryption key*, ie RSA algorithm
- Certificates have expiration time
- The certificates themselves are signed, with the OS having public keys to check their integrity , ie they are signed with the private keys of the CA

Cont.

- Eve would not be able to forge a DC for *www.facebook.com* (eg needed for her phishing attacks), because she would need to sign it with the private key of a CA
- Note: this is secure only as long as one can *trust* the CAs, but some of those have been compromised in the past…


Secure | https://www.google.no
Certificate
General
Details
Certification Path
Certificate Information
This certificate is intended for the following purpose(s):
• Ensures the identity of a remote computer
• 2.23.140.1.2.2
Issued to: *.google.no
Issued by: Google Internet Authority G3
Valid from 20-Mar-18 to 12-Jun-18
Issuer Statement
OK
Gmail
Images
Google Search
I'm Feeling Lucky

What if browser detects that the certificate is invalid?



If Certification Fails

- 2 main possibilities
- 1) Expired certificate: developers have not renewed their certificates with the CA
- 2) Man-in-the-middle attack
  - Trying to pretend to be the web app you want to connect to, so can steal your login/password, eg by serving a fake login page that is equal to the original one
  - Easy way to do such attack? Use a router, have an open WiFi called “Free WiFi”, go close to a bar/restaurant, and wait for people to connect to it, give them fake pages with fake certificates, and wait for those people to click “Proceed” when browser complains about certificate is invalid
  - **NEVER PRESS “PROCEED” WITH INVALID CERTIFICATE ON UNTRUSTED NETWORK!!!** And if you really have to, do not provide any sensitive information, eg passwords, although it would be still a problem with cookies…

HTTPS in This Course

- When building and testing apps locally, we will use HTTP
  - dealing with HTTP**S** would add a lot of complications
- But, when we will deploy apps in the "*cloud*", those will be behind gateways using HTTP**S**
  - we will not need to do anything special to handle it

Login

Authentication/Authorization

- **Authentication**:
  - do I know who a user X is?
  - how to distinguish X from a different user Y?
- **Authorization**:
  - once I know that the current user is X, what is X allowed to do?
  - can s/he delete data?
  - can s/he see data of other users?
  - etc.
- Of course, they only make sense with encryption (eg HTTPS), so no one can decode and tamper with the messages…

Authentication/Authorization failures

- If not authenticated, server can:
  - in SSR, redirect to login page, HTTP status code 3xx
  - error page, HTTP status 401 *Unauthorized*
- If authenticated but not authorized
  - eg user X tries to access data of Y
  - 3xx redirection
  - HTTP status 403 *Forbidden*

Blacklisting vs Whitelisting

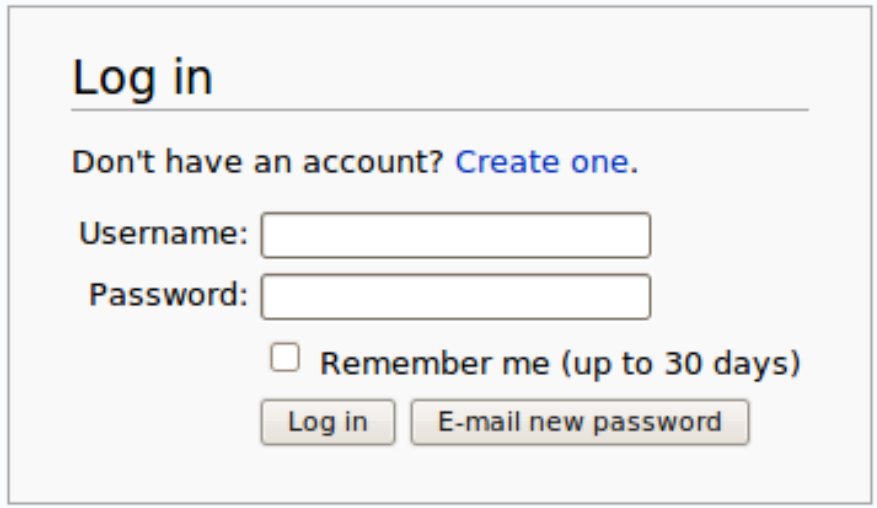
- Authorization is done on the server, and will depend on the language/framework
  - JEE, Spring, PHP, .Net, NodeJS, etc.
  - user will just get either a 3xx or 403 response
- *Blacklisting*: everything is allowed by default. What is not allowed for a given user/group has to be explicitly stated
  - Usually not a good idea, as easy to forget to blacklist some critical operation
- *Whitelisting*: nothing is allowed by default. What is allowed has to be explicitly stated
  - “forgetting to allow something” (reduced functionality) is much, much better than “forgetting to forbid something” (security problem)

Authentication: first steps

- Server does not know who the user is
- Server only sees incoming HTTP/S messages
  - not necessarily from a browser… user can do direct TCP connections from scripts
- HTTP/S is stateless
- Need a way to tell that sequence of HTTP/S calls come from same user
- User has to send information of who s/he is at **each** HTTP/S call
- But users can **lie**… (eg, hackers)

Ids and Passwords

- A user will be registered with a *unique* id
- Need also secret password to login
  - Otherwise anyone could login with the ids of other users…
- HTTP/S does not prevent attempts to login to accounts of other users



Sending *id/pwd*

- Need to send userId (*id*) and password (*pwd*) at **EACH** HTTP request
- Can put them inside the HTTP header *Authorization*
- Can be different formats to specify how *id/pwd* should be encoded
- *Basic* (RFC-7617): string "*id:pwd*" in Base64 encoding
- Ex *id=test* and *pwd=123£*, then header on **EACH** request:

  Authorization: Basic dGVzdDoxMjPCow==

Problems

- Base64 is NOT encrypted… it is just a mapping from bits into printable ASCII codes
- When sending *id/pwd*, must use HTTP**S**
  - otherwise, anyone on the network can read them
  - anyway, always use HTTP**S** instead of HTTP…
- What if someone intercepts a HTTP in clear, or has direct access to the browser (eg, via a malware)?
  - s/he will get the password

Authentication Token

- “Login” with *id/pwd* only **once**
- Server will return a *token* associated with that user *id,* stating s/he authenticated (assuming *pwd* was correct)
- From now on, instead of sending *id/pwd,* rather send the *token*
- Token will be valid only for a certain amount of time, after that, need to get new one via *id/pwd*
- *Benefits???*

Stolen Token

- If token is stolen, hacker can use it only for a *limited* amount of time, until it expires
- If user does **logout**, then token becomes invalid, and server will reject any further HTTP request with such token
  - so, even if hacker has the token, it will become *useless* for him/her
- *Critical* operations like changing password or transfer money could require a new login with *id/pwd*
  - and so hacker with stolen token cannot use it

Creating a Token

- Server could be instructed to create a token when receiving a HTTP request with header “*Authorization: Basic …*”
- This could be on any endpoint…
- … and/or could have a specific endpoint, e.g. “*/login*”
- But, in that case, I could choose how I want to send the *id/pwd* pair

How to send *id/pwd* to create a token?

- When talking about security and what to implement on the server, think about HTTP/S messages, *not necessarily coming from browsers*.
- Could have endpoint to get *token* from server given userId/password
  - Use such token on each following request as parameter
- GET /login?**userId=x**&**password=y**
  - userId/password as URL parameters to the /login endpoint
  - get back new token Z associated to this user, as HTTP/S response body, no HTML page
- GET /somePageIWantToBrowse**?token=z**
  - pass “token=z” parameter to each HTTP/S request

Awful Solution

- That solution would work, but…
- "*/login?userId=x&password=y*" would be *cached* in your browser history, even after you logout
- How to handle the adding of "*?token=z*" to all your <*a*> tags in the HTML pages?
  - doable, but quite cumbersome
- How to handle browser bookmarks?
  - tokens would be there, and made the links useless once they expire, eg after a logout

Login with POST

- User ids and passwords should *never* be sent with a GET
  - GET specs do not allow body in the requests
- Should be in HTTP body of a POST
  - This is typical case in HTML forms, and also what we need in SPAs
- *POST /login {“userId”: id, “password”: pwd}*
  - in SPAs, wants to send in *JSON* instead of *x-www-form-urlencoded* to help protecting from CSRF attacks… more on this later

Storing Tokens

- Browser needs to store authentication *tokens* somewhere
- Tokens need to be added at each HTTP request
- *Best* way to store tokens is HTTP **Cookies** marked with *HttpOnly*
  - automatically added on each HTTP request
  - cannot be read by JavaScript
- If you do *not* store authentication tokens in *HttpOnly* cookies, you are *more* vulnerable to **XSS** attacks!!!
  - We will see XSS in next class
  - Complex story… even with cookies, still vulnerable to XSS, but it would stop as soon as you close the browser… without cookies, token could be sent to malicious server via AJAX, and attacks continue from there
  - Note: this is a **huge** problem if you make the mistake of using JWT with no stateful whitelist/blacklist logout…

Cookies

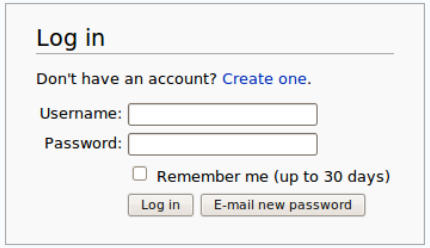
- Authentication "*tokens*" should not be in URLs, but in the HTTP Headers
- **Cookie**: special HTTP header that will be used to identify the user
- The user does not choose the cookie, it is the server that assigns them
- Recall: user can craft its own HTTP messages, so server needs to know if cookie values are valid

Login with Cookies

- Browser: POST /login
  - Username X and password as HTTP body
- Server: if login is successful, respond to the POST with a "*Set-Cookie*" header, with some *unique* and *non-predictable* identifier Y
  - Server needs to remember that cookie Y is associated with user X
  - *Set-Cookie: <cookie-name>=<cookie-value>*
- Browser: from now on, each following HTTP request will have "*Cookie:* Y" in the headers
- *Logout*: remove association between cookie Y and user X on server.
- Server: HTTP request with no cookie or invalid/expired cookie, give 401 error message

Request HTML page

GET /index.html



Send credentials by AJAX

POST /api/login
userId=foo&password=bar

Validate the credentials. If correct, create a session, identified by a cookie id

HTTP/1.1 204
Set-cookie: **123456**

Add cookie header in all following HTTP requests

GET /api/someData
Cookie: **123456**

Cookies and Sessions

- Servers would usually send a "*Set-Cookie*" regardless of login
  - want to know if requests are coming from same user, regardless if s/he is registered/authenticated
  - ie cookies used to define "*sessions*"
- After login could create a new session (ie, invalidate old cookie and create a new one) or use the existing session cookie (eg, the one set by the server when login page was retrieved with the first GET)
- Problem with re-using session cookies: make sure all the pages were served with HTTPS and not HTTP
  - Ie, use HTTPS for all pages, even the login one
  - do not use HTTP and then switch to HTTPS once login is done

Handling Cookies

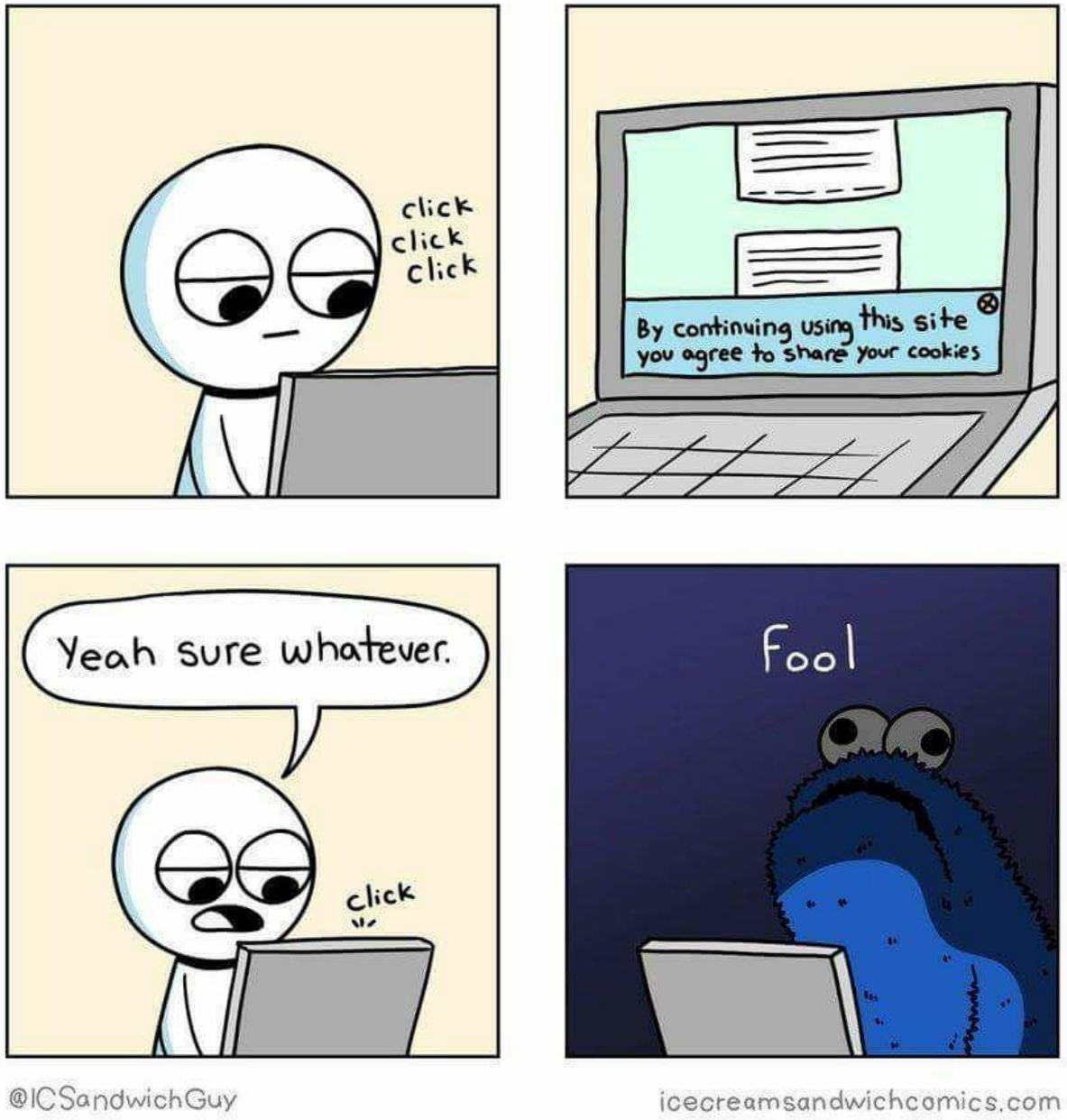
- The browser will store cookie values locally
- At each HTTP/S request, it will send the cookies in the HTTP headers
- Cookies are sent only to same server who asked to set them
  - eg, cookies set from "*foo.com*" are not going to be sent when I do GET requests to "*bar.org*"
- If not protected with *HttpOnly*, JavaScript can read those cookie values on the browser
- What is the problem with it?
  - You can fabricate a website with JS that reads all cookies, and send them back to you, so that you can use them to access the user's Google/Facebook/Bank accounts
- As cookies are arbitrary strings, they can be used to store data
  - usually up to 4K bytes per domain can be stored in a browser

Expires / Secure / HttpOnly

- **Set-Cookie: <name>=<value>; Expires=<date>; Secure; HttpOnly**
- *Expires*: for how long the cookie should be stored
- *Secure*: browser should send the cookie only over HTTPS, and NEVER on HTTP
  - There are kinds of attacks to trick a page to make a HTTP toward the same server instead of HTTPS, and so could read authentication cookies in plain text on the network
- *HttpOnly*: do not allow JS in the browser to read such cookie
  - This is critical for authentication cookies to avoid XSS attacks

Cookie Tracking

- Besides session/login cookies that have an expiration date, server can setup further cookies (ie *Set-Cookie* header)
- There are special laws regarding handling of cookies
- Why? Tracking and privacy concerns…

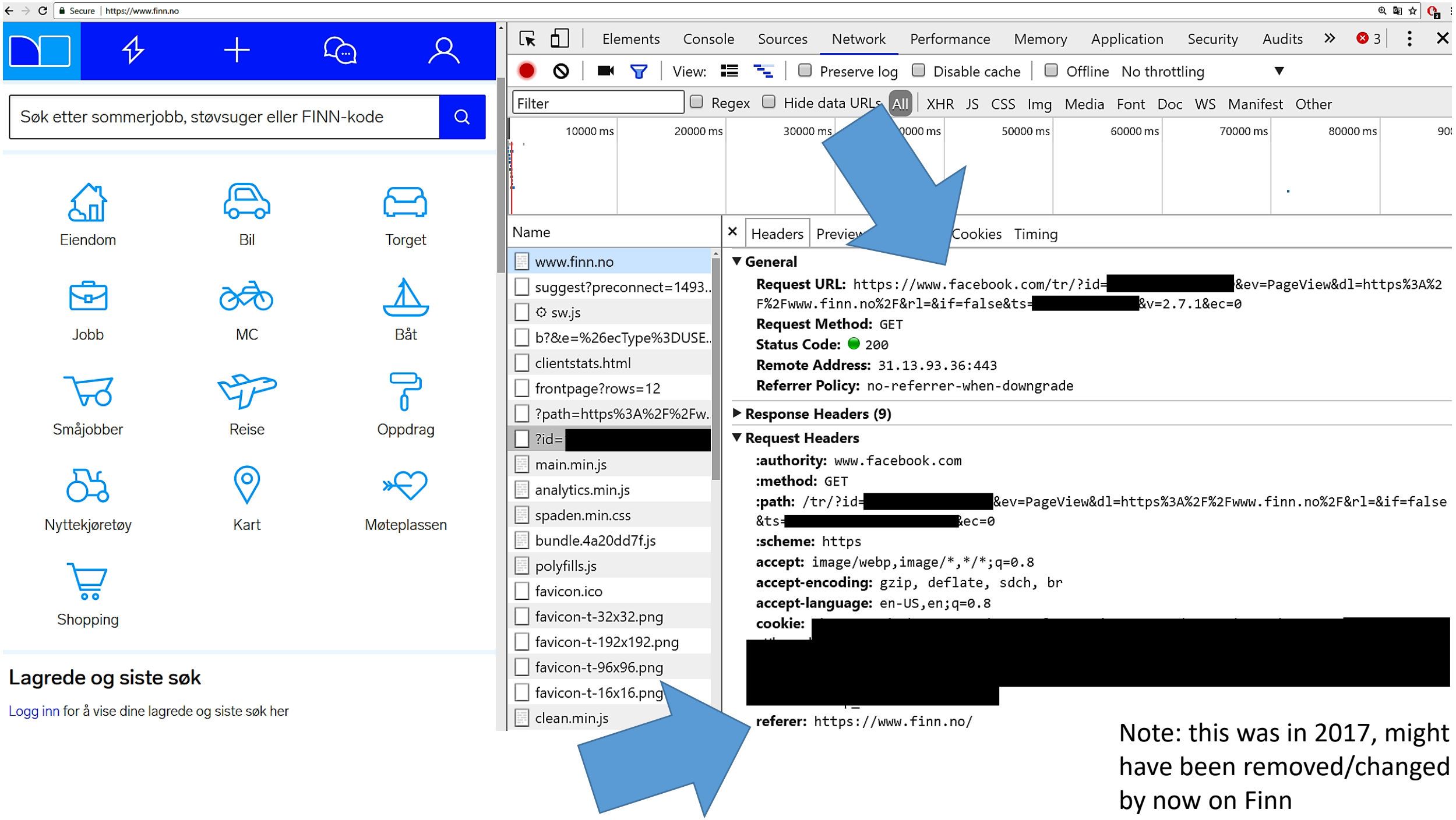


Tracking

- Many sites might rely on resources provided by other sites
  - Images, JavaScript files, CSS files, etc.
  - eg, FaceBook "*Like*" button
- When you download a HTML page from domain X (eg *finn.no*) which uses a resource from Y (eg, *facebook.com*), the HTTP GET request for Y will include previous cookies from Y
- So, even if you are logged out from Facebook, FB can know which pages you visit (as long as they do use FB resources), as can have permanent cookies stored on your browser and not related to a current session on FB
- Even worse, FB can track your browser even if you have never used FB!!!
- This happens by simply opening the page from X, no need to click anything!!!
- *referer* HTTP header: domain origin of request to Y from page not from Y
  - Eg, "*Referer: X*" is added when page loaded from X ask for resource in Y

```
<html>
  <head>
     <link href="foo.com/style.css"/>
     <script src="bar.com/x.js" />
  </head>
  <body>
      <img src="cat.com/cute.jpg">
  </body>
</html>
```

- Assume your site "*mysite.no*"
- 3 HTTP calls to external sites
- Each including cookie set for those domains
- Plus added **referer** header including string "*mysite.no*"



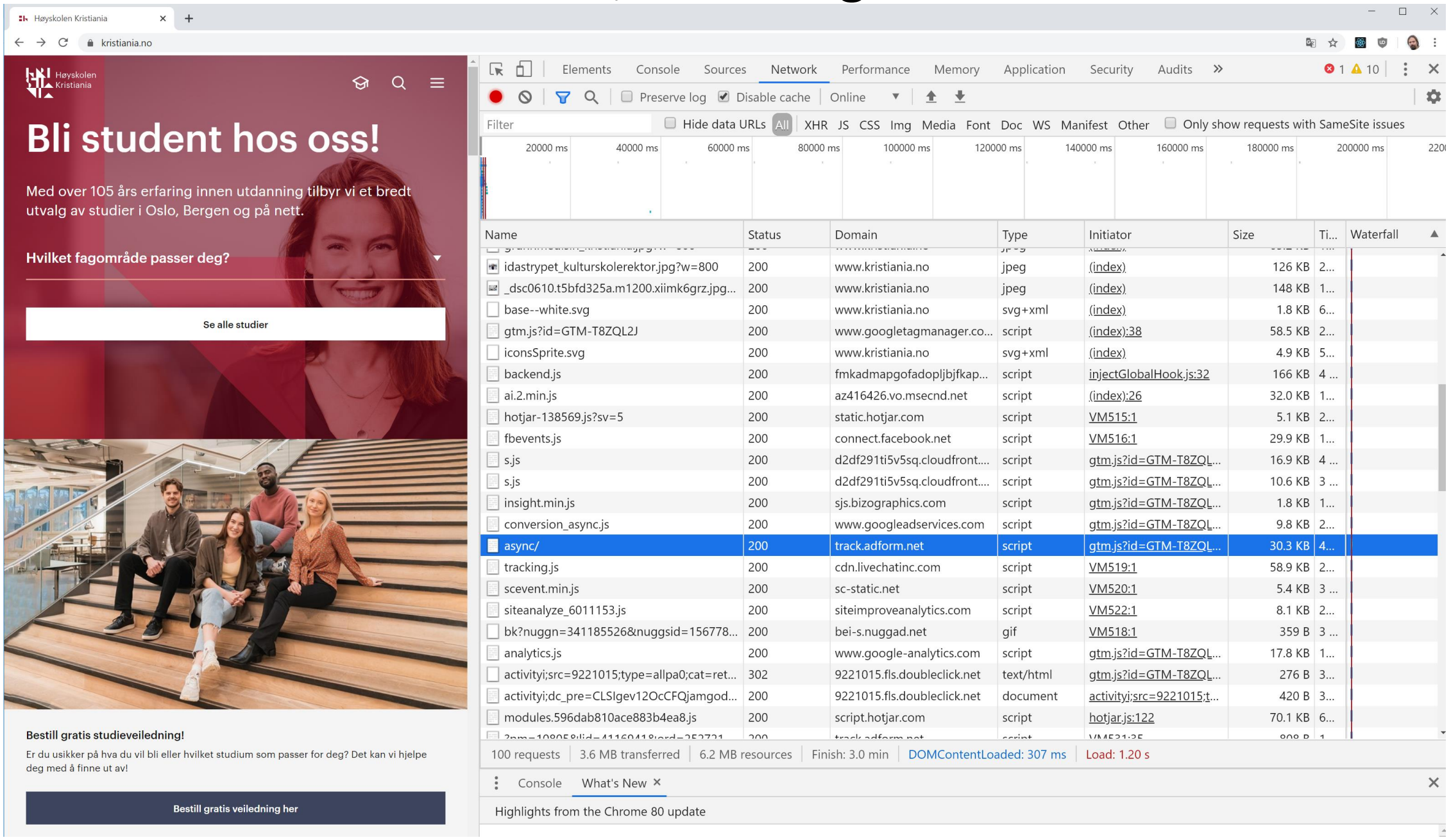
Note: this was in 2017, might have been removed/changed by now on Finn

How To Protect Yourself?

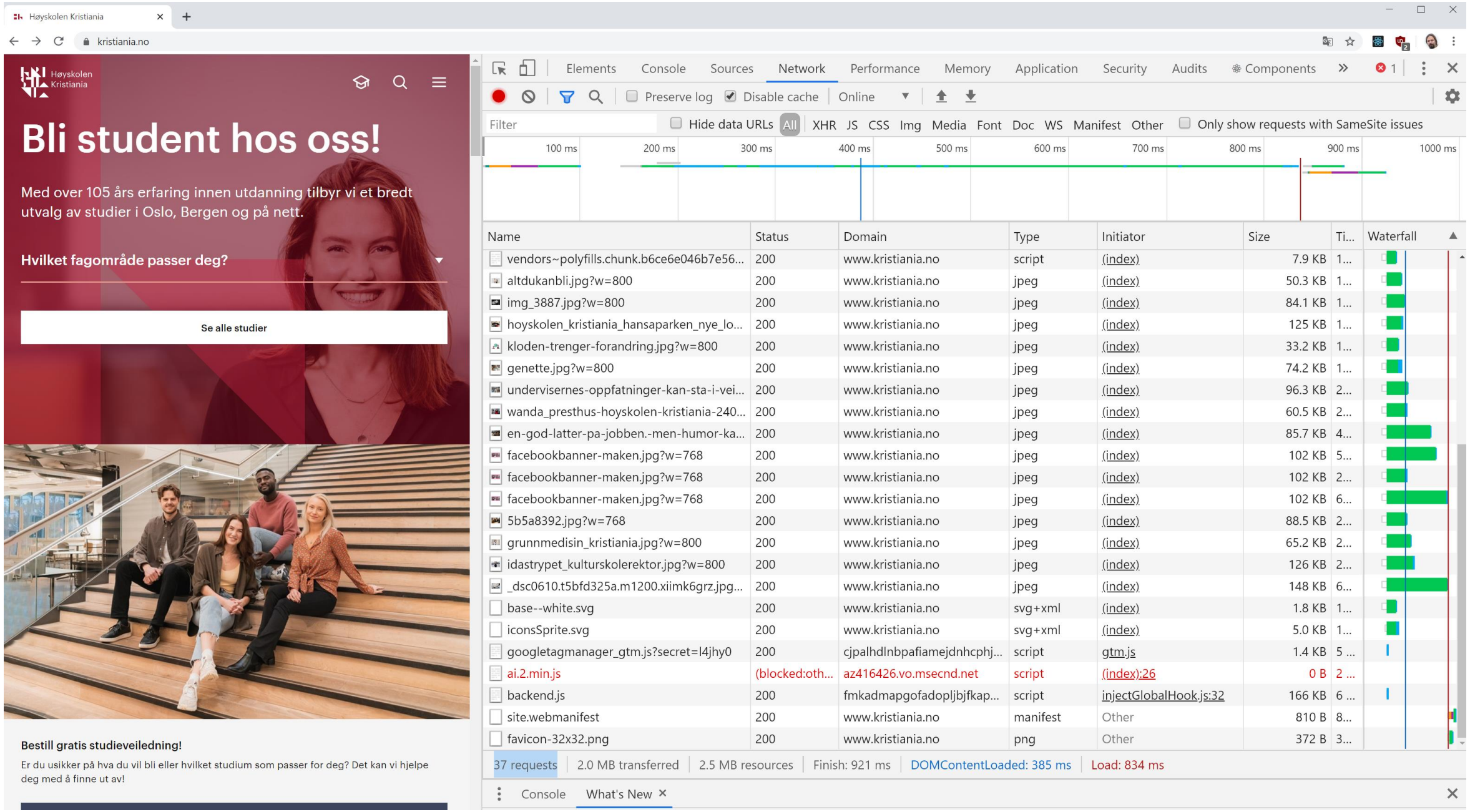
- Use a browser extension such as **uBlock Origin**
  - it automatically blocks requests to known external trackers
  - to not be confused with **uBlock**, which is a completely different product

- Some browsers like **Firefox** started to do this by default
  - slim chances this will be ever implemented by *Chrome*, as it is developed by an *advertising* company

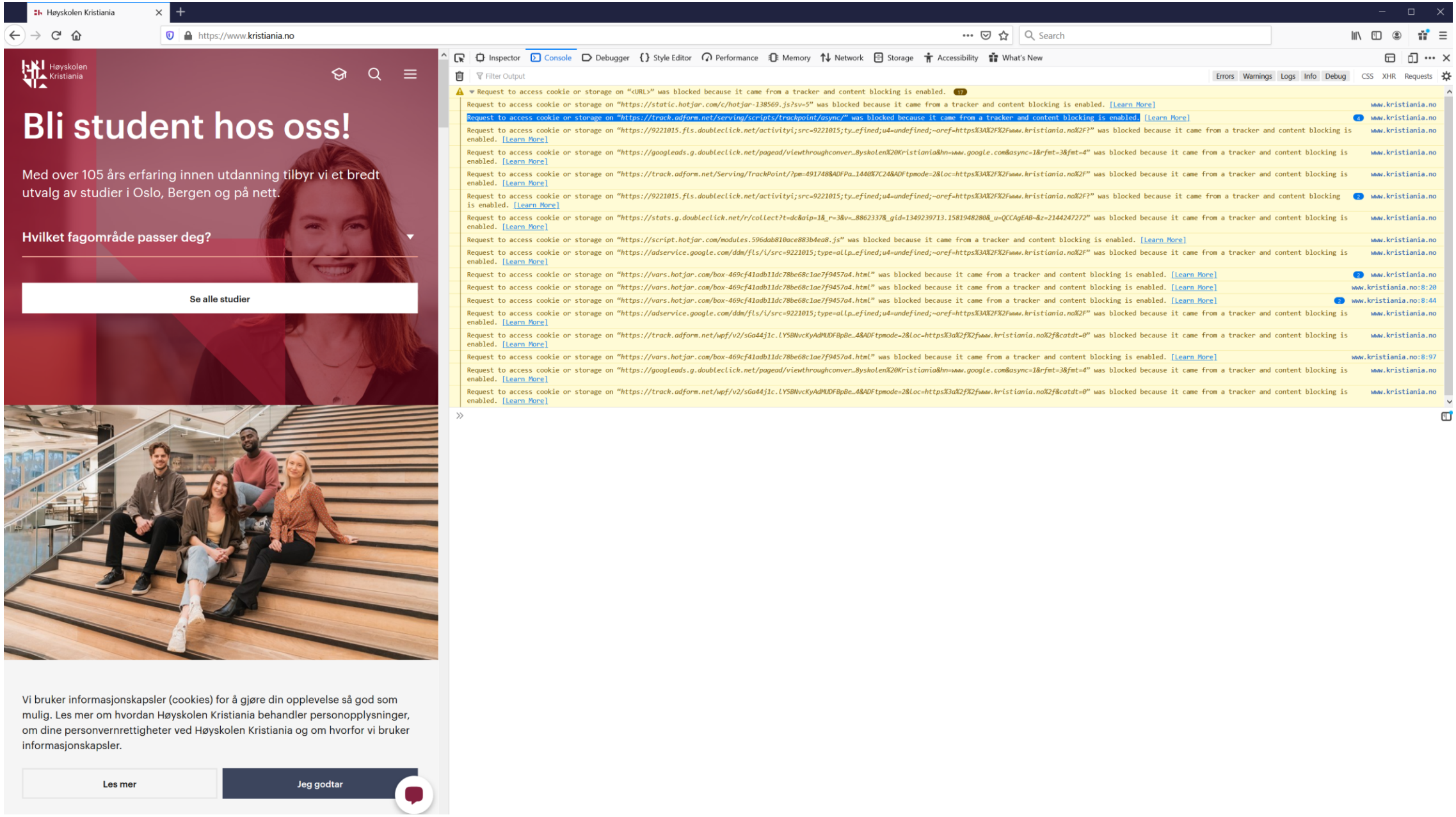
kristiania.no: 100 HTTP calls, including to **track.adform.net**



uBlock Origin: only 37 requests, all others all blocked



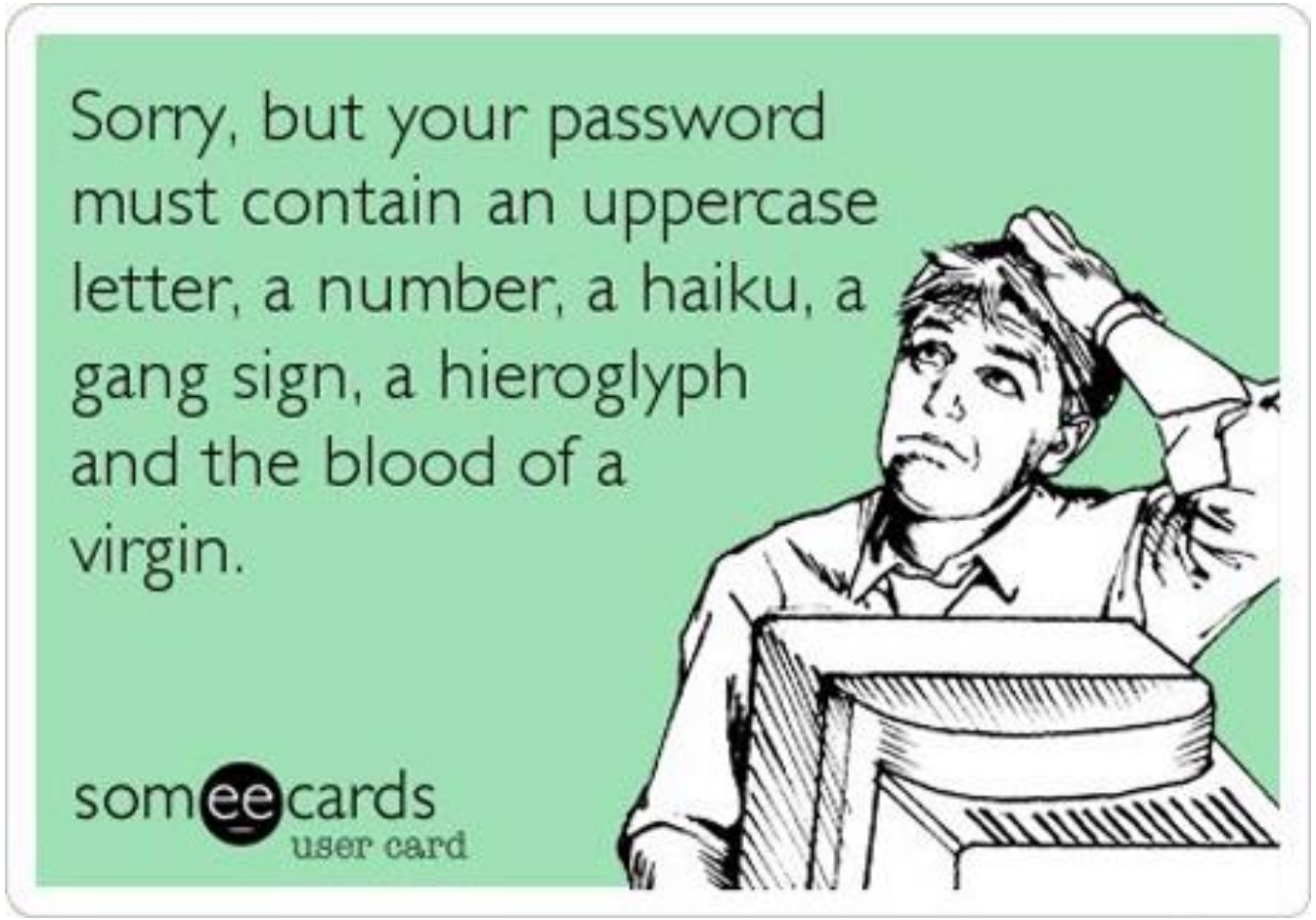
FireFox: 17 requests blocked by default, eg **track.adform.net**



Passwords

Passwords

- Needed to verify identity of a user
- Not too short or simple, otherwise too easy to crack with brute-force
- *Security* vs *Usability:* hard to get a good balance
  - eg, ideally would have different passwords for each different site, and change them often, eg every week… but who the heck is going to do that???



Password Storage

- When creating new user, need to save password somewhere, usually a database
- **NEVER SAVE A PASSWORD IN PLAIN TEXT**
- Passwords need to be *hashed*
- Even if an hacker has full access to database, shouldn't be able to get the passwords
  - Typical case is a successful SQL Injection attack
  - But many more cases: eg disgruntled employee, recovery from broken thrown away hard-drive, etc.
- Besides being able to impersonate a user, hacker can try the same password on other sites (Amazon/Facebook/etc)
- **WARNING**: in this course, we will not deal with hashing and proper storage of passwords

CORS and CSRF

HTTP and Cookies

- When browser requests resource for "*foo.com*", all cookies set by that domain are sent in the headers, session ones included
- This applies to **all** HTTP calls
  - HTML <*a*> and <*form*>
  - AJAX requests made with *XMLHttpRequest* and *fetch()*
- *Do you see the problem here?*
  - *Cross-Site Request Forgery* (CSRF) attack

Login to dnb.no
Set-cookie: **dnb=123**

www.dnb.no

Example of CSRF attack

Malicious AJAX POST
Cookie: **dnb=123**
Transfer all money to Eve

Visit malicious site, with
malicious JavaScript
automatically run on page load

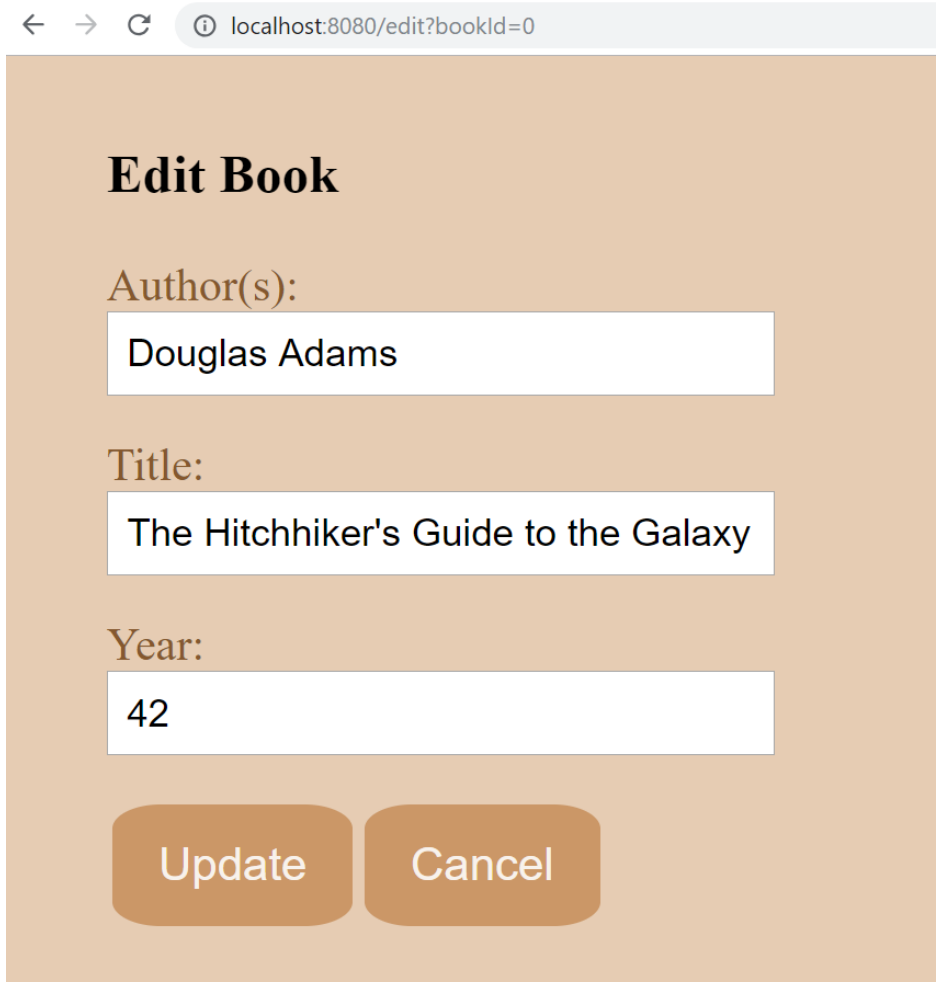
www.evil.no

Cross-Origin Resource Sharing (CORS)

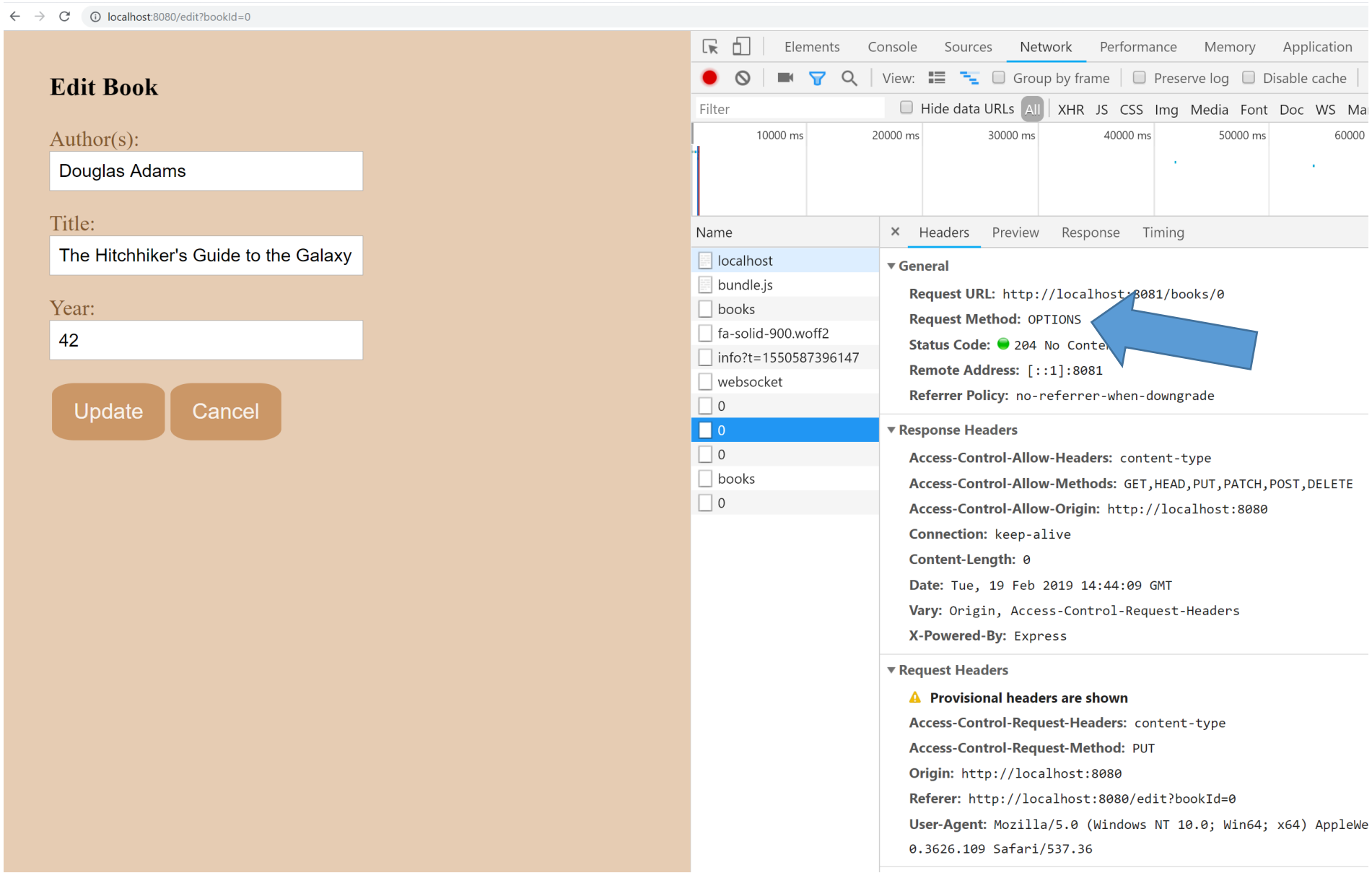
- By default, JS downloaded from site X cannot do AJAX calls to another domain Y
  - browsers will allow only AJAX calls toward the same domain (*ip:port*) of where the JS was downloaded from
  - eg, JS downloaded from *evil.no* can only do AJAX towards *evil.no*
- When trying to do such a HTTP call, a browser will first **preflight** it with an OPTIONS HTTP call
  - this will ask if the original HTTP call can be done to the server Y
  - Y will answer telling the browser whether to do or not the HTTP call
  - if Y said it was OK, then browser will do the original HTTP call
  - so, up to 2 HTTP calls

Separated Frontend and Backend

- Recall example of book app, where frontend was served from *localhost:8080*, whereas REST API for backend was on *localhost:8081*
- At that time we HAD to handle CORS on the backend
- Eg, what happens when we want to do a PUT to modify the state of a book?



Browser first does an OPTIONS to check if allowed to do the PUT



OPTIONS Request Headers

```
Access-Control-Request-Headers: content-type
Access-Control-Request-Method: PUT
Origin: http://localhost:8080
Referer: http://localhost:8080/edit?bookId=0
```

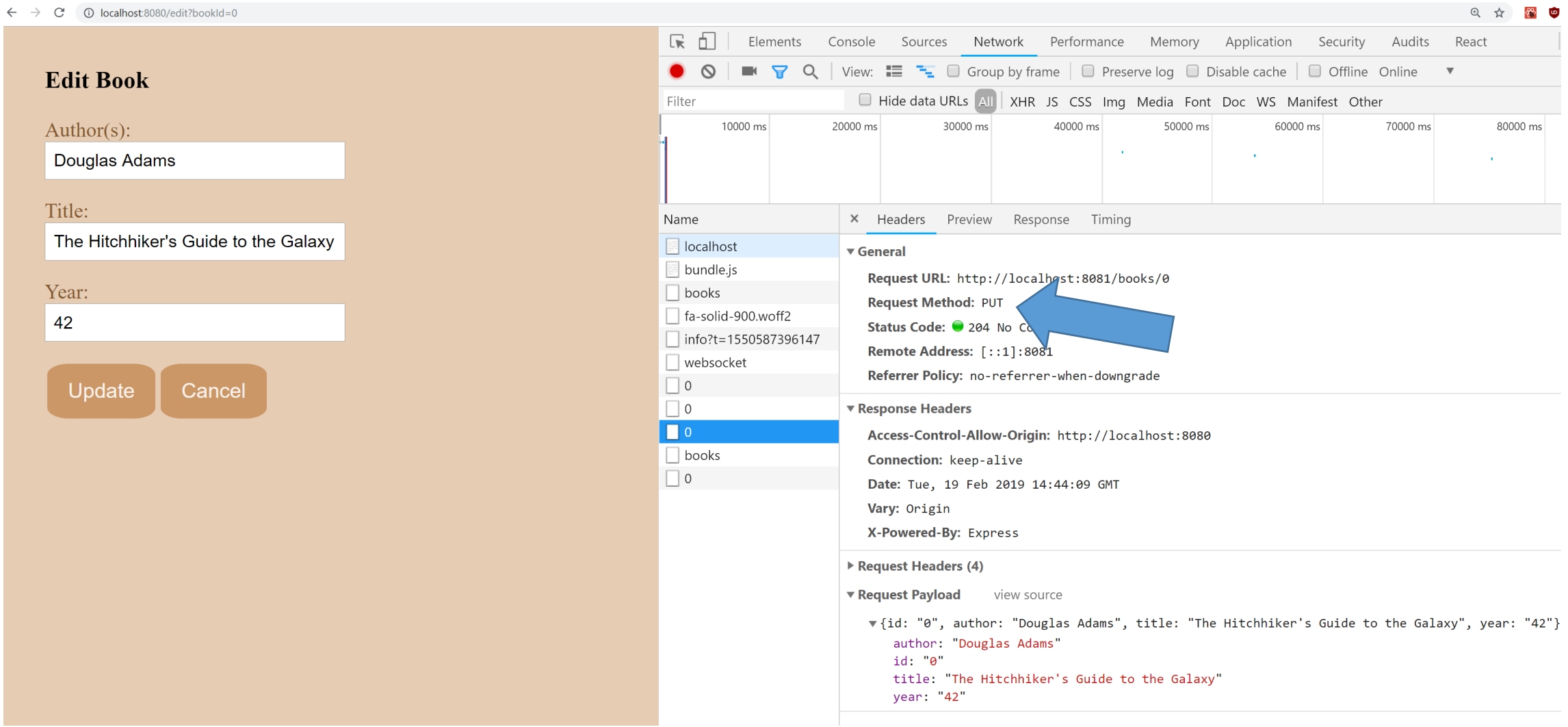
- **Access-Control-Request-Method**: which HTTP method we want to use
  - eg, PUT in the previous example
- **Access-Control-Request-Headers**: any custom header we want to use
  - eg, in our PUT, we want to specify that the payload is in JSON
- **Origin**: specify from where the JS making the AJAX call was downloaded
  - automatically added when making OPTIONS CORS calls
  - server will check this field
  - set by browser, cannot modify it with JS
- **Referer**: like Origin, but containing full path
  - used also outside of CORS, but could be blocked for privacy reasons

OPTIONS Response Headers

```
Access-Control-Allow-Headers: content-type
Access-Control-Allow-Methods: GET,HEAD,PUT,PATCH,POST,DELETE
Access-Control-Allow-Origin: http://localhost:8080
```

- Tell the browser what is allowed on that endpoint
  - eg which HTTP methods can be called using **Access-Control-Allow-Methods**
- By default, most servers will not allow cross-site requests
- If needed, you have to setup the server to add such CORS allowing headers
- This can be based on the **Origin**
  - eg, different origins might be allowed different rights

Browser will make the PUT request only if in the response of OPTIONS the server said it is OK



A Note on Chrome and Firefox

- As of Chrome 79, Developer Tools does NOT show preflight OPTION requests anymore
  - due to technical reasons, might be fixed in a following release
  - the previous screenshots were taken with Chrome < 79

- If need to debug CORS issues, just use *Firefox*…

OPTIONS No-Preflight

- Browser does **not** preflight *all* HTTP requests
- *Exceptions*: **GET**, **HEAD** and **POST** with specific **content-type**
  - **application/x-www-form-urlencoded**
  - **multipart/form-data**
  - **text/plain**
- Note: this is for "*historical*" reasons, but if not handled properly, it is a **SECURITY HOLE**

No-Preflight GET

Login to dnb.no
Set-cookie: **dnb=123**

www.dnb.no

Malicious AJAX GET
Cookie: **dnb=123**
Obtain sensitive info, like account balance

Can steal sensitive information

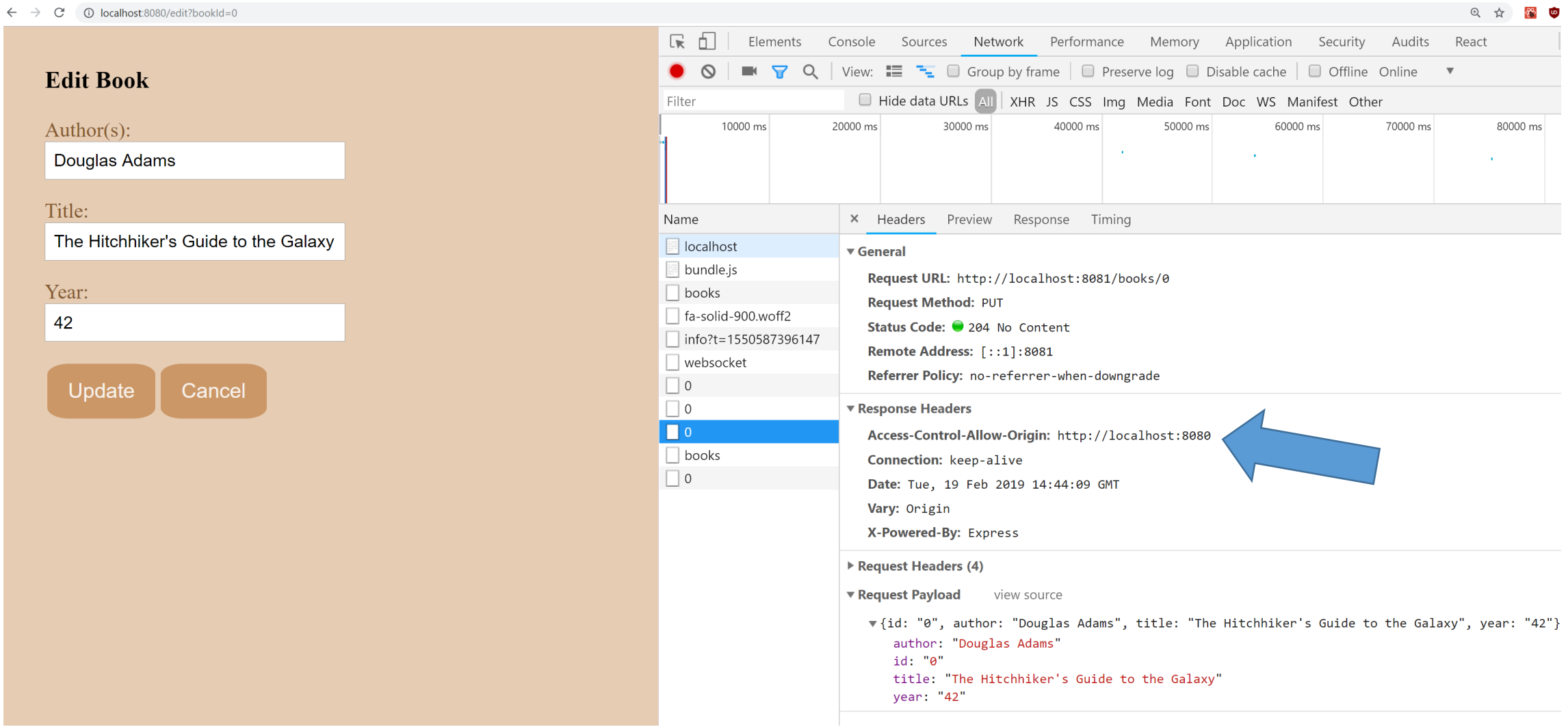
Visit malicious site, with malicious JavaScript automatically run on page load

www.evil.no

CORS and GET

- Although GET requests are not preflighted with OPTIONS, **they can still be secure**
- Server can respond with **Access-Control-Allow-Origin** on any request, including GET, and not just OPTIONS
- If such header does not match the origin, then the browser **will delete the content of the response**, including for example the status code!
  - Ie, HTTP GET will still be made, but JS will not be able to read response

Even if HTTP call is successfully executed, it does not mean JS is allowed to read the response, as it depends if **Origin** is valid



GET and Side-Effects

- GET requests are not preflighted with OPTIONS
- If CORS not matching **Origin**, JS not allowed to read response
  - so, no information leak
- But, the GET request is still made!
- *If side-effects on server, those will still happen regardless of CORS protection!*
  - eg, creation/deletion of resources, like *"GET /api/data?action=delete"*
- **It is PARAMAOUNT to follow HTTP specs, and have GET requests be side-effect free!!!**

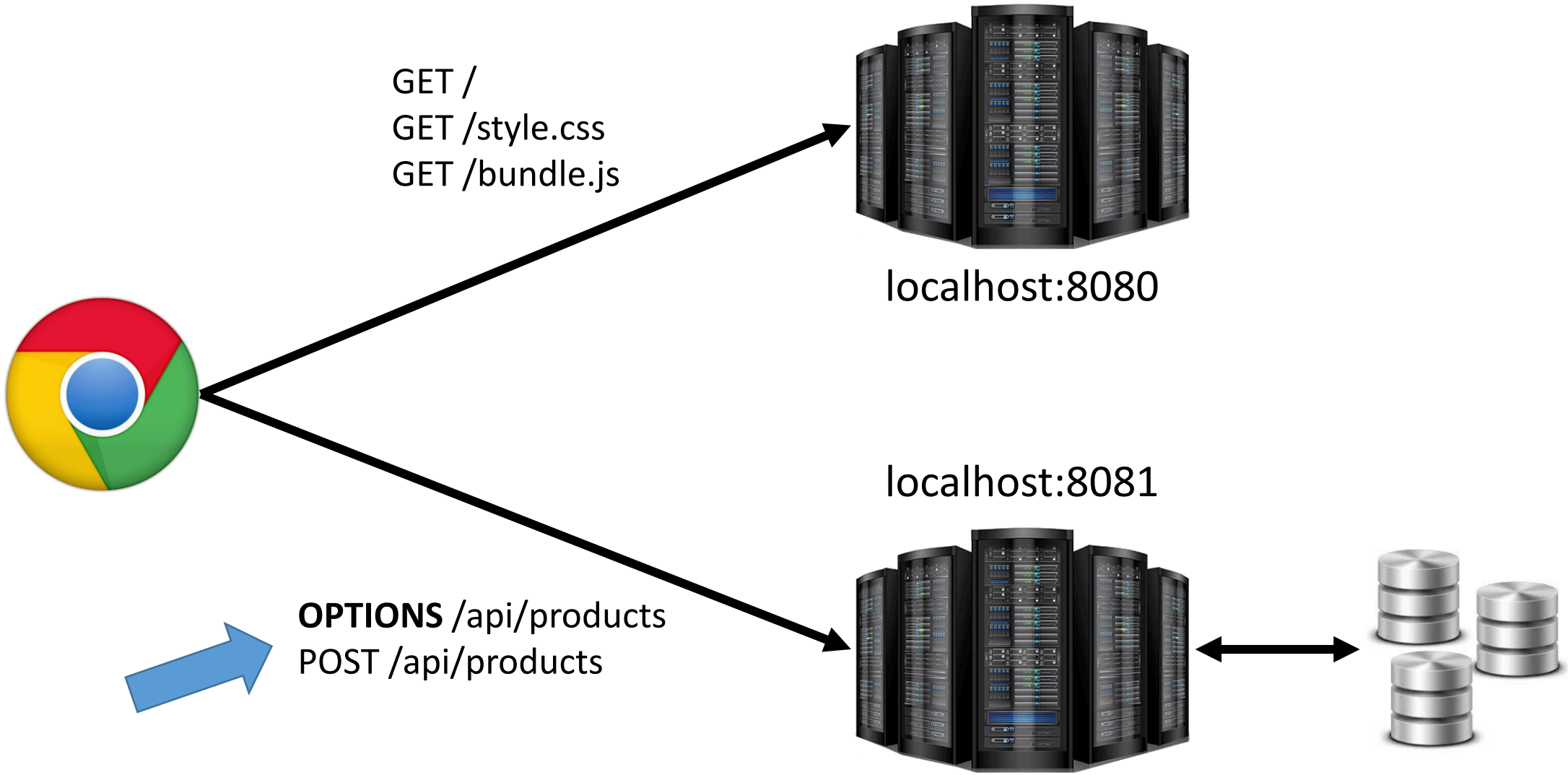
No-Preflighted POST

- This happens for following content-type:
  - **application/x-www-form-urlencoded**
  - **multipart/form-data**
  - **text/plain**
- *In SPAs, if you stick with JSON APIs, you will be "usually" fine*
- Issues when dealing with traditional Server-Sider-Rendering frameworks, as HTML <*form*> requests are not preflighted
  - ie, as typically using **application/x-www-form-urlencoded**
- Solution: **CSRF Tokens**, but we will not need them in this course
  - also the **SameSite** set-cookie option can help here

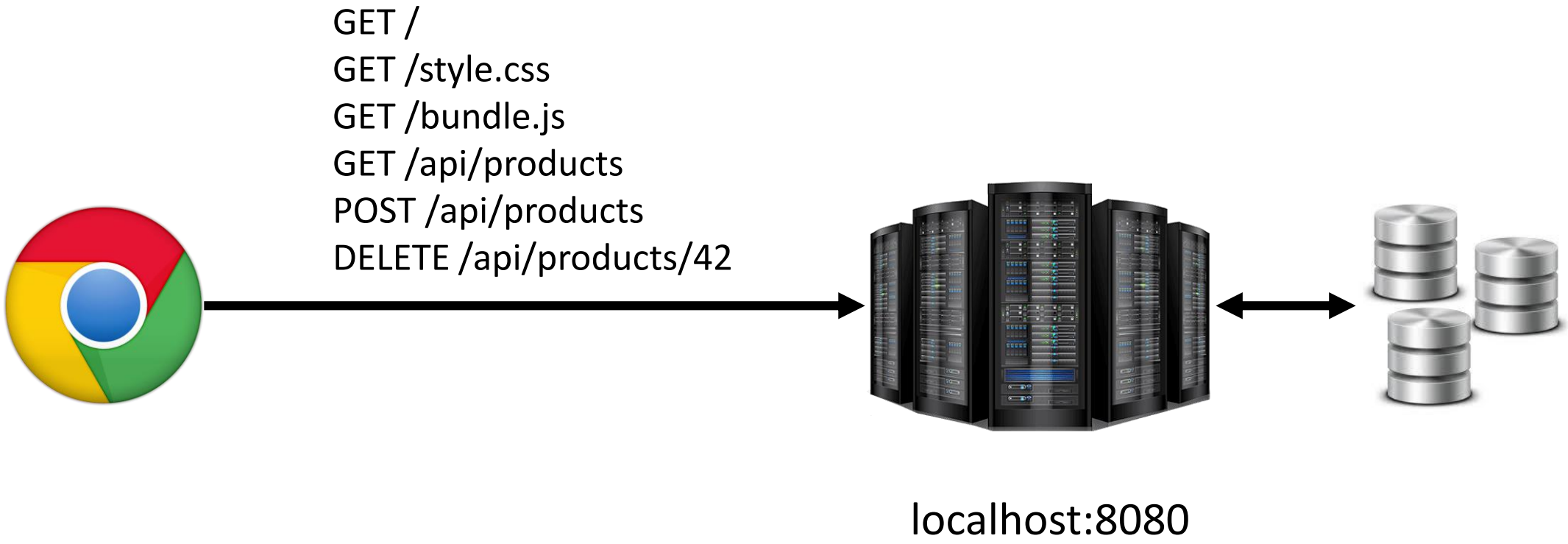
Performance

- Preflighting is not free, as doubling number of HTTP calls
- Caching can be used to save some requests, but problem persists
- Note: do **NOT** have the brilliant idea to pass JSON data with content-type **text/plain**… you will “speed up” performance by bypassing CORS preflight requests, but then making site completely vulnerable to CSRF!!!

- If *frontend* and *backend* servers are separated, you must enable CORS headers on the *backend* responses
- Still performance issues with preflighting

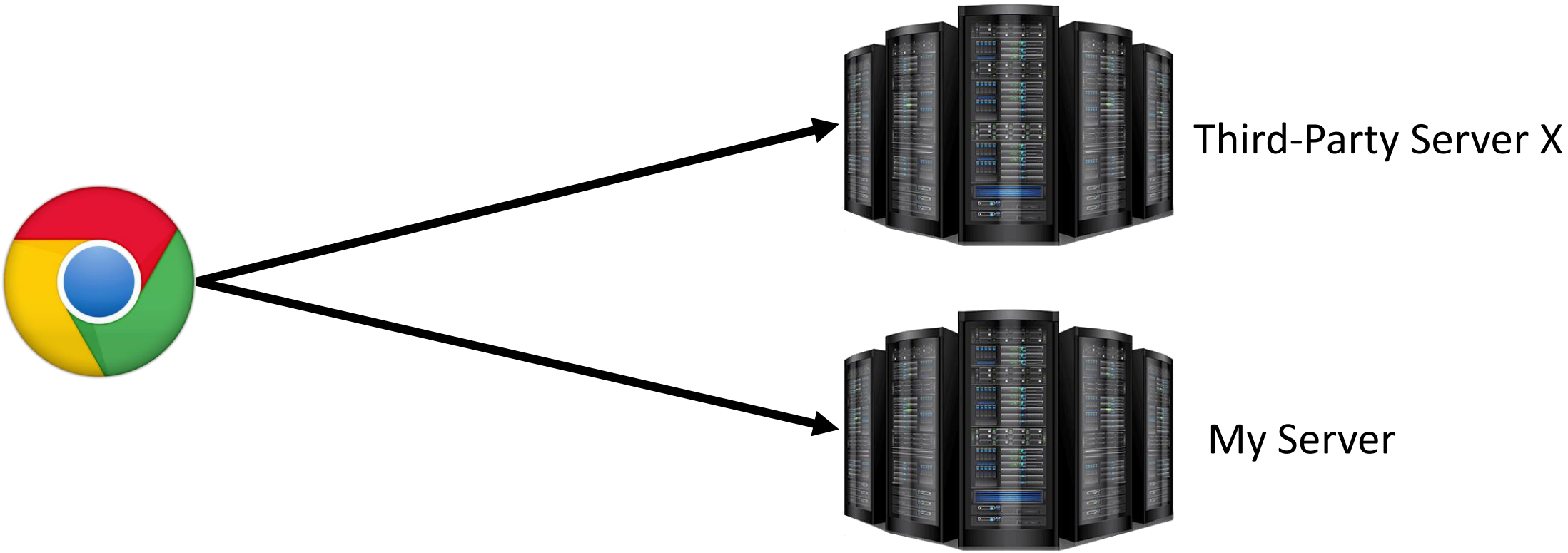


- If everything coming from same **Origin**, then do not enable CORS headers on server, and you should have no problems with CSRF
- Note: could also be different servers behind a single gateway



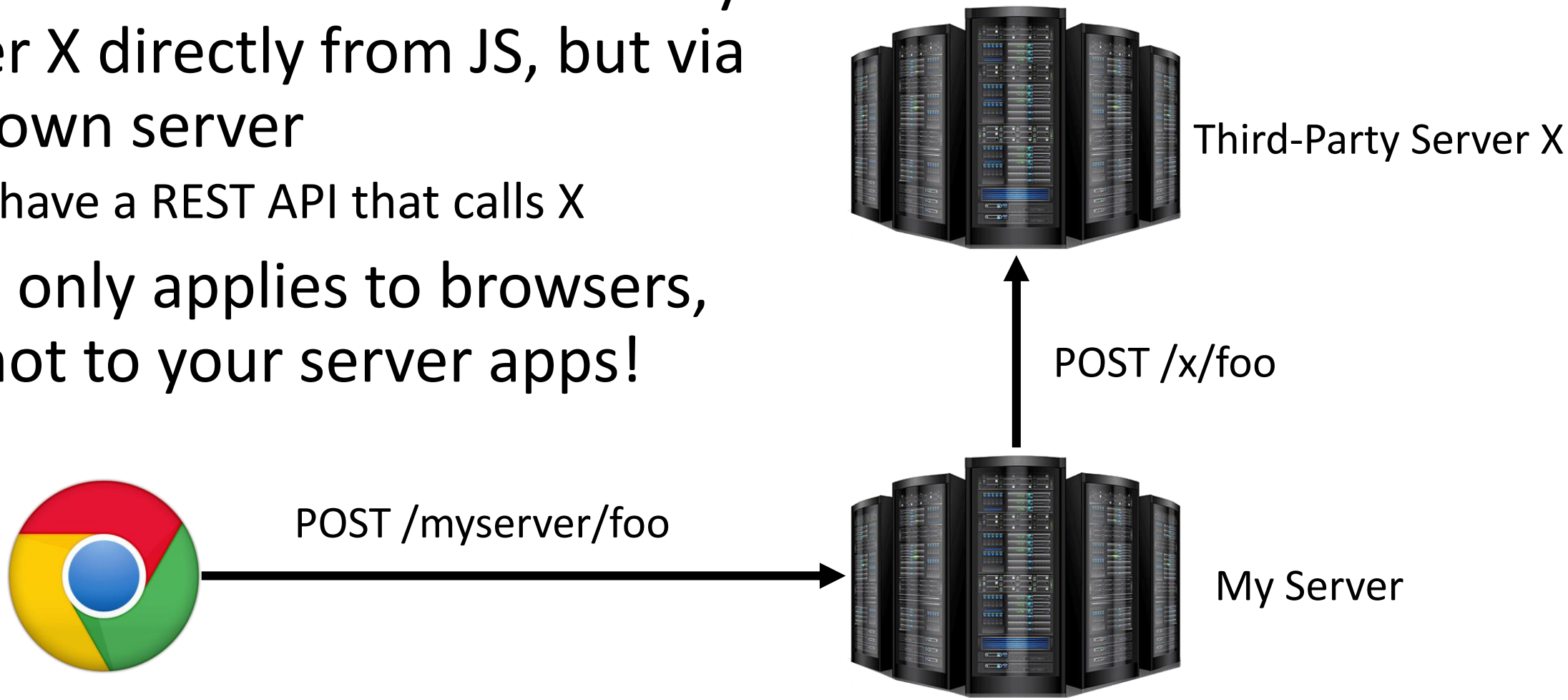
Third-Party APIs

- Still have to enable CORS in those remote servers, if want to contact them directly from JS
- But what if I cannot change those settings, or want to avoid preflight requests?



Proxy Requests

- Option: do not call a Third-Party Server X directly from JS, but via your own server
  - Eg, have a REST API that calls X
- CORS only applies to browsers, and not to your server apps!



Disabling CORS

- People that do not understand CORS can be tempted to disable it by setting "**Access-Control-Allow-Origin: ***" in their servers
  - ie, "*" means all origins are valid
- This "*could*" be fine for read-only services with no sensitive data
- What if need to do *authenticated* requests with cookies?
- Some browsers have "*idiot-proof*" mechanisms that block authenticated requests to servers responding with "**Access-Control-Allow-Origin: ***"
  - ie, it would be pointless to have an auth system if then you disable CORS protection…

SameSite Cookie

- Another option for cookies, besides *Secure* and *HttpOnly*
- Introduced by Chrome in 2016
  - all other major browsers started to support it afterwards
- Explicitly added to fight CSRF attacks
  - and so prevent most of the issues discussed so far

3 Settings

- **None**: send Cookies in CSRs, but only if marked **Secure**
  - ie, need to use HTTPS
- **Lax**: block CSR requests, but allow **<a>** navigation **GET**s
- **Strict**: block all requests but for the same **Origin**

Reasons for **Lax**

- Why not be safe and block everything with **Strict**?
- Assume someone in their webpages has a **<a>** link to a your website
- You want users clicking on such **<a>** to be authenticated if already logged in, and not being redirected to login page
  - which could happen with **Strict**, as no cookie would be included in the GET toward your website
- So, **Lax** is a good compromise between security and usability
  - but remember **NEVER** have side-effects on your GET handlers

2020 Big Changes

- If **SameSite** is missing, Chrome assumes it to be **Lax**
  - other major browsers will/have done the same
- This was a GREAT thing
  - CSR should be denied by default, unless explicitly allowed
  - This made the web more secure
- Issue 0: still need to support old browsers that do not have such feature
- Issue 1: this can break websites relying on cross-origin requests all using the same auth cookie

Blog Posts and Tutorials

- Security is a very complex topic
- Unfortunately, many universities do not cover it, or only superficially
- Result: plenty of resources online written by people with no clue of what they are talking about
- Recommendation: *be wary of this issue, and do not trust blindly when reading about security* (**including these slides…**)

Security in React and NodeJS

Passport

- To enable auth in an Express/NodeJS application, we will use the library called **Passport**
- We will use *session-based auth with cookies*
- We will need to create a REST API to handle *signup*, *login* and *logout* operations
- When HTTP requests with valid session cookie, *Passport* will automatically generate a "*user*" object identifying the logged-in user
- Note: we will **NOT** see how to properly store passwords…

Auth in React

- *React* has no concept of auth
- Still, want to render components based on whether logged-in or not
  - eg, display a "*Login*" button if not logged-in, or a "*Logout*" otherwise
- Whether we are logged in depends if the browser has a valid session cookie
- But *React* (and JS in general) **CANNOT** access a *HttpOnly* cookie
  - and even if it could, would not know if server would still accept such cookie

State for Logged-in

- Could use a variable to represent if user is logged-in
  - with rendering of components based on such variable
- At **each** HTTP call in the whole app requiring auth, update such variable if needed
  - eg, mark as logged-out if we get any 401
- *React* cannot show the current login/logout state, but rather the state at the last HTTP interaction
  - note: even if no direct logout, a cookie can still expire

Frontend “Security”

- In *frontend*, choosing what to display based on login status has **no impact on security**… it impacts only *usability*
- Security MUST be handled on the *backend*… anyone can open a TCP connection and use HTTP to send messages to your *backend* without using the GUI!
- Each REST endpoint that needs protection has to handle auth regardless of *frontend*

Not logged-in?
No button with AJAX for protected resource

My Server

Logged-in?
Render button with AJAX

onClick = POST /api/protected

My Server