Web Development and API Design

Lesson 10: CORS, CSRF and XSS

Prof. Andrea Arcuri

Goals

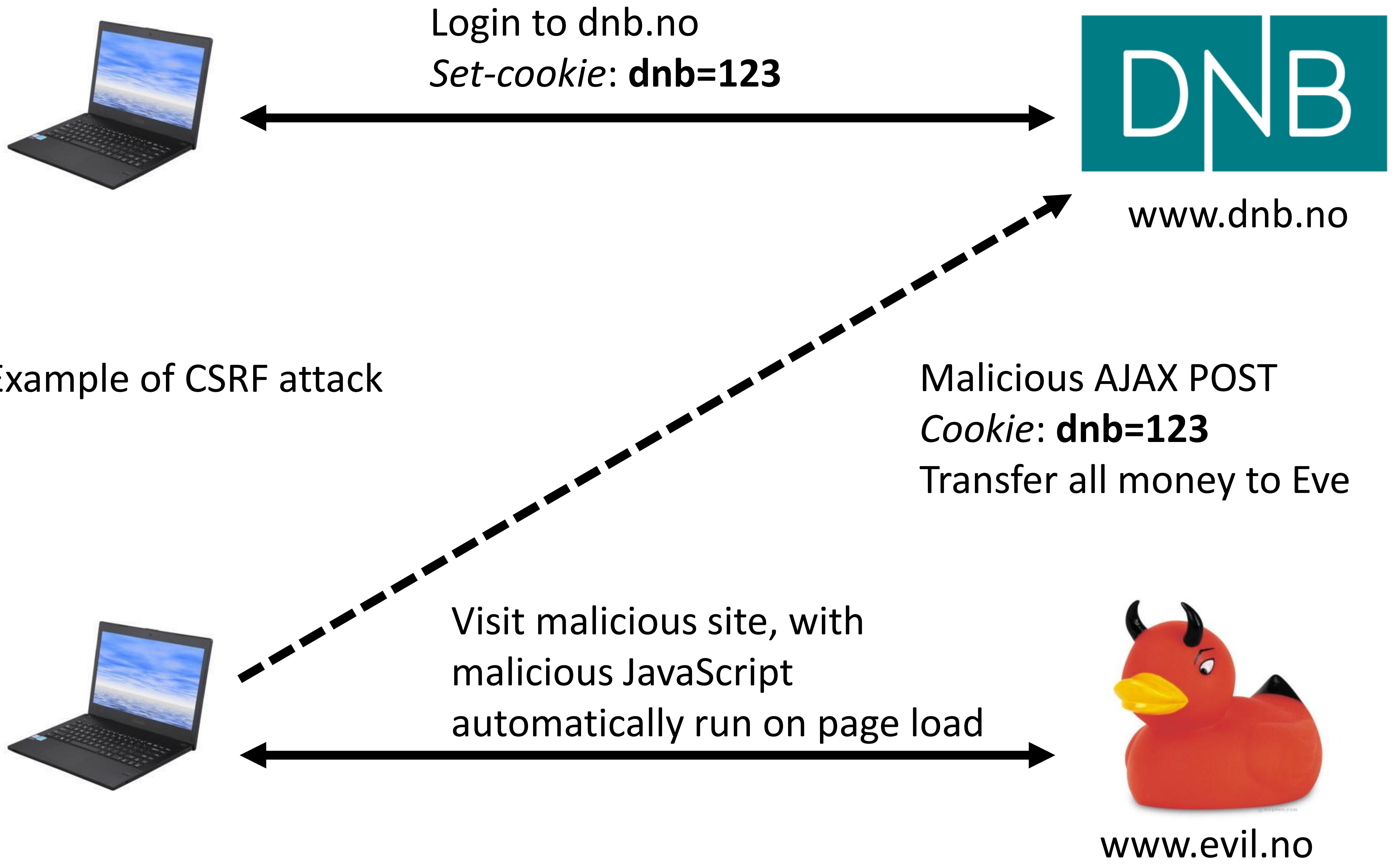
- Understand what *Cross-Origin Resource Sharing* (CORS) is
- Understand the risks of *Cross-Site Request Forgery* (CSRF)
- Revise knowledge on user-input sanitization and escaping
- Understand what XSS attacks are carried out
  - and see how libraries/frameworks like *React* help to prevent some XSS, **but not all!!!**

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

Example of CSRF attack

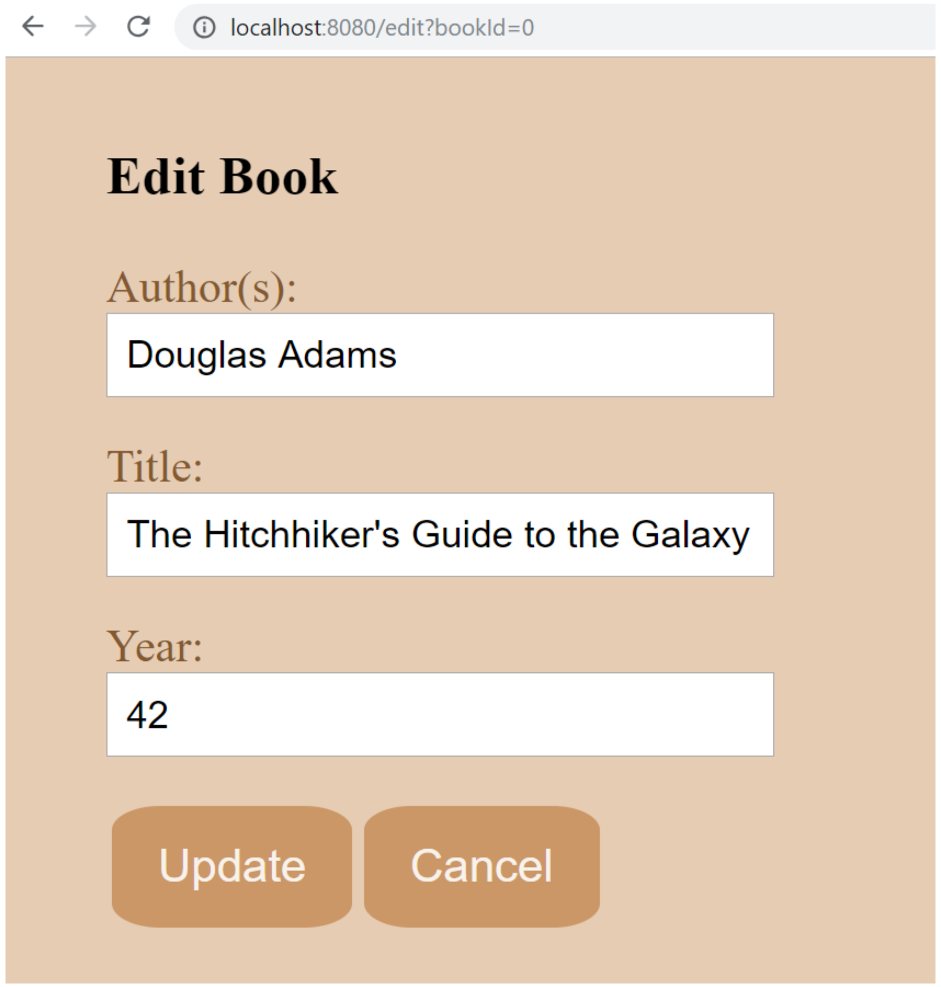

Login to dnb.no
Set-cookie: **dnb=123**
DNB
www.dnb.no
Malicious AJAX POST
Cookie: **dnb=123**
Transfer all money to Eve
Visit malicious site, with
malicious JavaScript
automatically run on page load
www.evil.no

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:808**0***, whereas REST API for backend was on *localhost:808**1***
- 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?


localhost:8080/edit?bookId=0

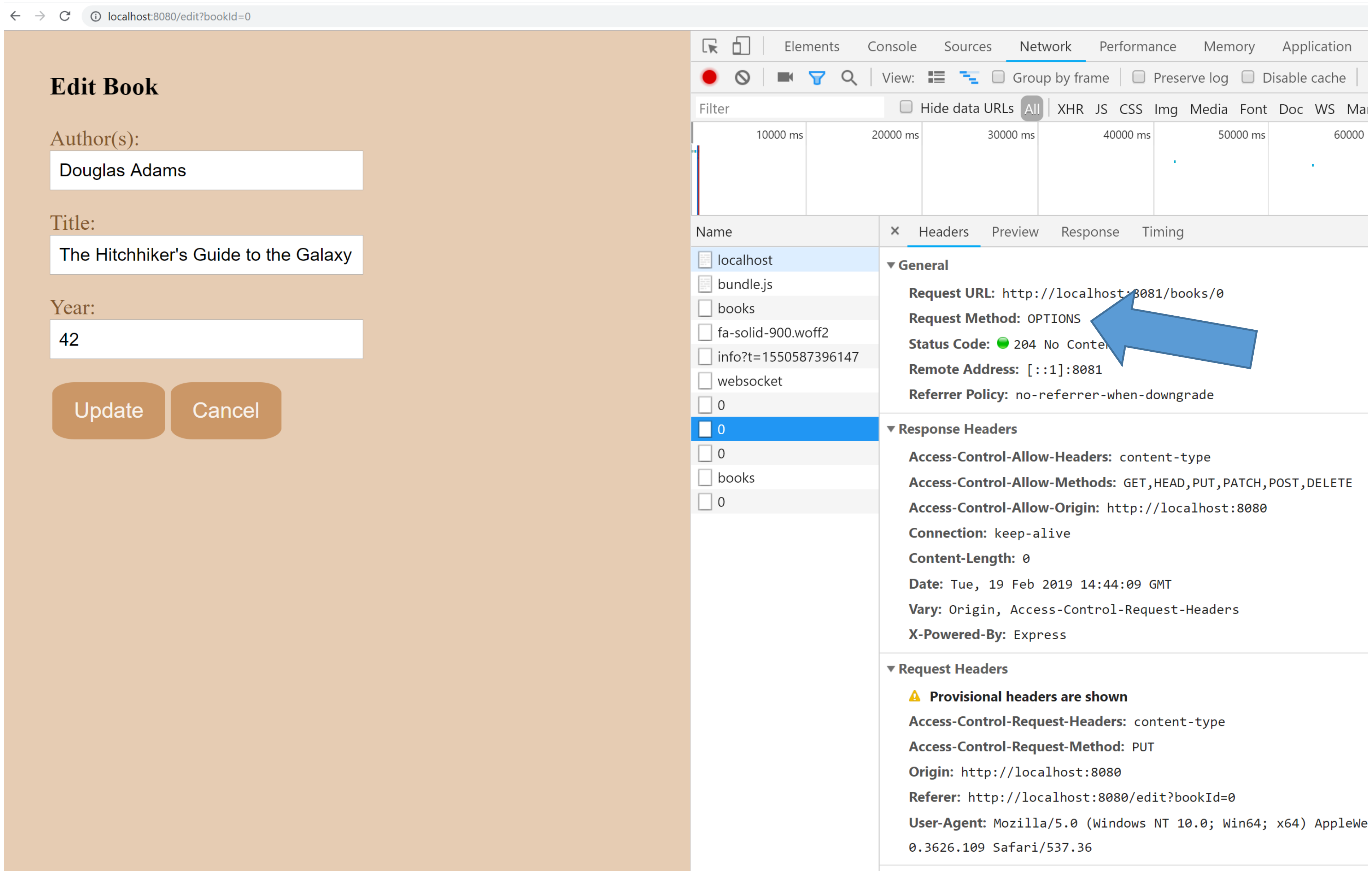
Edit Book

Author(s): Douglas Adams

Title: The Hitchhiker's Guide to the Galaxy

Year: 42

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



Edit Book

Author(s): Douglas Adams

Title: The Hitchhiker's Guide to the Galaxy

Year: 42

Update Cancel

localhost
bundle.js
books
fa-solid-900.woff2
info?t=1550587396147
websocket
0
0
0
books
0

▼ General

```
Request URL: http://localhost:8081/books/0
Request Method: OPTIONS
Status Code: ● 204 No Conte
Remote Address: [::1]:8081
Referrer Policy: no-referrer-when-downgrade
```

▼ Response Headers

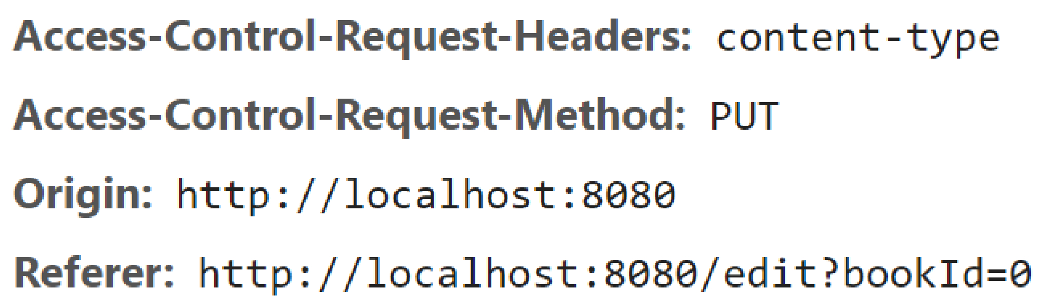
```
Access-Control-Allow-Headers: content-type
Access-Control-Allow-Methods: GET,HEAD,PUT,PATCH,POST,DELETE
Access-Control-Allow-Origin: http://localhost:8080
Connection: keep-alive
Content-Length: 0
Date: Tue, 19 Feb 2019 14:44:09 GMT
Vary: Origin, Access-Control-Request-Headers
X-Powered-By: Express
```

▼ Request Headers

⚠ Provisional headers are shown

```
Access-Control-Request-Headers: content-type
Access-Control-Request-Method: PUT
Origin: http://localhost:8080
Referer: http://localhost:8080/edit?bookId=0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWe
0.3626.109 Safari/537.36
```

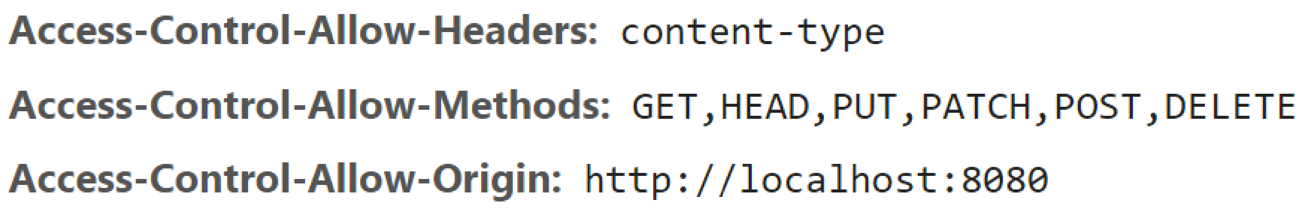
OPTIONS Request Headers



```
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

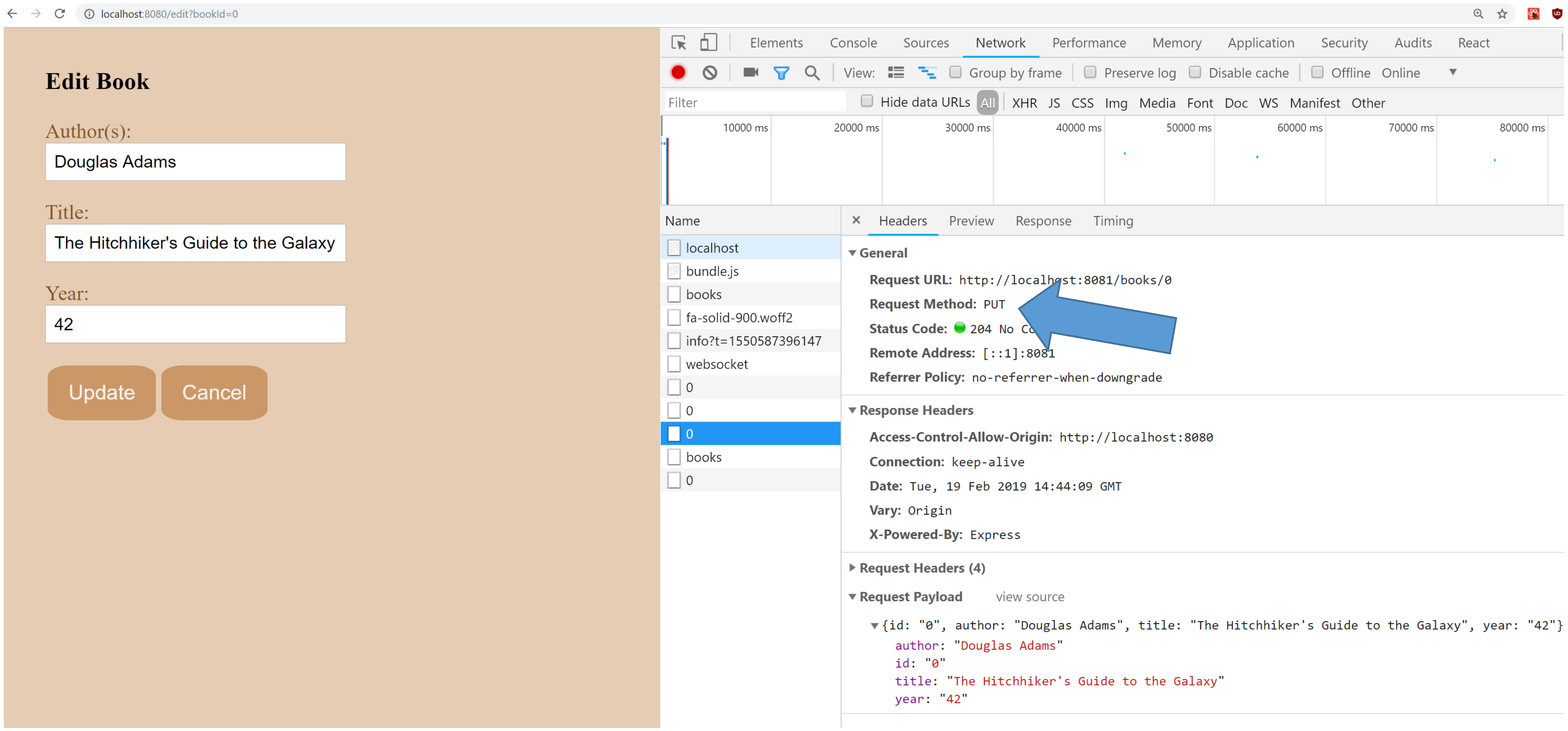
OPTIONS Response Headers



```
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



General

```
Request URL: http://localhost:8081/books/0
Request Method: PUT
Status Code: 204 No Co
Remote Address: [::1]:8081
Referrer Policy: no-referrer-when-downgrade
```

Response Headers

```
Access-Control-Allow-Origin: http://localhost:8080
Connection: keep-alive
Date: Tue, 19 Feb 2019 14:44:09 GMT
Vary: Origin
X-Powered-By: Express
```

Request Headers (4)

Request Payload view source

```
{id: "0", author: "Douglas Adams", title: "The Hitchhiker's Guide to the Galaxy", year: "42"}
  author: "Douglas Adams"
  id: "0"
  title: "The Hitchhiker's Guide to the Galaxy"
  year: "42"
```

A Note on Chrome and Firefox

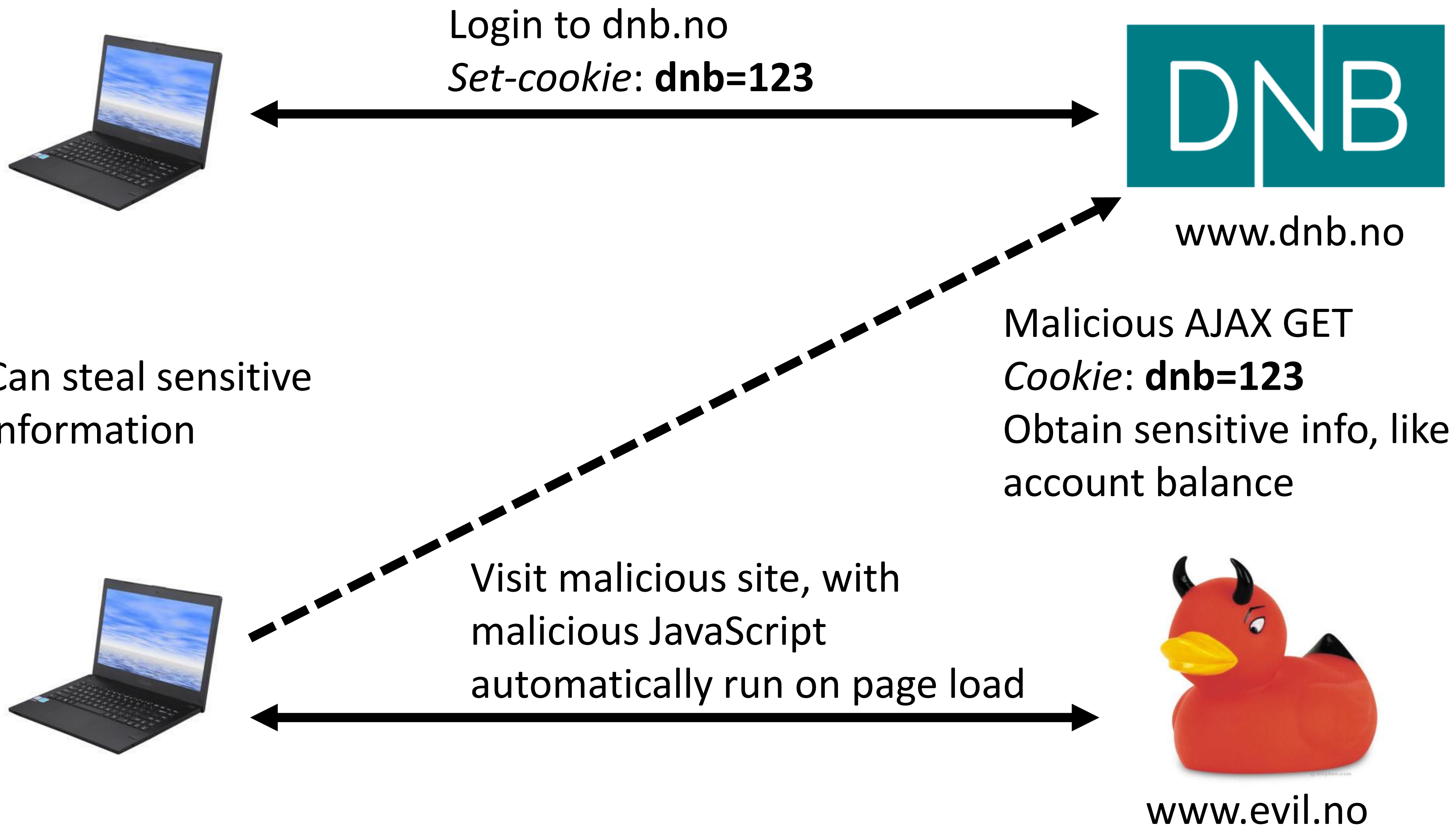
- As of Chrome 79, Developer Tools does NOT show preflight OPTION requests anymore
  - due to technical reasons, might be fixed in a following release
  - the previous screenshots were taken with Chrome < 79

- 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**

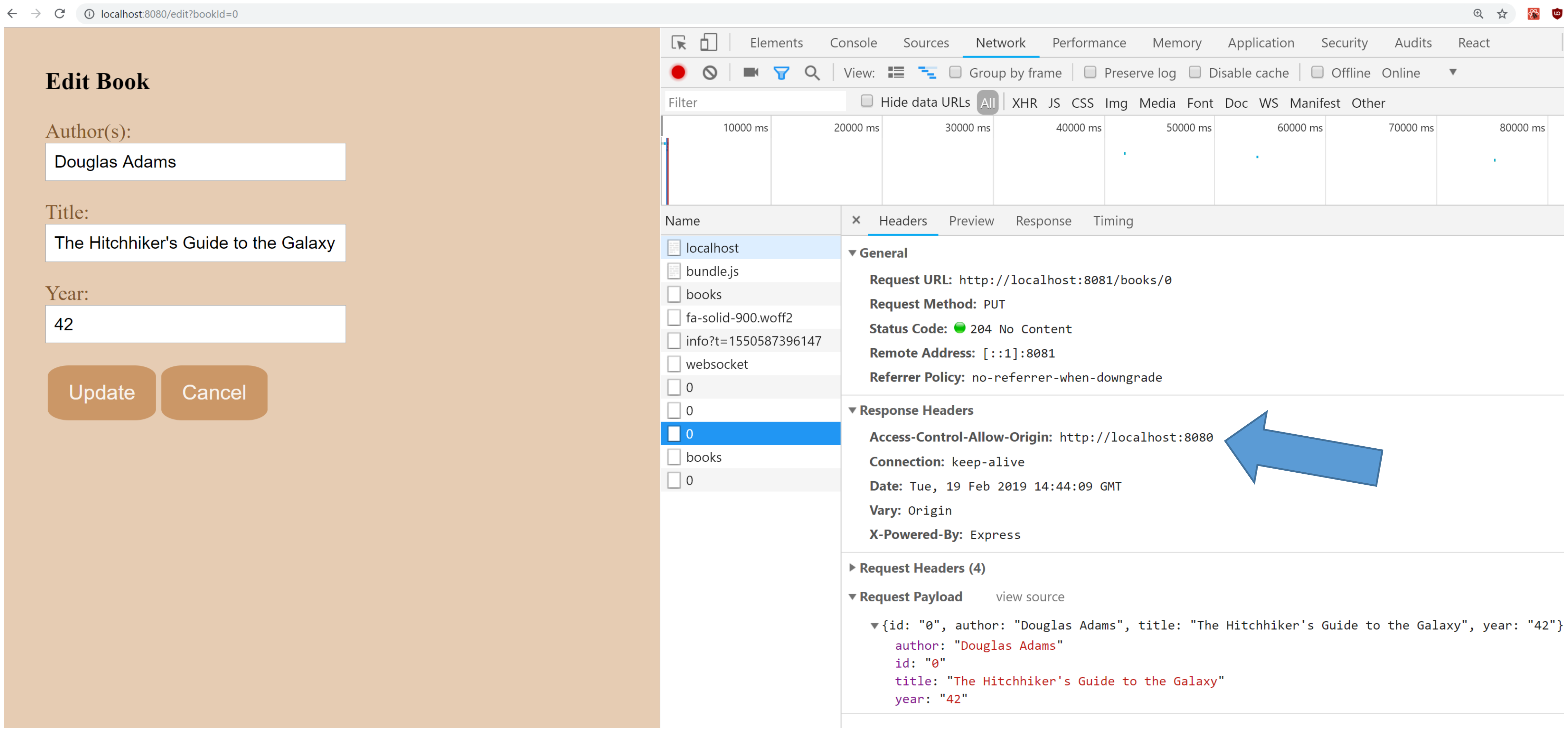
No-Preflight GET


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



General

```
Request URL: http://localhost:8081/books/0
Request Method: PUT
Status Code: 204 No Content
Remote Address: [::1]:8081
Referrer Policy: no-referrer-when-downgrade
```

Response Headers

```
Access-Control-Allow-Origin: http://localhost:8080
Connection: keep-alive
Date: Tue, 19 Feb 2019 14:44:09 GMT
Vary: Origin
X-Powered-By: Express
```

Request Headers (4)

Request Payload view source

```
{id: "0", author: "Douglas Adams", title: "The Hitchhiker's Guide to the Galaxy", year: "42"}
  author: "Douglas Adams"
  id: "0"
  title: "The Hitchhiker's Guide to the Galaxy"
  year: "42"
```

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!!!**

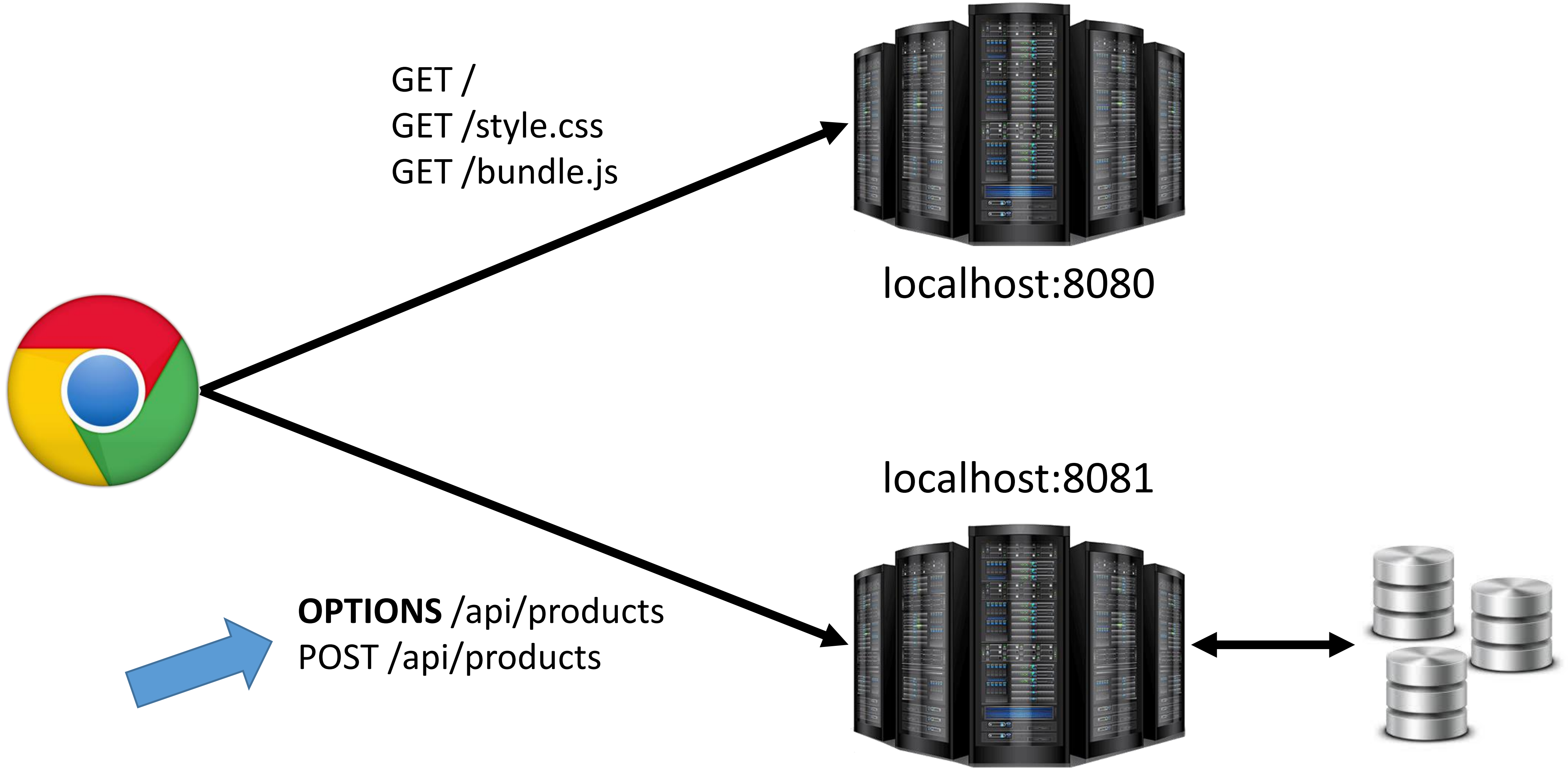
No-Preflighted POST

- 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

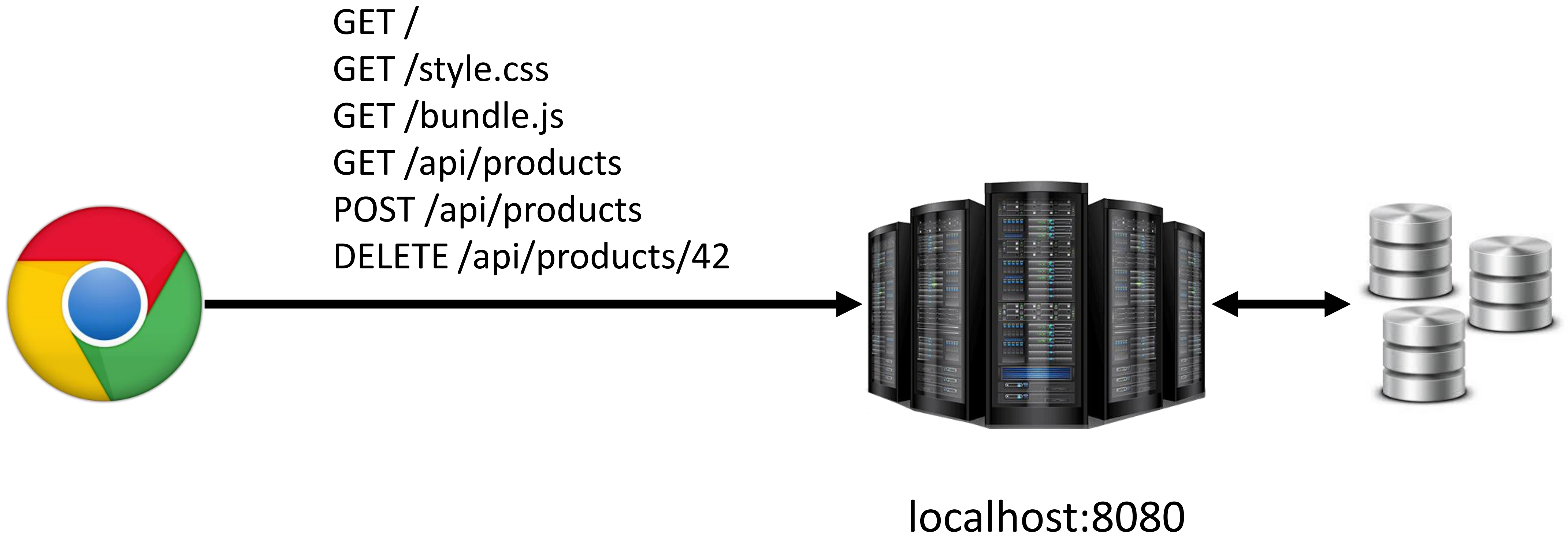
Performance

- 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

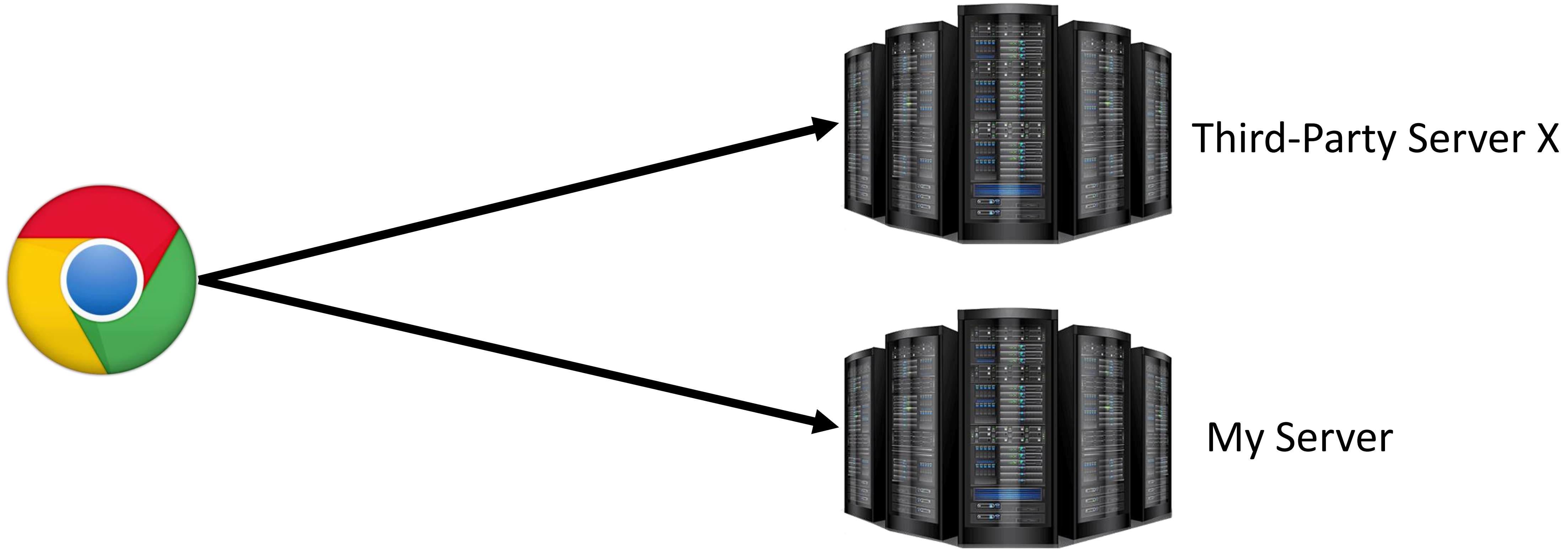


- If everything coming from same **Origin**, then do not enable CORS headers on server, and you should have no problems with CSRF
- Note: could also be different servers behind a single gateway



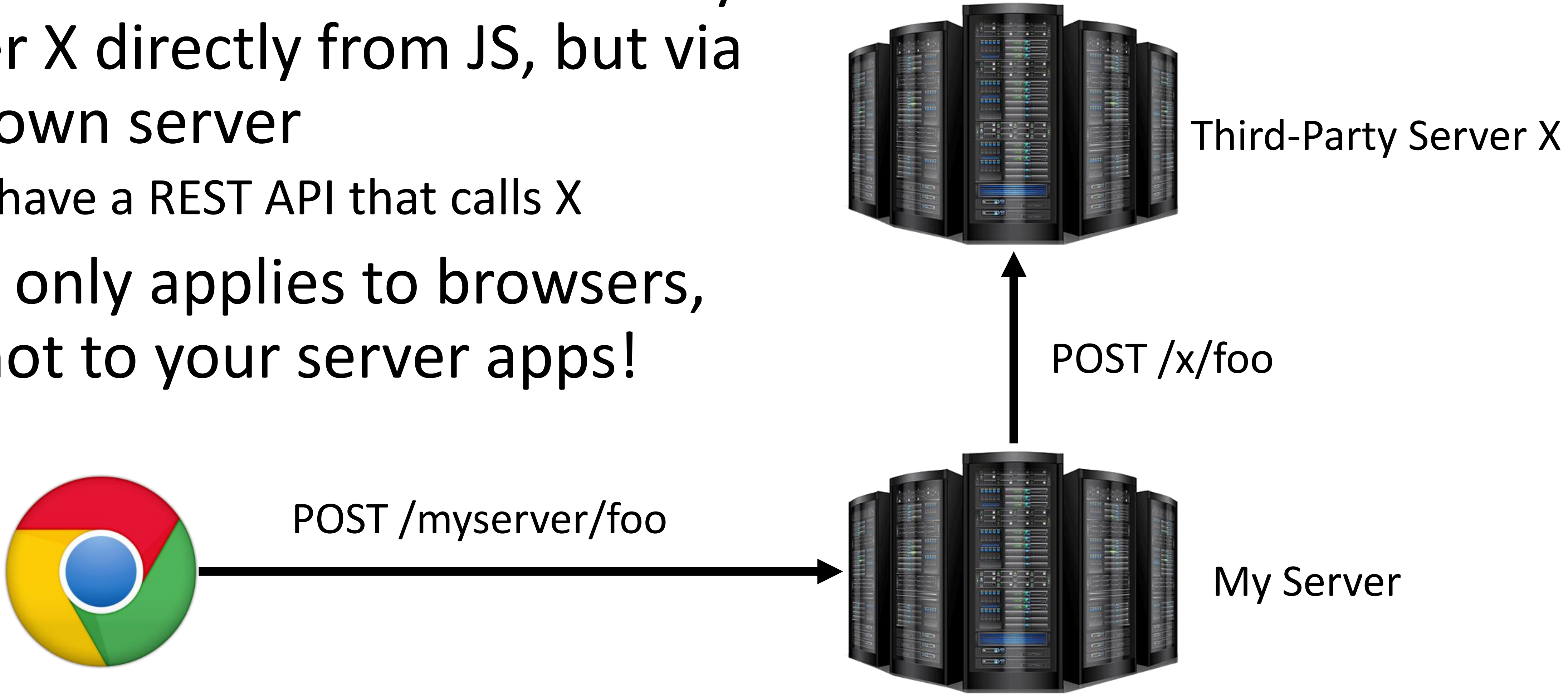
Third-Party APIs

- 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

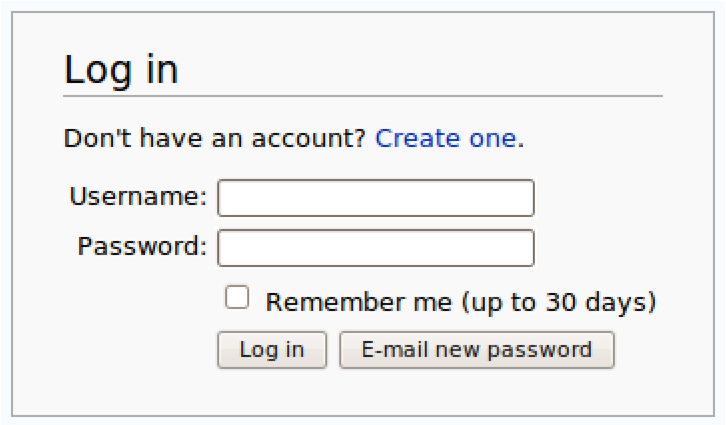
- If **SameSite** is missing, Chrome assumes it to be **Lax**
  - other major browsers will/have done the same
- This was a GREAT thing
  - CSR should be denied by default, unless explicitly allowed
  - This made the web more secure
- Issue 0: still need to support old browsers that do not have such 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…**)

Data Escaping/Sanitization

HTML Form Data


Log in
Don't have an account? Create one.
Username:
Password:
Remember me (up to 30 days)
Log in
E-mail new password

- How is data sent in a HTML Form?
- What is the structure of payload of the HTTP POST request?
- JSON? eg *{“username”:“foo”, “password”:123}*
- XML? eg *<data><username>foo</username><password>123</password></data>*

x-www-form-urlencoded

- For textual data, like inputs in a HTML form
  - For binary data like file uploads, can use *multipart/form-data*
- Old format which is part of the HTML specs
  - *https://www.w3.org/TR/html/sec-forms.html#urlencoded-form-data*
- Each form element is represented with a pair *<name>**=**<value>*, where each pair is separated by a **&**
- Eg.: *username=foo**&**password=123*

What if values contain “**=**” or “**&**”?

- Eg, password: “123&bar=7”
- (Wrong) result: username=foo&**password=123**&*bar=7*
- The “*bar=7*” would be wrongly treated as a third input variable called “*bar*” with value “*7*”, and not be part of the “password” value

Solution: Special Encoding

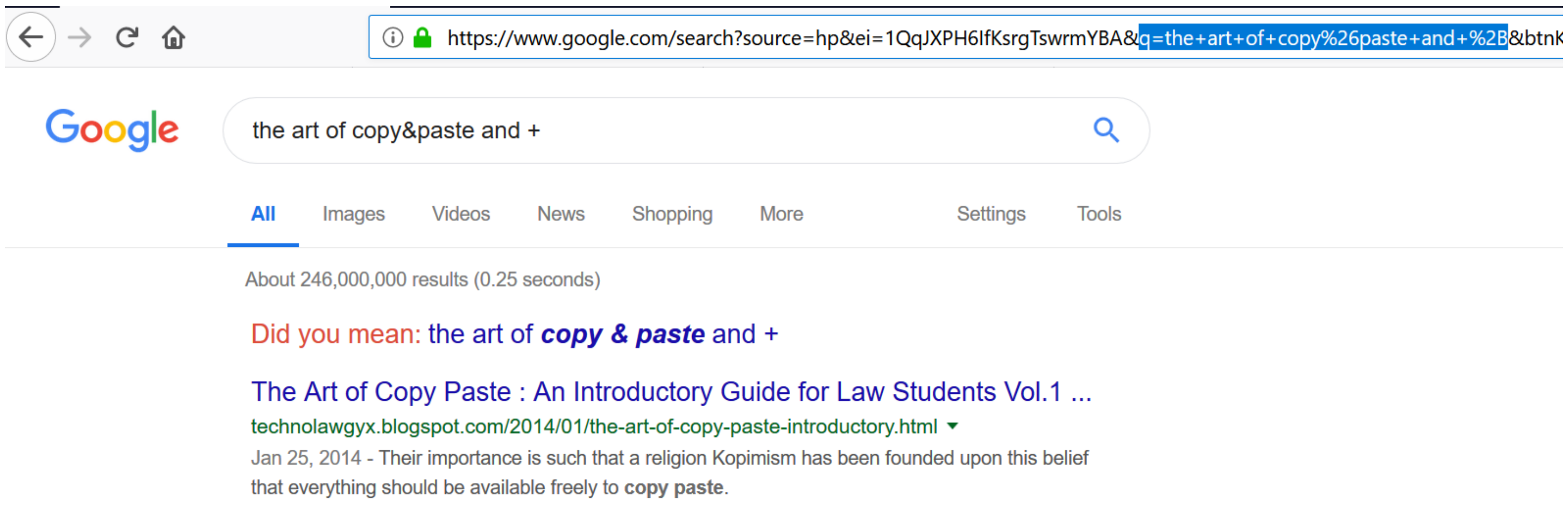
- Stay same: “*”, “-”, “.”, “_”, 0-9, a-z, A-Z
- Space “ ” becomes a “+”
- The rest become “%HH”, a percent sign and two hexadecimal digits representing the code of the character (default UTF-8)
- So, “123&bar=7” becomes “123**%26**bar**%3D**7”
- %26 = (2*16)+6 = 38, which is the code for **&** in ASCII
- %3D = (3*16)+13 = 61, which is the code for **=** in ASCII
  - Recall, hexadecimal D=13 (A=10,…, F=15)

But…

- What if I have a “%” in my values? Would not that mess up the decoding?
- E.g, password=“%3D”, don’t want to be wrongly treated as a “=”
- Not an issue, as symbol “%” is encoded based on its ASCII code 37, ie “*%25*3D”
  - %25 = (2*16)+5 = 37

URLs and Query Parameters

- Query parameters in a URL are sequences of *<key>=<value>* pairs, separated by the symbol **&**
- What if a key or a value need to use special symbols like **=** or **&**?
- Those will be escaped as well, using the same kind of *%HH* escaping used in HTML forms
  - one difference though: “ ” empty char will be replaced with a “+”, whereas the symbol “+” is escaped with %2B
  - %2B = (2*16) + 11 = 43 , which is the ASCII code for +


https://www.google.com/search?source=hp&ei=1QqJXPH6IfKsrgTswrmYBA&q=the+art+of+copy%26paste+and+%2B&btnk

the art of copy&paste and +

About 246,000,000 results (0.25 seconds)

Did you mean: the art of ***copy & paste*** and +

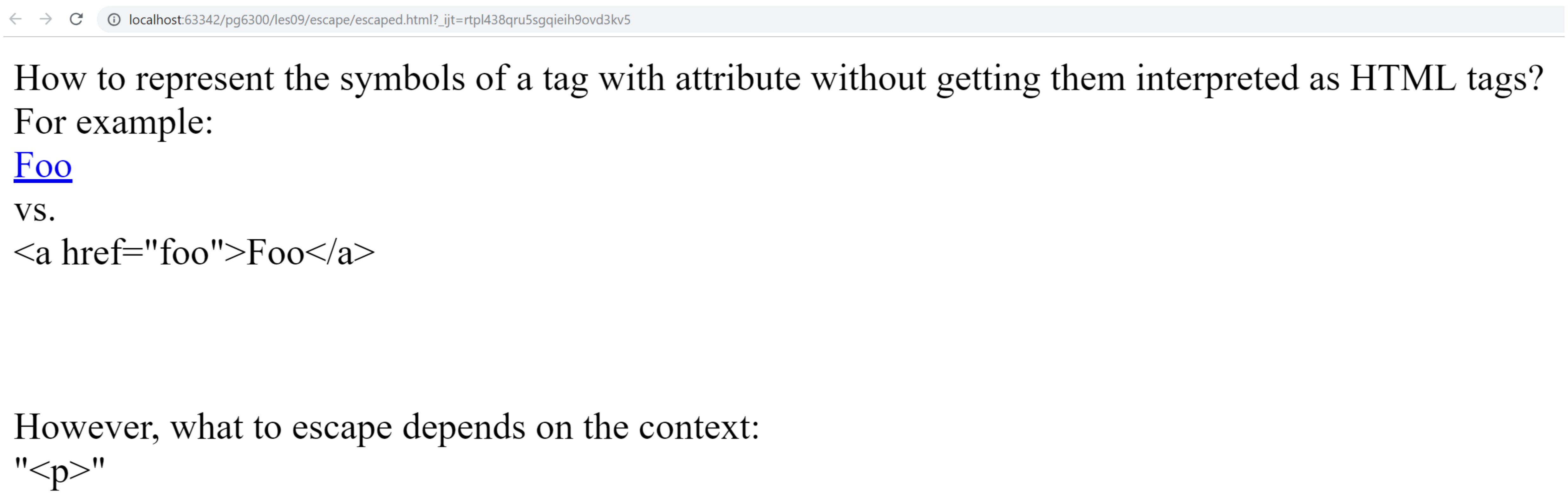
The Art of Copy Paste : An Introductory Guide for Law Students Vol.1 ...
technolawgyx.blogspot.com/2014/01/the-art-of-copy-paste-introductory.html
Jan 25, 2014 - Their importance is such that a religion Kopimism has been founded upon this belief that everything should be available freely to **copy paste**.

- Assume in Google you search for "*the art of copy&paste and +*"
- The browser will make a GET request with query parameters, including the pair: q=the**+**art**+**of**+**copy**%26**paste**+**and**+%2B**
- Notice how empty spaces are replaced with **+**, & with **%26**, and + with **%2B**

Text Transformations

- We can represent text in various formats, eg, HTML, XML, JSON, *x-www-form-urlencoded*
- Such formats use special symbols to define *structures* of the document
  - eg = and & in HTML form data, and <> in HTML/XML documents
- Input text values should NOT use those special structure/syntax symbols
- Need to be *transformed* (aka *escaped*) into non-structure symbols
  - & into %26, and = into %3D in HTML form data

What About HTML???



localhost:63342/pg6300/les09/escape/escaped.html?_ijt=rtpl438qru5sgqieih9ovd3kv5

How to represent the symbols of a tag with attribute without getting them interpreted as HTML tags?
For example:
Foo
vs.
<a href="foo">Foo</a>

However, what to escape depends on the context:
"<p>"

HTML/XML Escaping

- “**&**” followed by name (or code), closed by “**;**”
- **&quot;** for **“** (double quotation mark)
- **&amp;** for **&** (ampersand)
- **&apos;** for **‘** (apostrophe)
- **&lt;** for **<** (less-than)
- **&gt;** for **>** (greater-than)
- These are most common ones

See “escaped.html” file

```
<a href="foo">Foo</a>
```

vs.

```
&lt;a href=&quot;foo&quot;&gt;Foo&lt;/a&gt;
```

What actually needs to be escaped depends on context

- `<div  id="&quot;<p>&quot;">`
  `"&lt;p&gt;"`
  `</div>`
- Representing **“<p>”** (quotes included)
- In attributes, quotes “ need to be escaped (**&quot;**), but no need there for **<>**, as those latter are no string delimiters
- In node content, it is the other way round

XSS

User Content

- Text written by user which is displayed in the HTML pages when submitted (eg HTML form)
  - eg, Chats and Discussion Forums
  - but also showing back the search query when doing a search
- Also query parameters in URLs are a form of user input if crafted by an attacker
  - eg, *www.foo.com?x=10* if then value of *x* is displayed in the HTML
  - recall, attacker can use social engineering to trick a user to click on a link
- *What is the most important rule regarding user content given as input to a system???*

NEVER TRUST USER INPUTS!!!

NEVER

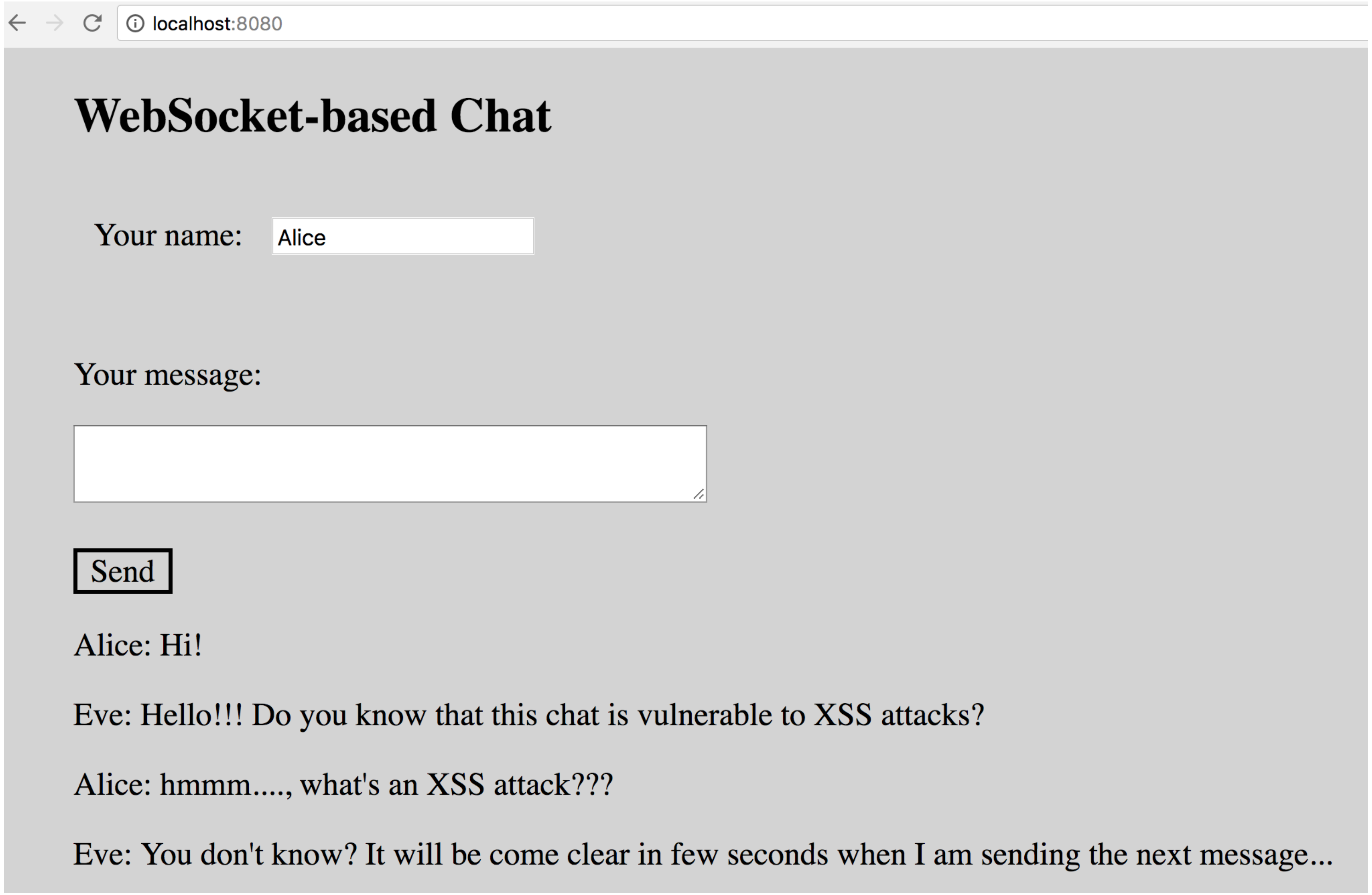
TRUST

USER

INPUTS!!!

NEVER TRUST USER INPUTS!!!

But Why???



WebSocket-based Chat

Your name: Alice

Your message:

Send

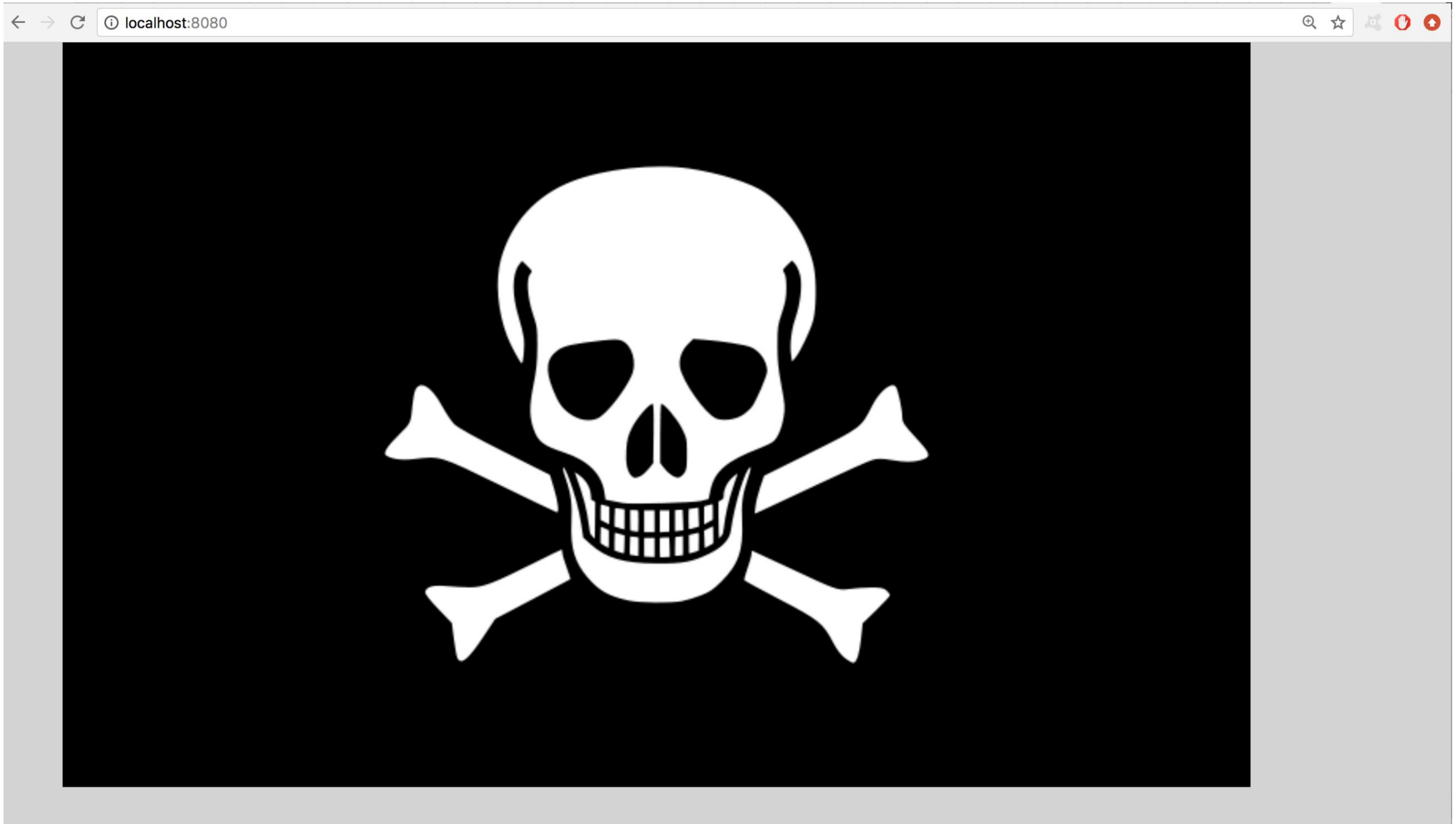
Alice: Hi!

Eve: Hello!!! Do you know that this chat is vulnerable to XSS attacks?

Alice: hmmm...., what's an XSS attack???

Eve: You don't know? It will be come clear in few seconds when I am sending the next message...

After Eve’s message, chat program is gone on Alice’s browser…



What was the problem?

```
let msgDiv = "<div>";

for(let i=0; i<messages.length; i++){
    const m = messages[i];
    //WARNING: this is exploitable by XSS!!!
    msgDiv += "<p>" + m.author + ": " + m.text + "</p>";
}
msgDiv += "</div>";
```

And the message sent was…

```
<img src='x'
   onerror="document.getElementsByTagName('body')[0].inne
rHTML = &quot;<img
src='https://upload.wikimedia.org/wikipedia/commons/thumb
/6/6c/Pirate_Flag.svg/750px-Pirate_Flag.svg.png'/>&quot;;" />
```

String Concatenation

- **msgDiv** += **"<p>"** + dto.**author** + **": "** + dto.**text** + **"</p>"**;
- Should **NEVER** concatenate strings directly to generate HTML when such data comes from user
    - ie, that is a very, very bad example of handling user inputs
- If data is not escaped, could have HTML <tags> that are interpreted by browser as HTML commands
- Could execute JavaScript!!! And so do whatever you want on a page
- Eg., *dto.text = "<script>…</script>"*

Cross-site Scripting (XSS)

- Type of attack in which malicious JavaScript is injected into a web page
- One of the most common type of security vulnerability on the web
- Typically exploiting lack of escaping/sanitization of user inputs when generating HTML dynamically (both client and server side)
- XSS is particularly nasty, as it adds JavaScript in the current page… so CORS will not help you here

Browser Security

- Most browsers will not execute any *<script>* block that has been dynamically added to the page
  - eg, when changing the HTML by altering “*innerHTML*”
- But that is simply futile… because you can still create HTML tags with JS handlers that are executed immediately
- *<img src=’aURLthatNotExist'* ***onerror=***“*… JS here…*”>

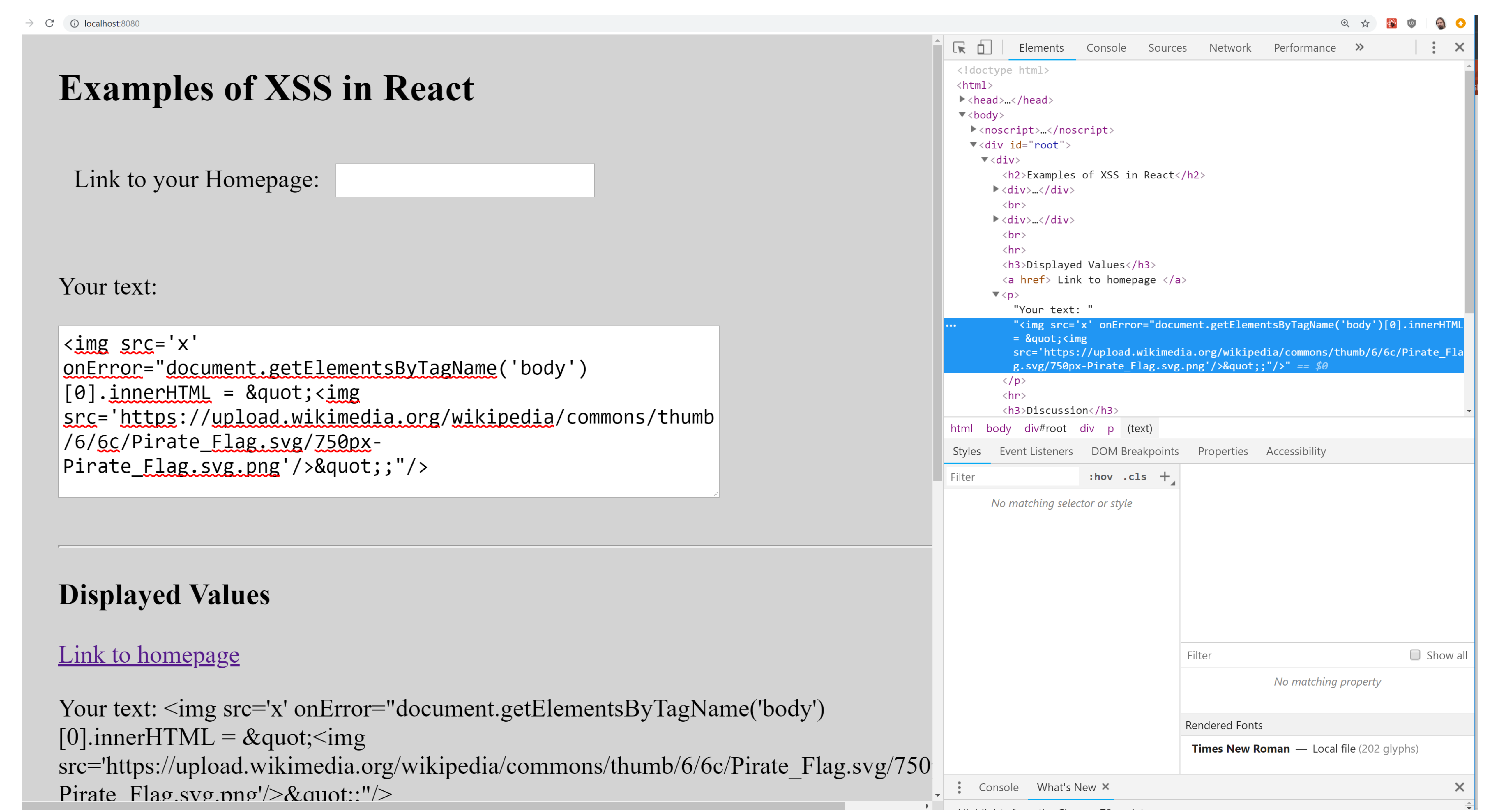
What To Do?

- When dealing with user inputs, always need to escape/sanitize them before use
- This applies both client-side (JS) and server-side (Java, PHP, C#, etc.)
- There are many edge cases, so must use an *existing* library to sanitize the inputs
  - This will depend on the programming language and framework
  - Do NOT write your own escape/sanitize functions

XSS and React

React Sanitization

- XSS is such a huge problem that many libraries/frameworks for HTML DOM manipulation do some form of input sanitization by default
- E.g., consider in JSX: **<p>Your text: {this.state.userInput}</p>**
- … and the **userInput** is **<a>**
- … then, React will *automatically* change it into **&lt;a&gt;** when rendering the HTML
- So, any **<** or **>** in the value will not be interpreted as an HTML tag



Examples of XSS in React

Link to your Homepage:

Your text:

```
<img src='x'
onError="document.getElementsByTagName('body')
[0].innerHTML = &quot;<img
src='https://upload.wikimedia.org/wikipedia/commons/thumb
/6/6c/Pirate_Flag.svg/750px-
Pirate_Flag.svg.png'/>&quot;;"/>
```

Displayed Values

Link to homepage

Your text: <img src='x' onError="document.getElementsByTagName('body')
[0].innerHTML = &quot;<img
src='https://upload.wikimedia.org/wikipedia/commons/thumb/6/6c/Pirate_Flag.svg/750
Pirate_Flag.svg.png'/>&quot;;"/>

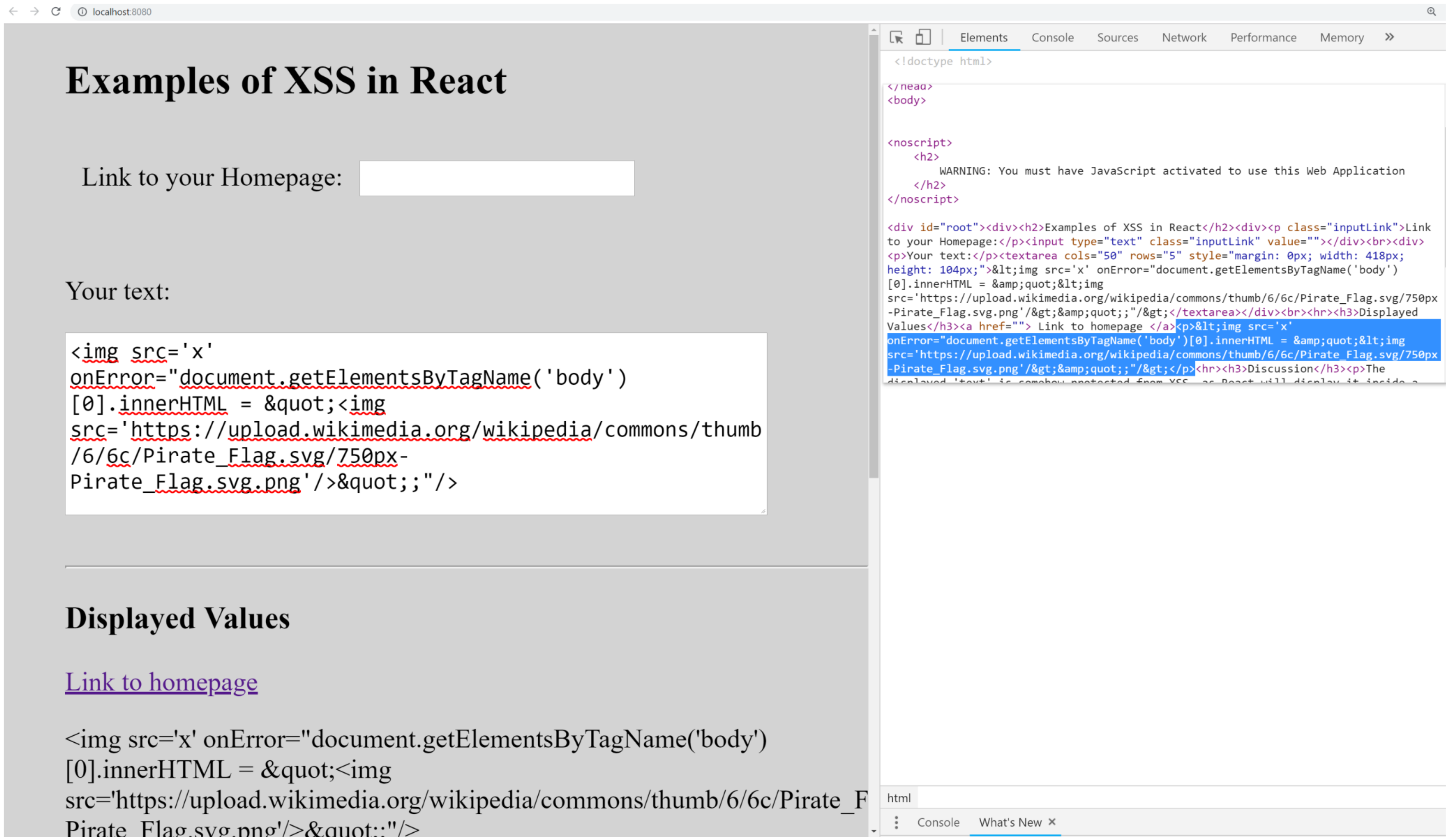
```
<!doctype html>
<html>
▶<head>…</head>
▼<body>
  ▶<noscript>…</noscript>
  ▼<div id="root">
    ▼<div>
       <h2>Examples of XSS in React</h2>
      ▶<div>…</div>
       <br>
      ▶<div>…</div>
       <br>
       <hr>
       <h3>Displayed Values</h3>
       <a href> Link to homepage </a>
      ▼<p>
         "Your text: "
         "<img src='x' onError="document.getElementsByTagName('body')[0].innerHTML
         = &quot;<img
         src='https://upload.wikimedia.org/wikipedia/commons/thumb/6/6c/Pirate_Fla
         g.svg/750px-Pirate_Flag.svg.png'/>&quot;;"/>" == $0
       </p>
       <hr>
       <h3>Discussion</h3>
```

html body div#root div p (text)

Rendered Fonts

Times New Roman — Local file (202 glyphs)

Note: CDT does not show you *raw* HTML by default, but you can see it by clicking for example "*Edit as HTML*"



localhost:8080

Examples of XSS in React

Link to your Homepage:

Your text:

```
<img src='x'
onError="document.getElementsByTagName('body')
[0].innerHTML = &quot;<img
src='https://upload.wikimedia.org/wikipedia/commons/thumb
/6/6c/Pirate_Flag.svg/750px-
Pirate_Flag.svg.png'/>&quot;;"/>
```

Displayed Values

Link to homepage

<img src='x' onError="document.getElementsByTagName('body')
[0].innerHTML = &quot;<img
src='https://upload.wikimedia.org/wikipedia/commons/thumb/6/6c/Pirate_F
Pirate_Flag.svg.png'/>&quot;;"/>

```
<!doctype html>

</head>
<body>


<noscript>
    <h2>
        WARNING: You must have JavaScript activated to use this Web Application
    </h2>
</noscript>

<div id="root"><div><h2>Examples of XSS in React</h2><div><p class="inputLink">Link
to your Homepage:</p><input type="text" class="inputLink" value=""></div><br><div>
<p>Your text:</p><textarea cols="50" rows="5" style="margin: 0px; width: 418px;
height: 104px;">&lt;img src='x' onError="document.getElementsByTagName('body')
[0].innerHTML = &amp;quot;&lt;img
src='https://upload.wikimedia.org/wikipedia/commons/thumb/6/6c/Pirate_Flag.svg/750px
-Pirate_Flag.svg.png'/&gt;&amp;quot;;"/&gt;</textarea></div><br><hr><h3>Displayed
Values</h3><a href=""> Link to homepage </a><p>&lt;img src='x'
onError="document.getElementsByTagName('body')[0].innerHTML = &amp;quot;&lt;img
src='https://upload.wikimedia.org/wikipedia/commons/thumb/6/6c/Pirate_Flag.svg/750px
-Pirate_Flag.svg.png'/&gt;&amp;quot;;"/&gt;</p><hr><h3>Discussion</h3><p>The
displayed 'text' is somehow protected from XSS, as React will display it inside a
```

So, are you safe from XSS when using React???

NO!!!

NO!!!

dangerouslySetInnerHTML

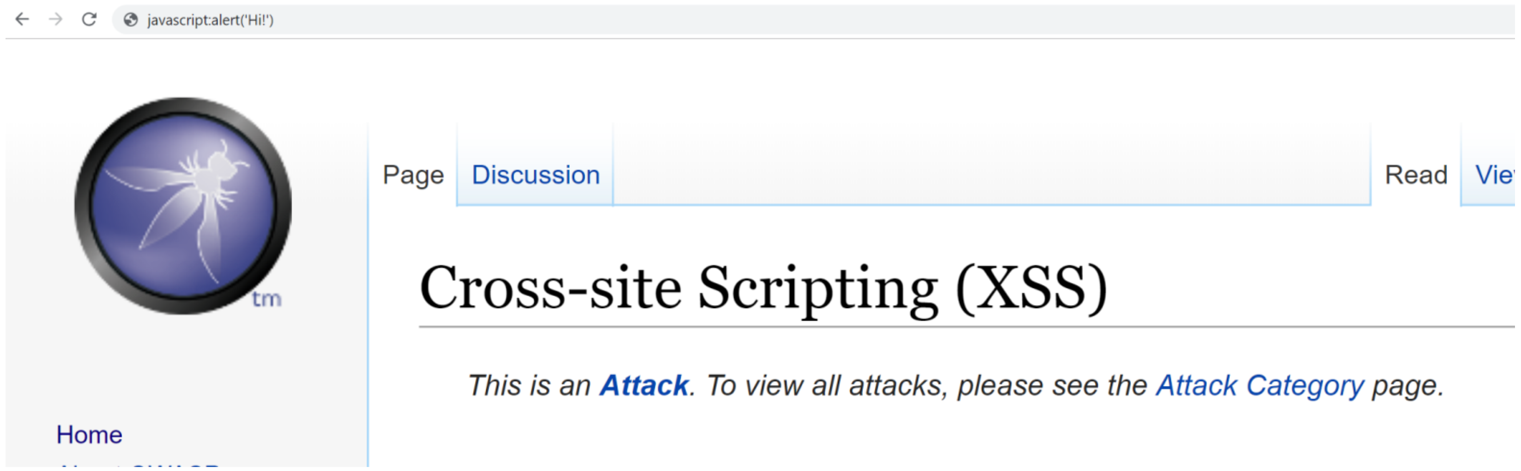
- React components have an attribute called **dangerouslySetInnerHTML** which enables to have raw HTML without escaping
  - note the word **dangerously** in its name…
- Even if you do not use it directly, it is a potential issue if you create attributes based on user inputs
- Eg: **<div {…jsonObjectComingFromUser} />**
- … as one of those fields could be **dangerouslySetInnerHTML**

Escaping of Attributes

- Issue when you have attributes that are interpreted as URLs:
  - **<a href={user_supplied} / >**
  - **<link rel="import" href={user_supplied}>**
  - **<button formaction={user_supplied}>**
- Why are URLs a potential issue?

For example, type **javascript:alert('Hi!')** in the address-bar of your browser and see what happens…

Note: you ll have to type it in, copy&paste would not work, as browsers would strip off the "javascript:" if coming from a copy&paste action…



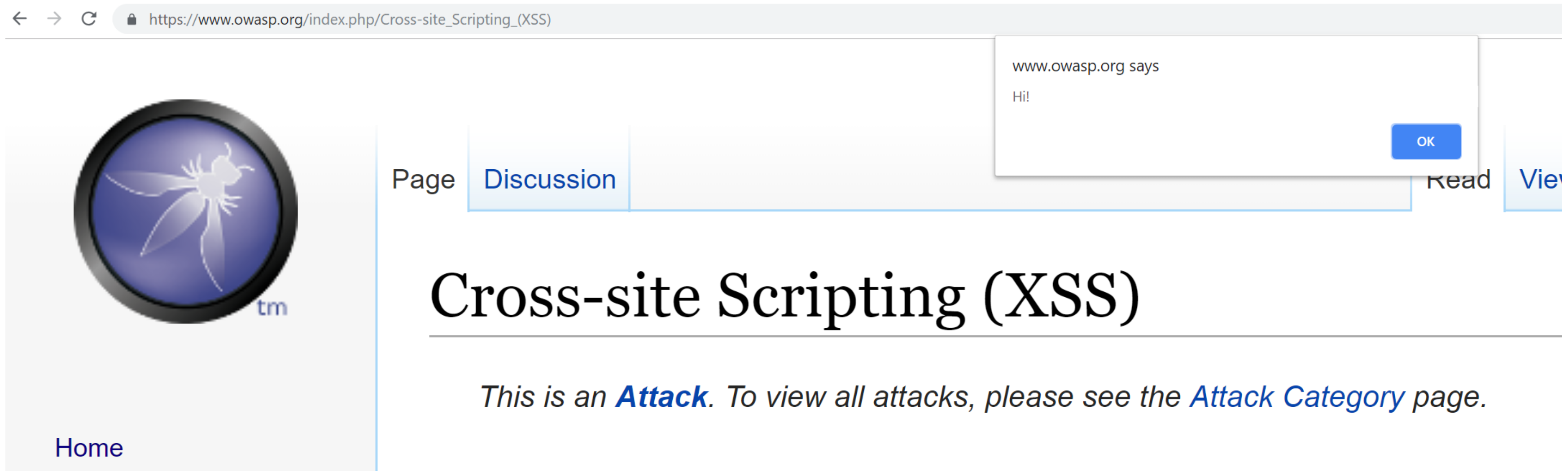
javascript:alert('Hi!')

Page Discussion Read Vie

Cross-site Scripting (XSS)

*This is an **Attack**. To view all attacks, please see the Attack Category page.*

Home



https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

www.owasp.org says

Hi!

Page Discussion Read Vie

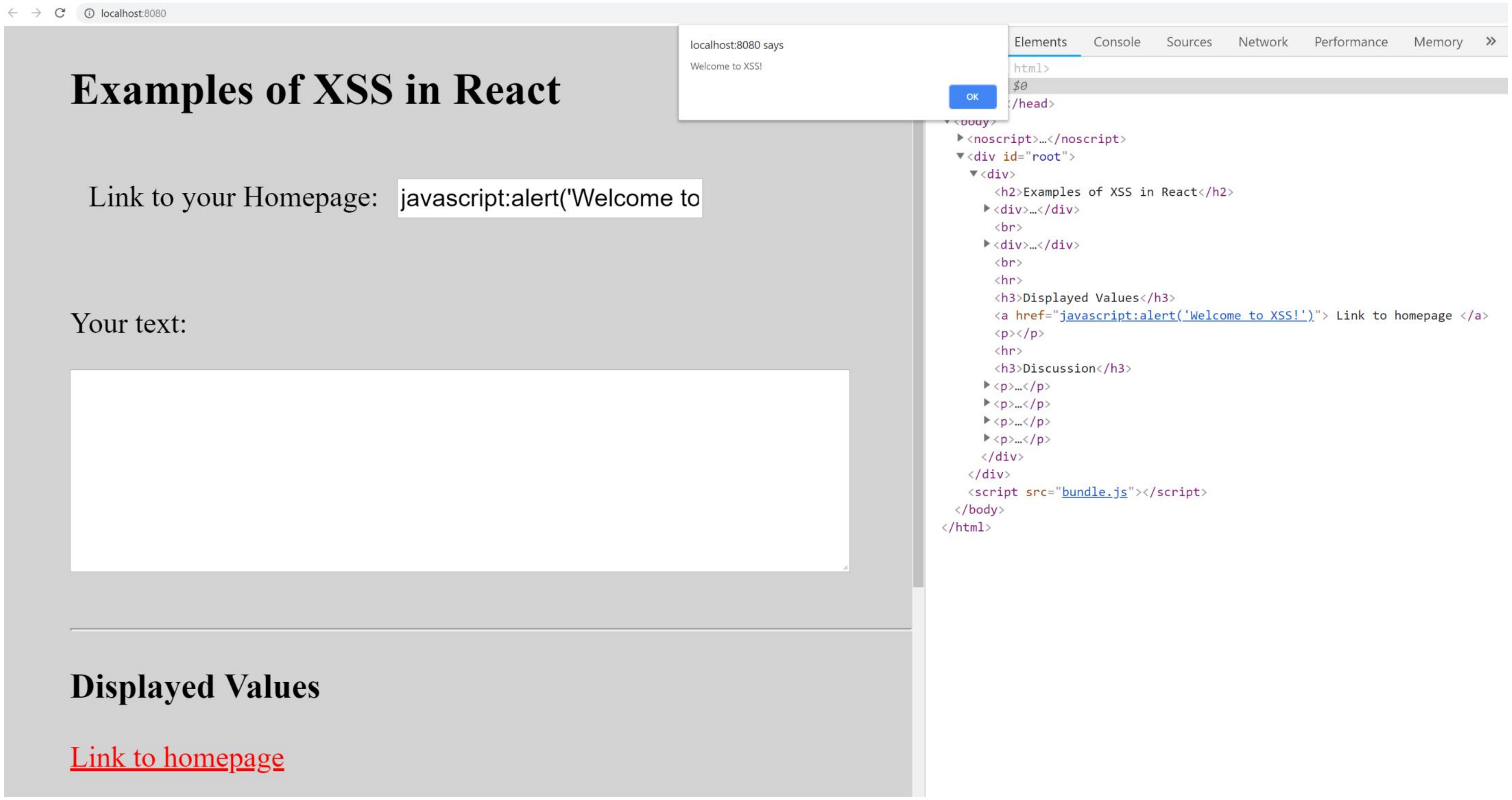
Cross-site Scripting (XSS)

*This is an **Attack**. To view all attacks, please see the Attack Category page.*

Home

<a href={this.state.homepageLink} > Link to homepage </a>

That is vulnerable to XSS when clicking the link!!!



localhost:8080 says

Welcome to XSS!

Examples of XSS in React

Link to your Homepage: javascript:alert('Welcome to

Your text:

Displayed Values

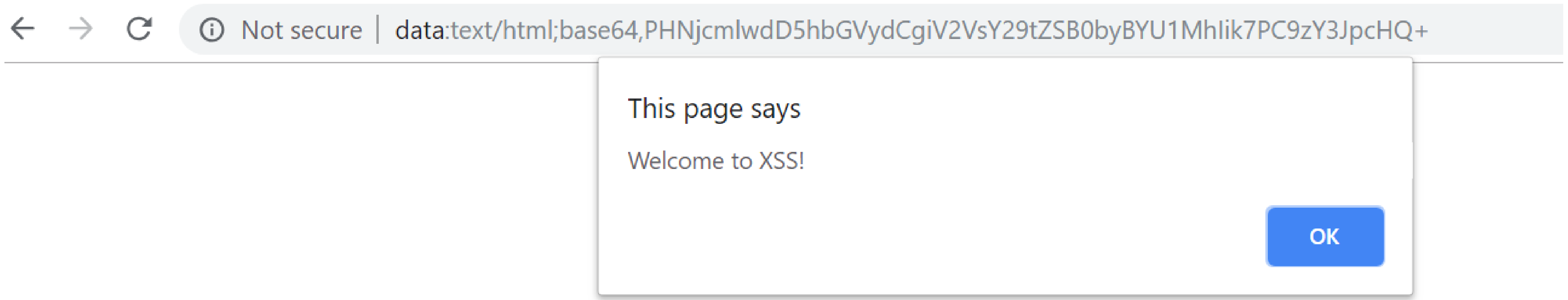
Link to homepage

```
html>
$0
/head>
<body>
  <noscript>…</noscript>
  <div id="root">
    <div>
      <h2>Examples of XSS in React</h2>
      <div>…</div>
      <br>
      <div>…</div>
      <br>
      <hr>
      <h3>Displayed Values</h3>
      <a href="javascript:alert('Welcome to XSS!')"> Link to homepage </a>
      <p></p>
      <hr>
      <h3>Discussion</h3>
      <p>…</p>
      <p>…</p>
      <p>…</p>
      <p>…</p>
    </div>
  </div>
  <script src="bundle.js"></script>
</body>
</html>
```

Sanitization

- In case of URLs, you need to manually sanitize the user inputs
  - eg, do not allow the "*javascript:*" protocol in the links
  - 2020 note: future versions of *React* will block it
- *As a rule of thumb, shouldn't write your own sanitization functions, but rather use existing libraries*
  - however, if you do, use *whitelisting*!!! Ie., allow "*http:*" and "*https:*", but block everything else… instead of *blacklisting* of just blocking "*javascript:*"
- For example, what do you think is going to happen if you use this string as URL???
  data:text/html;base64,PHNjcmlwdD5hbGVydCgiV2VsY29tZSB0by
  BYU1MhIik7PC9zY3JpcHQ+

Try it in the address-bar…



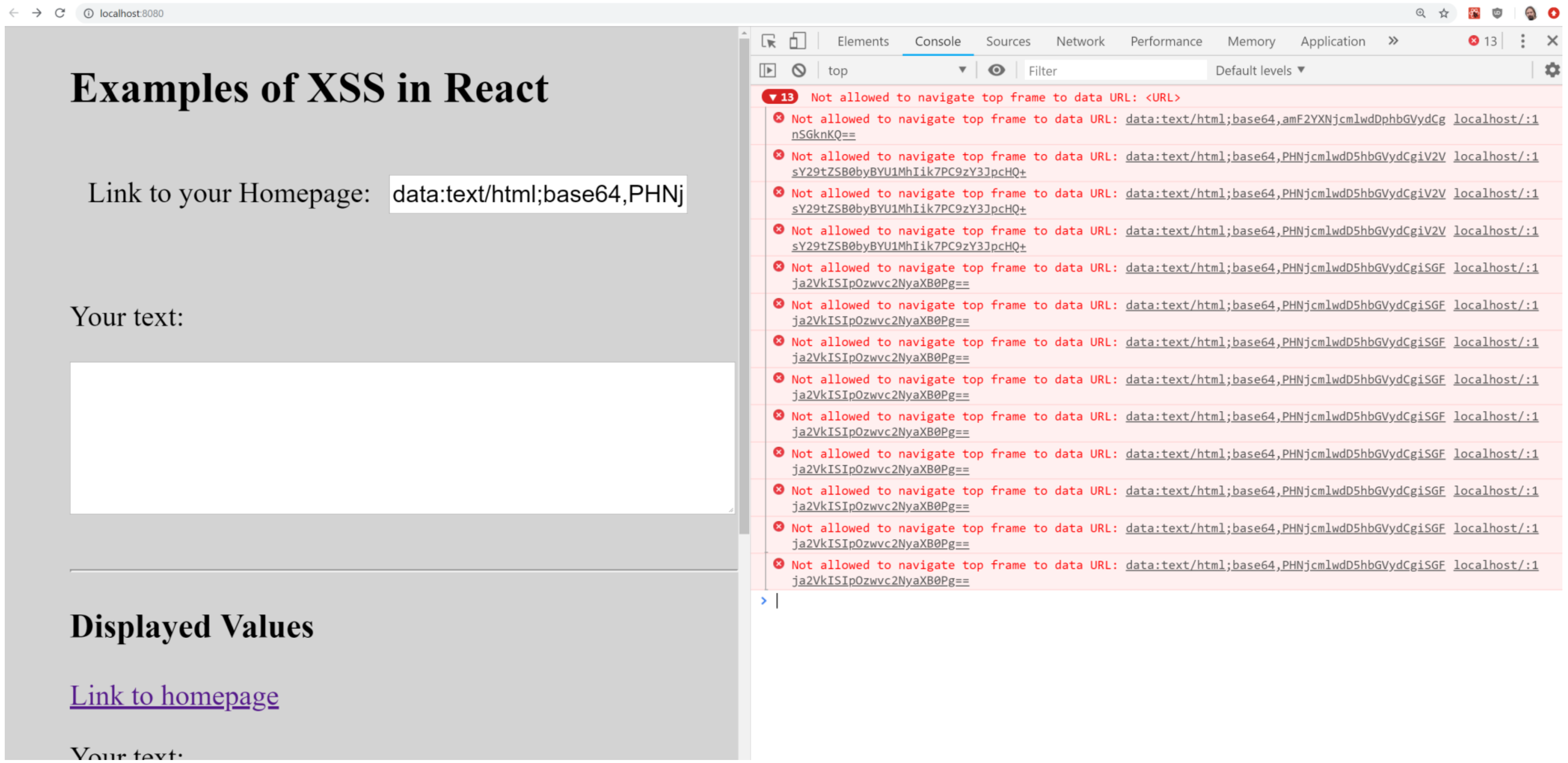
data:text/html;base64,PHNjcmlwdD5hbGVydCgiV2VsY29tZSB0byBYU1MhIik7PC9zY3JpcHQ+

This page says

Welcome to XSS!

PHNjcmlwdD5hbGVydCgiV2VsY29tZSB0byBYU1MhIik7PC9zY3JpcHQ+ is the string **<script>alert(“Welcome to XSS!");</script>** , encoded in the Base64 format

- But “feature” removed from HTML links in browsers in 2017 in the “*top frame*”, due to security concerns…
- still… good example to see why you should not write your own sanitization functions… so many weird edge cases exist!!!
  - eg, have fun looking at https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet



Examples of XSS in React

Link to your Homepage: data:text/html;base64,PHNj

Your text:

Displayed Values

Link to homepage

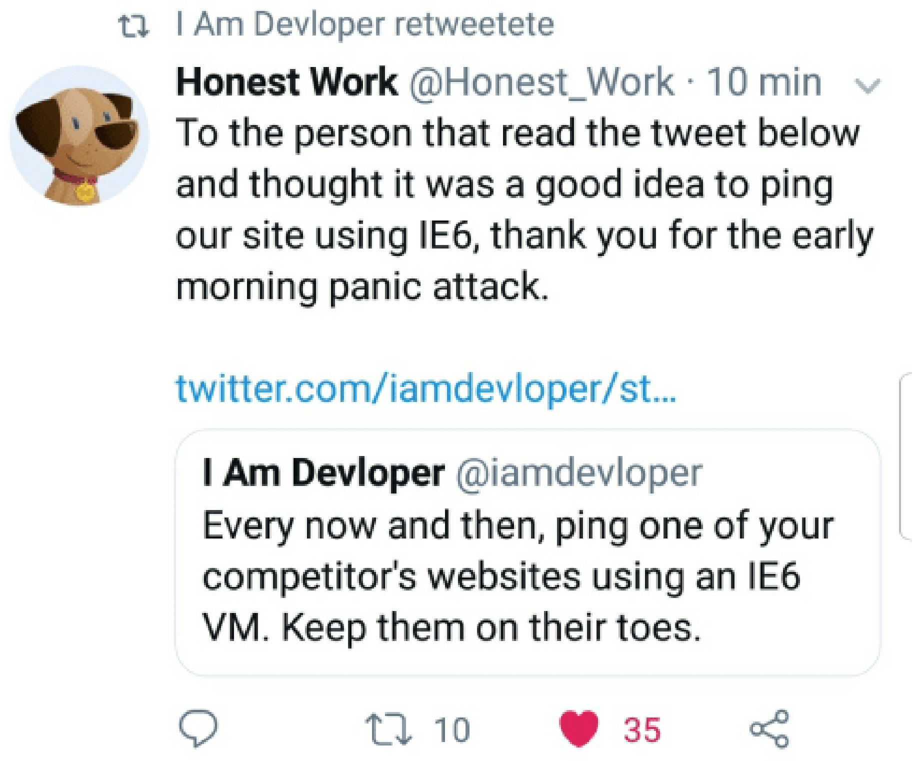
Your text:

```
▼13 Not allowed to navigate top frame to data URL: <URL>
Not allowed to navigate top frame to data URL: data:text/html;base64,amF2YXNjcmlwdDphbGVydCg localhost/:1
nSGknKQ==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiV2V localhost/:1
sY29tZSB0byBYU1MhIik7PC9zY3JpcHQ+
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiV2V localhost/:1
sY29tZSB0byBYU1MhIik7PC9zY3JpcHQ+
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiV2V localhost/:1
sY29tZSB0byBYU1MhIik7PC9zY3JpcHQ+
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
Not allowed to navigate top frame to data URL: data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGF localhost/:1
ja2VkISIpOzwvc2NyaXB0Pg==
>
```

User vs Developer

- *As a user*: **ALWAYS UPDATE TO LATEST BROWSER VERSION**
  - it will protect you from many known attacks
- *As a developer*: many of your clients will still use old browsers…
  - so you might still need to add extra layers of protection in your applications, even for attacks that would not be possible on recent browsers

- 2020: **Internet Explorer** still has a **1.7%** market share
  - 2.1% in Norway
  - In “theory” replaced by **Edge** in 2015…
- 2019: Edge was rebuilt in Chromium
- Legacy Edge in 2020
  - Global: 2.2%
  - Norway: 3.7%
- See https://gs.statcounter.com/



I Am Devloper retweetete

Honest Work @Honest_Work · 10 min

To the person that read the tweet below and thought it was a good idea to ping our site using IE6, thank you for the early morning panic attack.

twitter.com/iamdevloper/st…

I Am Devloper @iamdevloper

Every now and then, ping one of your competitor's websites using an IE6 VM. Keep them on their toes.

10 35