

Web App Security

Part 1 - Basics

arnaud.nauwynck@gmail.com

Practical Study :
Using No Code / Low Code

Java SpringBoot Security

StringInitializr [1/4]

Setup project: <https://start.spring.io/>

The screenshot shows the Spring Initializr web application at <https://start.spring.io/>. The interface is clean and modern, featuring a header with navigation icons and a search bar. The main content area is titled "spring initializr".

Project settings:

- Gradle - Groovy
- Gradle - Kotlin
- Maven

Language settings:

- Java
- Kotlin
- Groovy

Dependencies section:

No dependency selected

ADD DEPENDENCIES... CTRL + B

Spring Boot settings:

- 3.3.0 (SNAPSHOT)
- 3.3.0 (M1)
- 3.2.4 (SNAPSHOT)
- 3.2.3
- 3.1.10 (SNAPSHOT)
- 3.1.9

Project Metadata fields:

Group	com.example
Artifact	demo
Name	demo
Description	Demo project for Spring Boot
Package name	com.example.demo

Buttons at the bottom:

- GENERATE** CTRL + ↵
- EXPLORE** CTRL + SPACE
- SHARE...**

SpringInitializr [2/4]

Change to Maven

Enter Dependencies: Web, Security

Click "Generate" (Download zip)

Dependencies

[ADD DEPENDENCIES... CTRL + B](#)

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Security SECURITY

Highly customizable authentication and access-control framework for Spring applications.

SpringInitializr [3/4]

Compile & Launch Web Server

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** The project is named "demo-springboot-security". The "src" directory contains "main" and "test" packages. "main" has a "java" folder containing "com.example.demo" which holds the "DemoApplication.java" file.
- Code Editor:** The "DemoApplication.java" file is open, showing the main method annotated with @SpringBootApplication.
- Java Output Console:** The bottom pane displays the application logs:

```
2024-02-25T12:58:26.801+01:00 INFO 13924 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2024-02-25T12:58:26.801+01:00 INFO 13924 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.19]
2024-02-25T12:58:26.847+01:00 INFO 13924 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[]       : Initializing Spring embedded WebApplicationContext
2024-02-25T12:58:26.848+01:00 INFO 13924 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 977 ms
2024-02-25T12:58:27.208+01:00 WARN 13924 --- [           main] .s.s.UserDetailsServiceAutoConfiguration :
```

Using generated security password: 29e69305-0899-4677-95fe-36fef4972403

This generated password is for development use only. Your security configuration must be updated before running your application in production.

```
2024-02-25T12:58:27.303+01:00 INFO 13924 --- [           main] o.s.s.web.DefaultSecurityFilterChain      : Will secure any request with [org.springframework.security.web.DefaultSecurityFilterChain]
2024-02-25T12:58:27.356+01:00 INFO 13924 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer   : Tomcat started on port 8080 (http) with context path ''
2024-02-25T12:58:27.370+01:00 INFO 13924 --- [           main] com.example.demo.DemoApplication          : Started DemoApplication in 1.929 seconds (process running for 1.929)
2024-02-25T12:58:48.382+01:00 INFO 13924 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[]       : Initializing Spring DispatcherServlet 'dispatcherServlet'
2024-02-25T12:58:48.382+01:00 INFO 13924 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2024-02-25T12:58:48.382+01:00 INFO 13924 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 0 ms
```

SpringInitializr [3/4]

Alternative .. Compile & Launch Web Server using maven

\$ mvn package

\$ mvn spring-boot:run

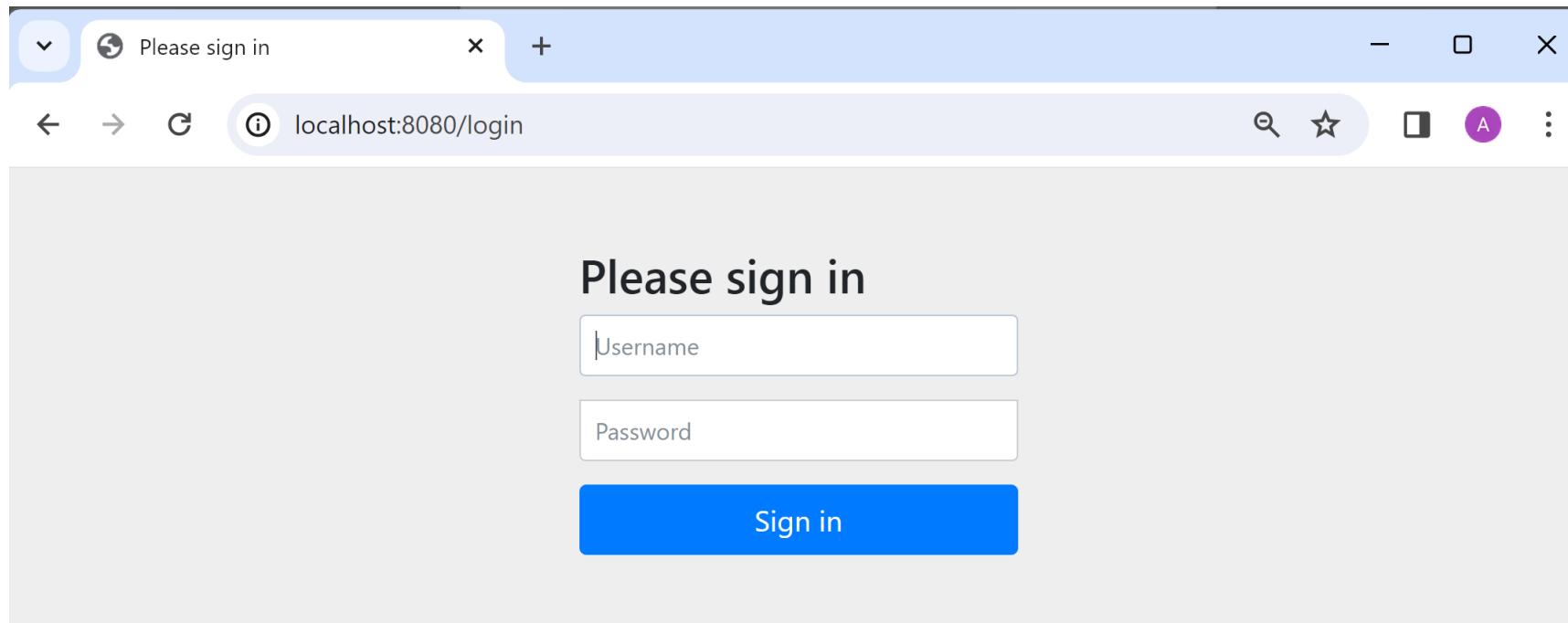


The screenshot shows a terminal window titled 'Cmder' running on a Windows system. The command line displays the following output:

```
arnaud@DesktopArnaud /cygdrive/c/arn/perso/cours/esilv/2024/security/demo-springboot-security
$ mvn spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.example:demo >-----
[INFO] Building demo 0.0.1-SNAPSHOT
[INFO]   from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] >>> spring-boot:3.2.3:run (default-cli) > test-compile @ demo >>>
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ demo ---
```

SpringInitializr [4/4]

Test: open <http://localhost:8080>



username = "user"
password = randomly generated at startup

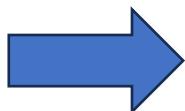
```
2024-02-25T13:35:55.221+01:00  WARN 13900 --- [           main] .s.s.UserDetailsServiceAutoConfiguration :
```

```
Using generated security password: 32b43163-eb64-428b-b616-094903ae17e6
```

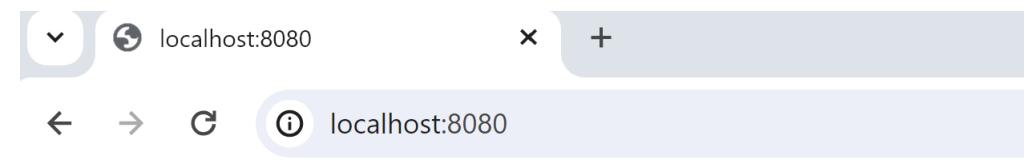
This generated password is for development use only. Your security configuration must be updated before running your application in production.

```
2024-02-25T13:35:55.543+01:00  INFO 13900 --- [           main] o.s.s.web.DefaultSecurityFilterChain      : Will secure any request with [org.springframework.security.web
```

```
2024-02-25T13:35:55.716+01:00  INFO 13900 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path ''
```



**"OK" but 404
(no more 401)**



Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

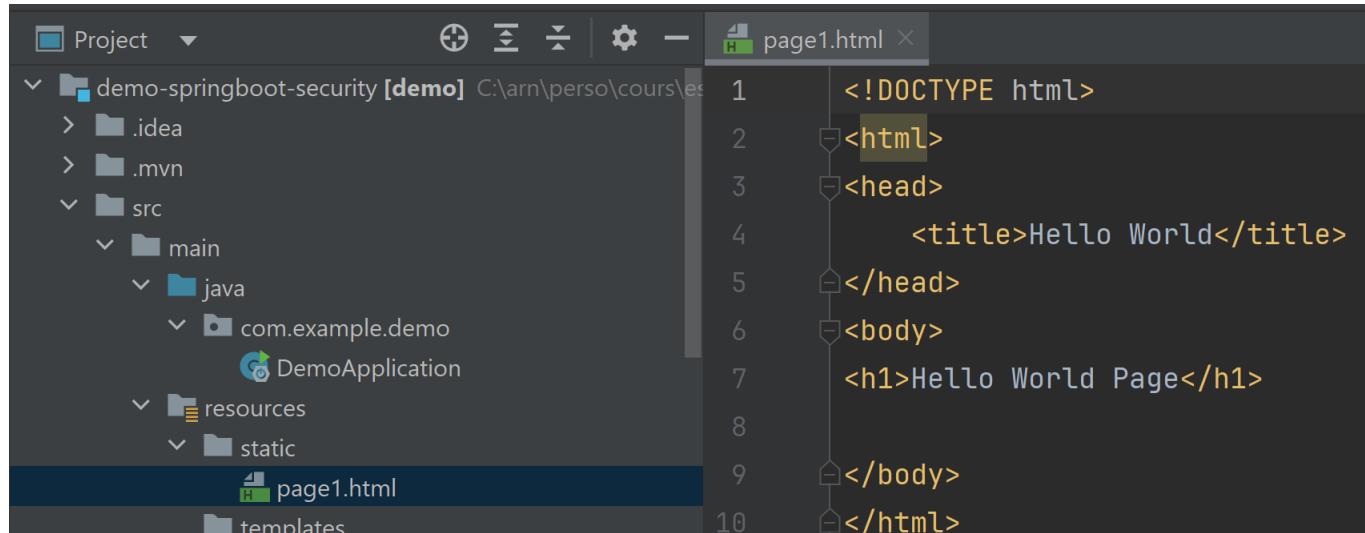
Sun Feb 25 13:56:12 CET 2024

There was an unexpected error (type=Not Found, status=404).

Add Resource static/page1.html

http GET /page1.html => Http 200

add file
**src/main/resource/
static/page1.html**



The screenshot shows a code editor with the following project structure:

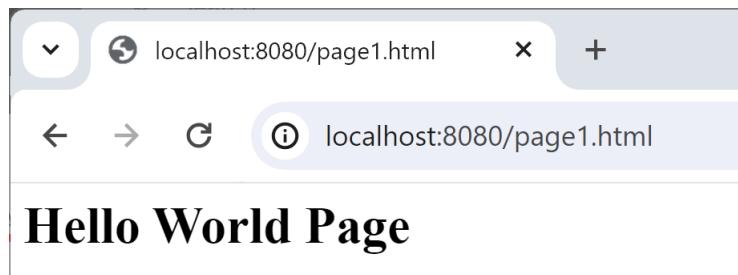
- demo-springboot-security [demo] C:\arn\perso\cours\es
- .idea
- .mvn
- src
 - main
 - java
 - com.example.demo
 - resources
 - static
 - templates

The file page1.html is selected in the resources/static folder. Its content is:

```
<!DOCTYPE html>
<html>
<head>
<title>Hello World</title>
</head>
<body>
<h1>Hello World Page</h1>
</body>
</html>
```

relaunch

http GET **/page1.html**



Install Demo & Hands-on

then pause

Detailed Analysis of 4 http scenarios

Scenario 1/4 : No authentication

Scenario 2/4 : BAD user/password

Scenario 3/4 : user/password OK => auth success, redirect, set Cookie

Scenario 4/4 : clear cookie

Detailed Analysis of 4 http scenarios



Scenario 1/4 : No authentication

Scenario 2/4 : BAD user/password

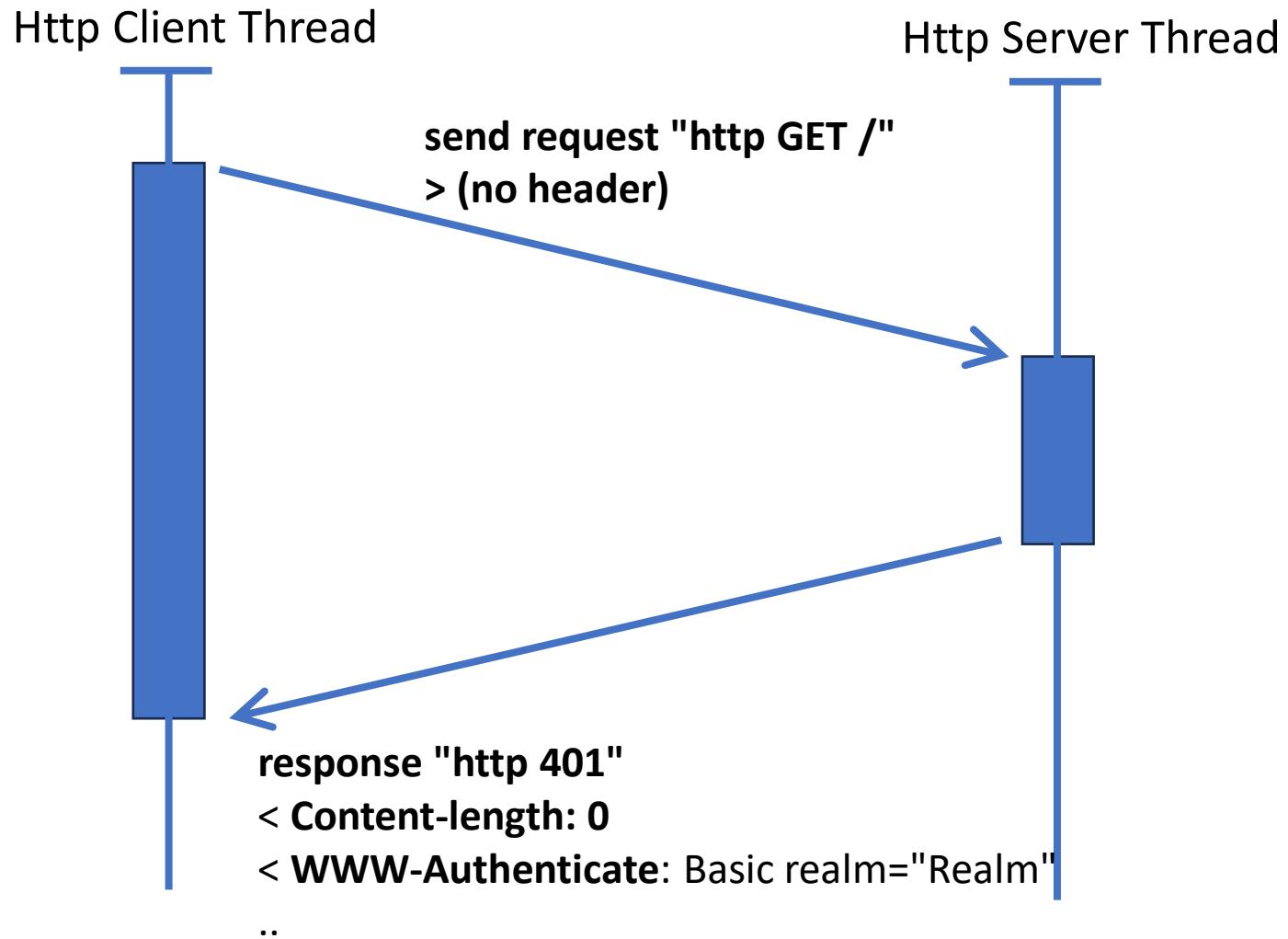
Scenario 3/4 : user/password OK => auth success, redirect, set Cookie

Scenario 4/4 : clear cookie

Scenario 1: NO authentication using curl

```
$ curl -vv http://localhost:8080
*   Trying [::1]:8080...
* Connected to localhost (::1) port 8080
> GET / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.4.0
> Accept: */*
>
< HTTP/1.1 401
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
< Set-Cookie: JSESSIONID=F4AC86090C48019BB14C4315942710AA; Path=/; HttpOnly
< WWW-Authenticate: Basic realm="Realm"
< X-Content-Type-Options: nosniff
< X-XSS-Protection: 0
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< X-Frame-Options: DENY
< Content-Length: 0
< Date: Sun, 25 Feb 2024 12:36:32 GMT
<
* Connection #0 to host localhost left intact
```

Http Request/Response Sequence



401 : Unauthorized



MDN Web Docs

<https://developer.mozilla.org> › Web › HTTP › Status ::

401 Unauthorized - HTTP - MDN Web Docs

Apr 10, 2023 — The HyperText Transfer Protocol (**HTTP**) **401** Unauthorized response status code indicates that the client request has not been completed ...

[403 Forbidden](#) · [WWW-Authenticate](#) · [402 Payment Required](#)

curl, no authentication data
=> NO result, only 401 response code

retry almost "Same" request in Web Browser
=> get different response !

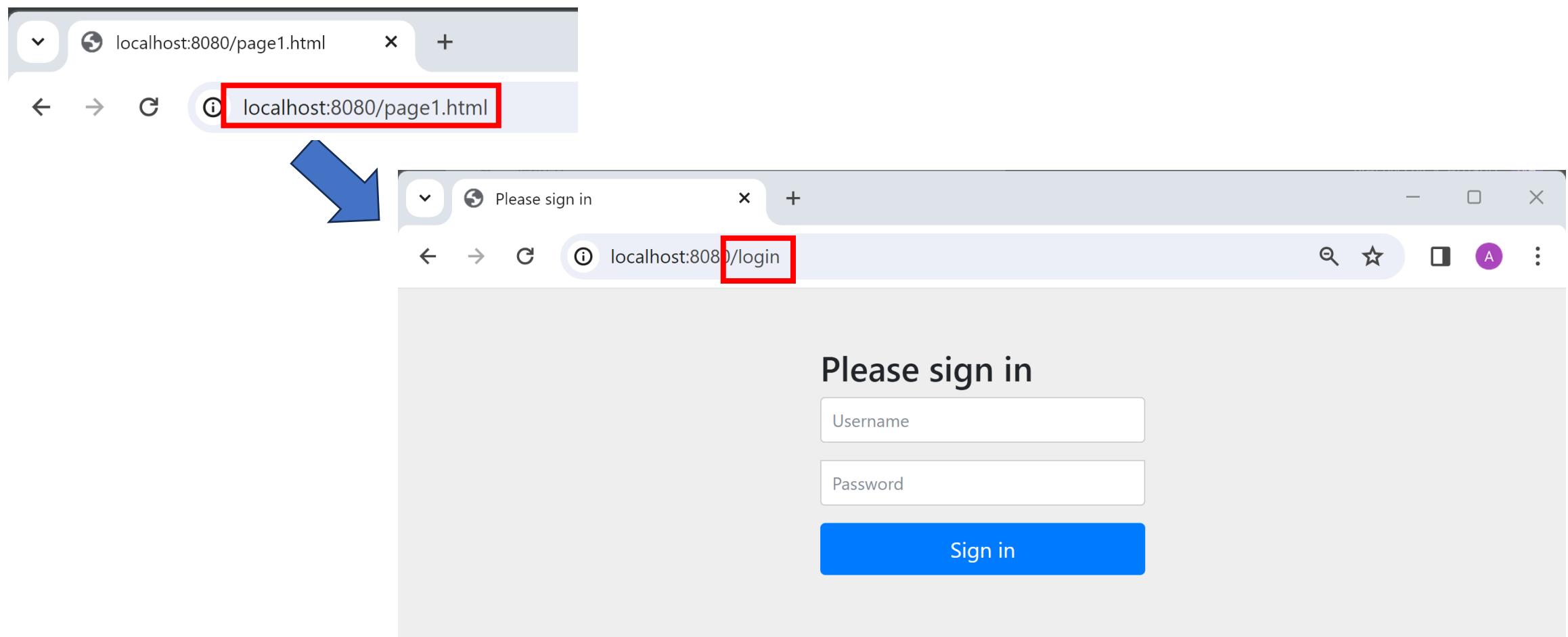
http 301 ... redirect to login html page

exactly "same" as Web Browser:
using curl to accept html content format... cf next

curl -H Accept:application/xhtml

```
$ curl -vv http://localhost:8080/page1.html -H Accept:text/html,application/xhtml
*   Trying [::1]:8080...
* Connected to localhost (::1) port 8080
> GET /page1.html HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.4.0
> Accept:text/html,application/xhtml
>
< HTTP/1.1 302
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
< Set-Cookie: JSESSIONID=3AF9670D9ACFF604C58140D4BA8159FB; Path=/; HttpOnly
< X-Content-Type-Options: nosniff
< X-XSS-Protection: 0
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< X-Frame-Options: DENY
< Location: http://localhost:8080/login
< Content-Length: 0
< Date: Sun, 25 Feb 2024 17:22:50 GMT
<
* Connection #0 to host localhost left intact
```

Scenario 1 .. redo using Web browser Chrome

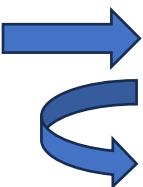


open (F12) : Developer Tools > Network "preserve log"=true

request

"/page1.html"

redirect "/login"



The screenshot shows the Chrome DevTools Network tab with the "Preserve log" checkbox selected. The timeline at the top shows a single request from 10 ms to 60 ms. Below the timeline, the requests table lists:

Name	Headers	Preview	Response	Initiator	Timing	Cookies
page1.html						
login						
bootstrap.min.css						
signin.css						

The "General" section of the right panel shows the following details for the "login" request:

- Request URL: http://localhost:8080/login
- Request Method: GET
- Status Code: 302 Found
- Remote Address: [::1]:8080
- Referrer Policy: strict-origin-when-cross-origin

The "Response Headers" section includes:

- Cache-Control: no-cache, no-store, max-age=0, must-revalidate
- Connection: keep-alive
- Content-Length: 0
- Date: Sun, 25 Feb 2024 14:28:26 GMT
- Expires: 0
- Keep-Alive: timeout=60
- Location: http://localhost:8080/login
- Pragma: no-cache
- Set-Cookie: JSESSIONID=6BA1F0AB3F8ADCAA5D78A3B8D376B904; Path=/; HttpOnly
- Vary: Origin
- Vary: Access-Control-Request-Method
- Vary: Access-Control-Request-Headers
- X-Content-Type-Options: nosniff
- X-Frame-Options: DENY

At the bottom of the Network tab, the stats are: 4 requests | 2.5 kB transferred | 127 kB resources.

Detailed Request Headers

	Headers	Preview	Response	Initiator	Timing	Cookies
X-Frame-Options:			DENY			
X-Xss-Protection:			0			
▼ Request Headers		<input type="checkbox"/> Raw				
Accept:	text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7					
Accept-Encoding:	gzip, deflate, br, zstd					
Accept-Language:	en,fr-FR;q=0.9,fr;q=0.8,en-FR;q=0.7,en-US;q=0.6					
Cache-Control:	max-age=0					
Connection:	keep-alive					
Cookie:	metabase.DEVICE=7856a6fd-dd9c-495d-880e-0b2ed106aab5; _ga=GA1.1.576560884.1694723966; JSESSIONID=41795E3A8958BD0E3F603879725681A0					
Host:	localhost:8080					
Sec-Ch-Ua:	"Not A(Brand";v="99", "Google Chrome";v="121", "Chromium";v="121"					
Sec-Ch-Ua-Mobile:	?0					
Sec-Ch-Ua-Platform:	"Windows"					
Sec-Fetch-Dest:	document					
Sec-Fetch-Mode:	navigate					
Sec-Fetch-Site:	none					
Sec-Fetch-User:	?1					
Upgrade-Insecure-Requests:	1					
User-Agent:	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.0.0 Safari/537.36					

Detailed Response Headers

Headers		Preview	Response	Initiator	Timing	Cookies
Referrer-Policy					strict-origin-when-cross-origin	
Response Headers						<input type="checkbox"/> Raw
Cache-Control:	no-cache, no-store, max-age=0, must-revalidate					
Connection:	keep-alive					
Content-Length:	0					
Date:	Sun, 25 Feb 2024 14:28:26 GMT					
Expires:	0					
Keep-Alive:	timeout=60					
Location:	http://localhost:8080/login					
Pragma:	no-cache					
Set-Cookie:	JSESSIONID=6BA1F0AB3F8ADCAA5D78A3B8D376B904; Path=/; HttpOnly					
Vary:	Origin					
Vary:	Access-Control-Request-Method					
Vary:	Access-Control-Request-Headers					
X-Content-Type-Options:	nosniff					
X-Frame-Options:	DENY					
X-Xss-Protection:	0					

Http response code 302 + Http Response Header "location:" => redirect

 Filter

In this article

Status

Specifications

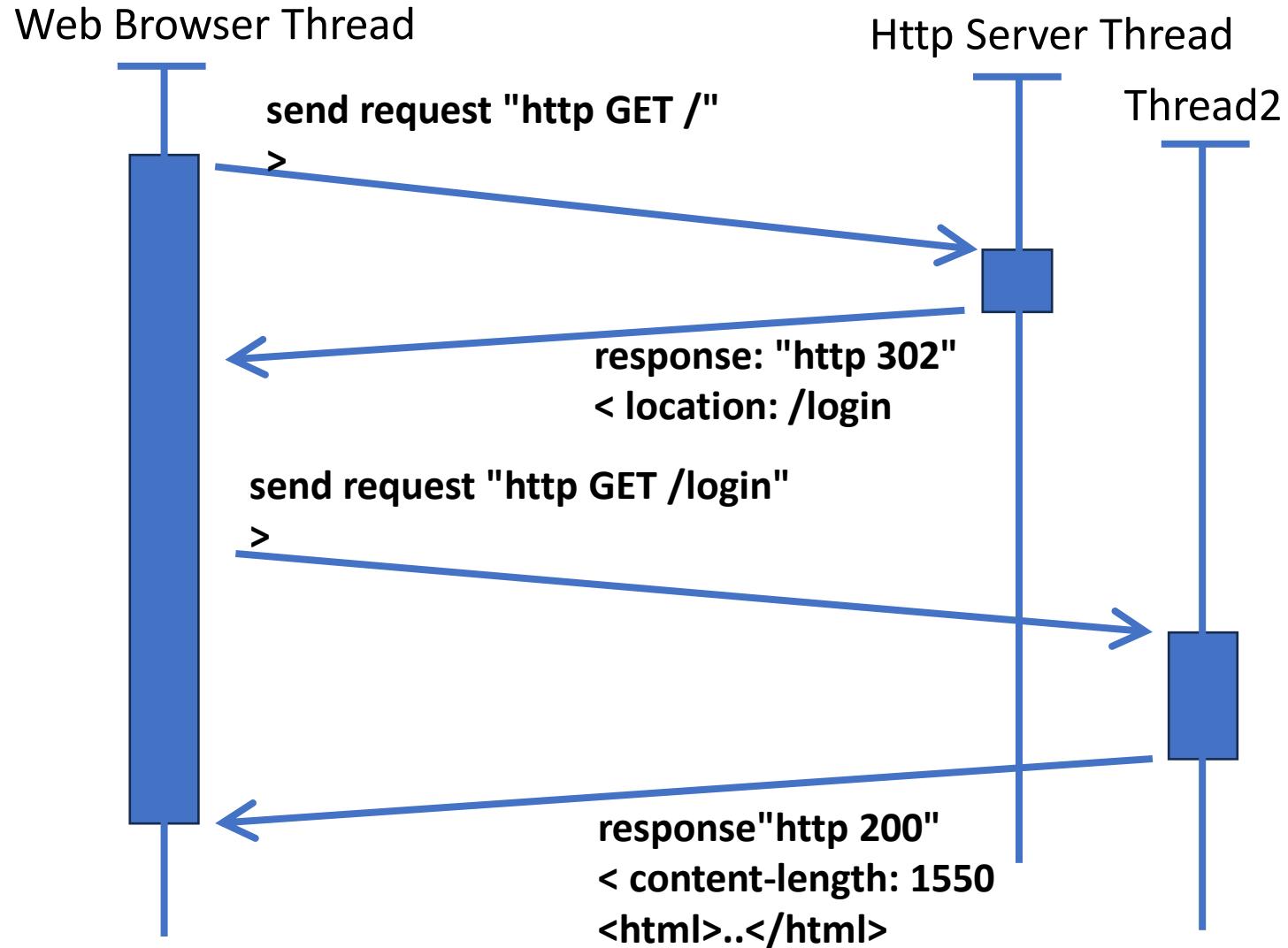
Browser compatibility

See also

302 Found

The HyperText Transfer Protocol (HTTP) `302 Found` redirect status response code indicates that the resource requested has been temporarily moved to the URL given by the `Location` header. A browser redirects to this page but search engines don't update their links to the resource (in 'SEO-speak', it is said that the 'link-juice' is not sent to the new URL).

Http Redirect Sequence



Next request (after redirected)

GET /login => http 200 <html>..</html>

Name	X	Headers	Preview	Response	Initiator	Timing	Cookies
page1.html							
login		▼ General					
bootstrap.min.css							
signin.css							
		Request URL:		http://localhost:8080/login			
		Request Method:		GET			
		Status Code:		● 200 OK			
		Remote Address:		[:1]:8080			
		Referrer Policy:		strict-origin-when-cross-origin			
		▼ Response Headers	<input type="checkbox"/> Raw				
		Cache-Control:		no-cache, no-store, max-age=0, must-revalidate			
		Connection:		keep-alive			
		Content-Length:		1550			
		Content-Type:		text/html;charset=UTF-8			
		Date:		Sun, 25 Feb 2024 14:28:26 GMT			
		Expires:		0			
		Keep-Alive:		timeout=60			
		Pragma:		no-cache			
		Vary:		Origin			
		Vary:		Access-Control-Request-Method			
		Vary:		Access-Control-Request-Headers			
		X-Content-Type-Options:		nosniff			
		X-Frame-Options:		DENY			
		X-Xss-Protection:		0			
4 requests		2.5 kB transferred		127 kB resources			

Detailed Analysis of 4 http scenarios

Scenario 1/4 : No authentication



Scenario 2/4 : BAD user/password

Scenario 3/4 : user/password OK => auth success, redirect, set Cookie

Scenario 4/4 : clear cookie

Wrong Password

401 : Unauthorized

```
$ curl -vv http://localhost:8080/page1.html -u user:WRONGPASS
*   Trying [::1]:8080...
*   Connected to localhost (::1) port 8080
*   Server auth using Basic with user 'user'
> GET /page1.html HTTP/1.1
> Host: localhost:8080
> Authorization: Basic dXNlcjpXUk9OR1BBU1M=
> User-Agent: curl/8.4.0
> Accept: */*
>
< HTTP/1.1 401
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
* Authentication problem. Ignoring this.
< WWW-Authenticate: Basic realm="Realm"
< X-Content-Type-Options: nosniff
```

Wrong User

401 : Unauthorized

```
Cmder
$ curl -vv http://localhost:8080/page1.html -u WRONGUSER:${PASSWORD}
* Trying [::1]:8080...
* Connected to localhost (::1) port 8080
* Server auth using Basic with user 'WRONGUSER'
> GET /page1.html HTTP/1.1
> Host: localhost:8080
> Authorization: Basic V1JPTkdVU0VSOjMyYjQzMjYzLWViNjQtNDI4Yi1iNjE2LTA5
> User-Agent: curl/8.4.0
> Accept: */*
>
< HTTP/1.1 401
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
* Authentication problem. Ignoring this.
< WWW-Authenticate: Basic realm="Realm"
< Content-Type: Options; nosniff
```

All 401 !
NO Difference between

No Auth / Wrong User / Wrong Password

Rule of Thumb : never give any information (to attackers)

Detailed Analysis of 4 http scenarios

Scenario 1/4 : No authentication

Scenario 2/4 : BAD user/password



Scenario 3/4 : user/password OK => auth success, set Cookie

Scenario 4/4 : clear cookie

http GET using login/password
=> OK + get body

curl -vv http://localhost:8080/page1.html -u user:\${PASSWORD}

```
Cmder
arnaud@DesktopArnaud /cygdrive/c/arn/devPerso
$ curl -vv http://localhost:8080/page1.html -u user:${PASSWORD}
*   Trying [::1]:8080...
*   Connected to localhost (::1) port 8080
* Server auth using Basic with user 'user'
> GET /page1.html HTTP/1.1
> Host: localhost:8080
> Authorization: Basic dXNlcjozMmI0MzE2My1lYjY0LTQyOGItYjYxNi0wOTQ5MDNhZTE3ZTY=
> User-Agent: curl/8.4.0
> Accept: */*
>
< HTTP/1.1 200
< Vary: Origin
```

empty line
(end of http headers)

61 bytes of data
then close TCP socket

```
< HTTP/1.1 200
< Vary: Origin
< Vary: Access-Control-Request-Method
< Vary: Access-Control-Request-Headers
< Last-Modified: Sun, 25 Feb 2024 11:57:53 GMT
< Accept-Ranges: bytes
< X-Content-Type-Options: nosniff
< X-XSS-Protection: 0
< Cache-Control: no-cache, no-store, max-age=0, must-revalidate
< Pragma: no-cache
< Expires: 0
< X-Frame-Options: DENY
< Content-Type: text/html
< Content-Length: 61
< Date: Sun, 25 Feb 2024 14:19:45 GMT
<
<html>
<body>
<H1>Hello World Page</H1>
</body>
</html>* Connection #0 to host localhost left intact
```

```
Notice: curl -u user:${password}
> Authorization: Basic dXNlcjpwYXNzd29yZA==
```

user:password is sent as Http Header "Authorization: Basic <<base64>>"
in **CLEAR TEXT** (encoded in base 64)

to decode, use "base64 -d"

```
Cmder
$ echo dXNlcjpwYXNzd29yZA== | base64 -d
user:password
```

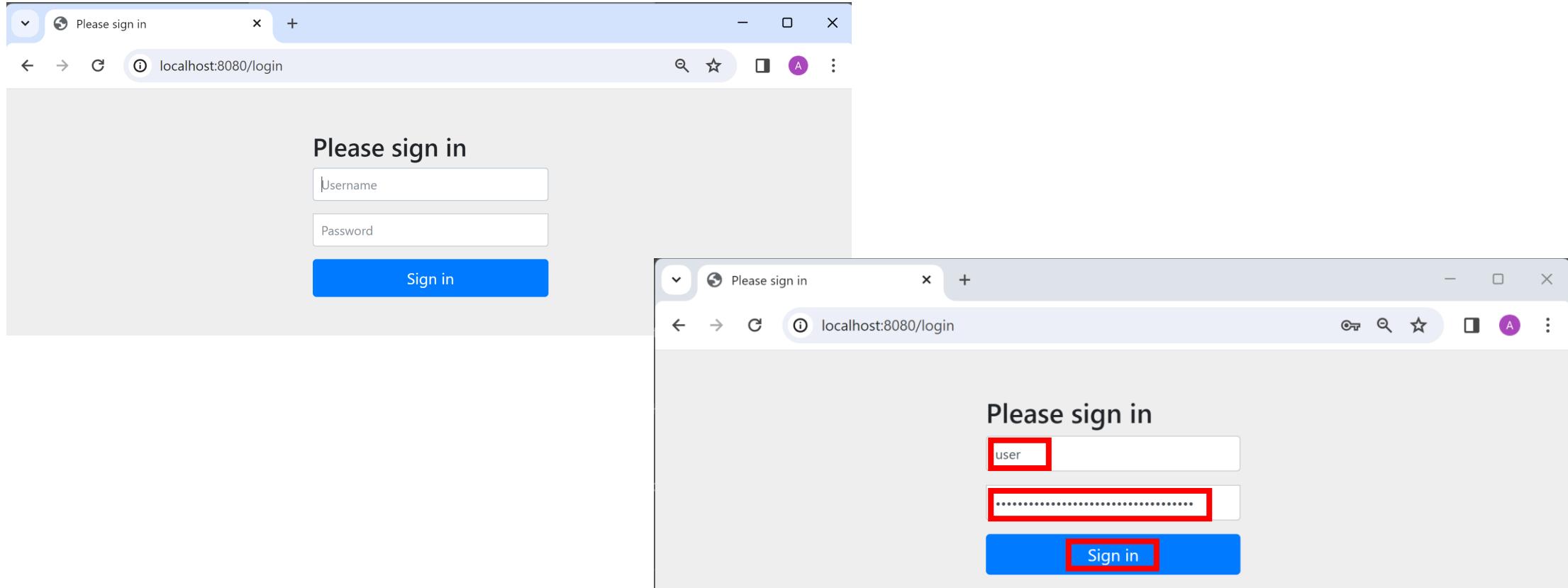
Remark 2 : when using explicit user:password
=> NO set-cookie

By default, Web Browsers do not send "user:password"

Server detect non interactive (batch) usage / Rest Api call

NO need to create interactive "session", so no set-cookie needed

user/password Using applicative Login Html page + Form + Button



Please sign in

Username

Password

Sign in

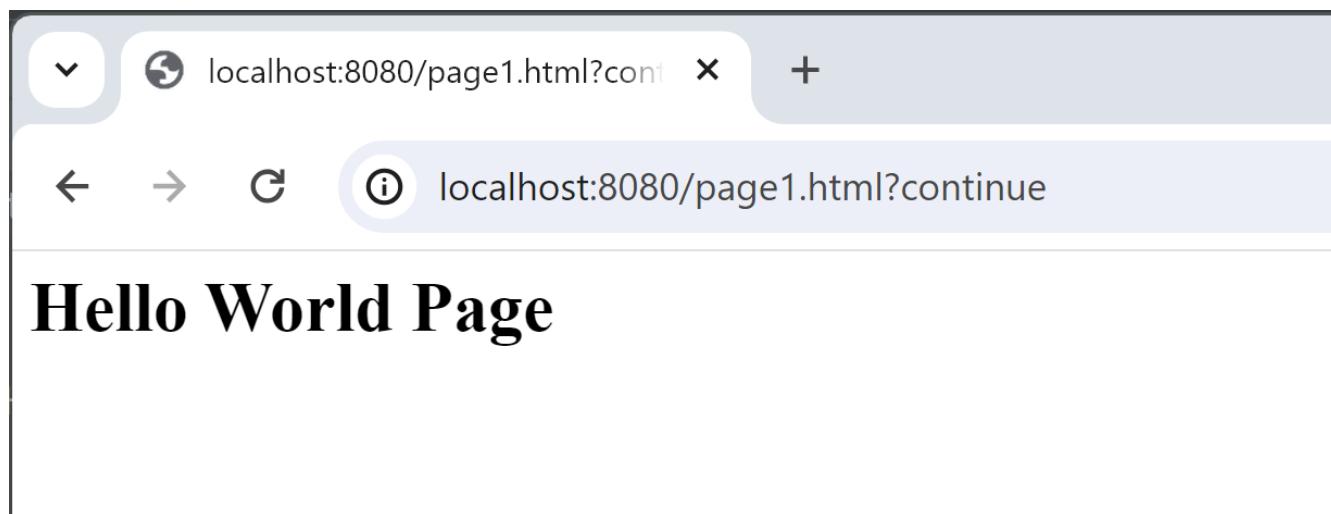
Please sign in

user

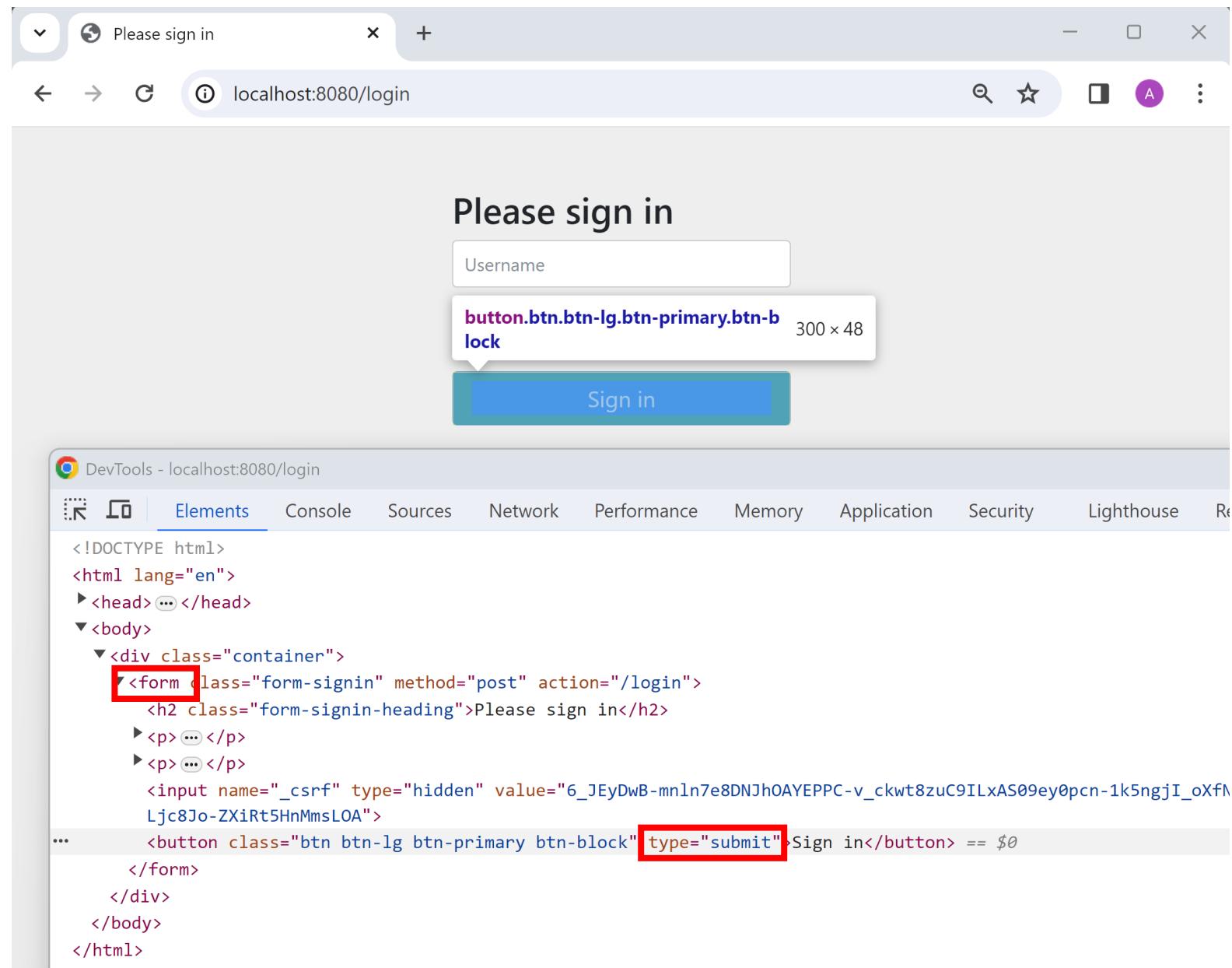
.....

Sign in

Authenticated OK => http 200
/page1.html?continue



F12 - Elements : Button Html Source



The screenshot shows a browser window with the title "Please sign in" and the URL "localhost:8080/login". Below the browser is the Google Chrome DevTools interface, specifically the "Elements" tab. The "Elements" tab is active, displaying the HTML structure of the page. A red box highlights the "button" element in the DevTools tree, which corresponds to the "Sign in" button on the page. Another red box highlights the "type='submit'" attribute in the DevTools code editor, also pointing to the same button. The browser window shows a "Username" input field above the button.

```
<!DOCTYPE html>
<html lang="en">
  <head> ... </head>
  <body>
    <div class="container">
      <form class="form-signin" method="post" action="/login">
        <h2 class="form-signin-heading">Please sign in</h2>
        <p>...</p>
        <p>...</p>
        <input name="_csrf" type="hidden" value="6_JEyDwB-mnln7e8DNJhOAYEPPC-v_ckwt8zuC9ILxAS09ey0pcn-1k5ngjI_oXfLJjc8Jo-ZXiRt5HnMmsLOA">
      ...
      <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>
    </form>
  </div>
</body>
</html>
```

Details on <form>

```
▼ <form class="form-signin" method="post" action="/login"> == $0
  <h2 class="form-signin-heading">Please sign in</h2>
  ▼ <p>
    <label for="username" class="sr-only">Username</label>
    <input type="text" id="username" name="username" class="form-control" placeholder="Username" required autofocus>
  </p>
  ▼ <p>
    <label for="password" class="sr-only">Password</label>
    <input type="password" id="password" name="password" class="form-control" placeholder="Password" required>
  </p>
  <input name="_csrf" type="hidden" value="6_JEyDwB-mnln7e8DNJhOAYEPPC-v_ckwt8zuC9ILxAS09ey0pcn-1k5ngjI_oXfNf9VCmc8E
-ZXiRt5HnMmsLOA">
  <button class="btn btn-lg btn-primary btn-block" type="submit">Sign in</button>
</form>
```

<form> ... behavior explanation

on click button "Signin"

=> submit Form

=> call http POST "/login"

with payload = FormData { username, password, _csrf }

Notice: _csrf ?

C.S.R.F. stands for "Cross Site Request Forgery"

For preventing "http://a-malicious-site/click-me-page"
to automatically call your http POST "/login" on your site
and filling sensitive data from your browser

cf Appendix

Http POST /login

X	Headers	Payload	Preview	Response	Initiator	Timing	Cookies
▼ General							
Request URL: http://localhost:8080/login							
Request Method:	POST						
Status Code:	302 Found						
Remote Address:	[:1]:8080						
Referrer Policy:	strict-origin-when-cross-origin						
▼ Response Headers							
<input type="checkbox"/> Raw							
Cache-Control:	no-cache, no-store, max-age=0, must-revalidate						
Connection:	keep-alive						
Content-Length:	0						
Date:	Sun, 25 Feb 2024 15:00:43 GMT						
Expires:	0						
Keep-Alive:	timeout=60						
Location:	http://localhost:8080/page1.html?continue						
Pragma:	no-cache						
Set-Cookie:	JSESSIONID=E4F23994BBF20E40D20C63CC76C98EA5; Path=/; HttpOnly						
Vary:	Origin						
Vary:	Access-Control-Request-Method						
Vary:	Access-Control-Request-Headers						
X-Content-Type-Options:	nosniff						
X-Frame-Options:	DENY						

POST ... payload user,password in CLEAR text FormData

The screenshot shows the Network tab of a browser developer tools interface. The 'Payload' tab is selected. Below it, under 'Form Data', there are three entries: 'username: user', 'password: 32b43163-eb64-428b-b616-094903ae17e6', and '_csrf: tJ1Tc6yZKHN83PhLjfaz3dwEhXeYz9hEjP-ve7vxdsT4ZYCM0fliQc6hHBdR5M177NuH6u1gqE77_7ppvM6YTd7GRqHIBLHp'. The 'username:' and 'password:' fields are highlighted with red boxes.

▼Form Data	view source view URL-encoded
username:	user
password:	32b43163-eb64-428b-b616-094903ae17e6
_csrf:	tJ1Tc6yZKHN83PhLjfaz3dwEhXeYz9hEjP-ve7vxdsT4ZYCM0fliQc6hHBdR5M177NuH6u1gqE77_7ppvM6YTd7GRqHIBLHp

POST ... payload view parsed or view source

X Headers **Payload** Preview Response Initiator Timing Cookies

▼Form Data [view source](#) [view URL-encoded](#)

username: user

password: 32b43163-eb64-428b-b616-094903ae17e6

_csrf: tJ1Tc6yZKHN83PhLjfaz3dwEhXeYz9hEjP-ve7vxdsT4ZYCM0fliQc6hHBdR5M177NuH6u1gqE77_7ppvM6YTd7GRqHIBLHp

X Headers **Payload** Preview Response Initiator Timing Cookies

▼Form Data [view parsed](#)

username= user & **password=** 32b43163-eb64-428b-b616-094903ae17e6 & **_csrf=** tJ1Tc6yZKHN83PhLjfaz3dwEhXeYz9hEjP-ve7' c6hHBdR5M177NuH6u1gqE77_7ppvM6YTd7GRqHIBLHp

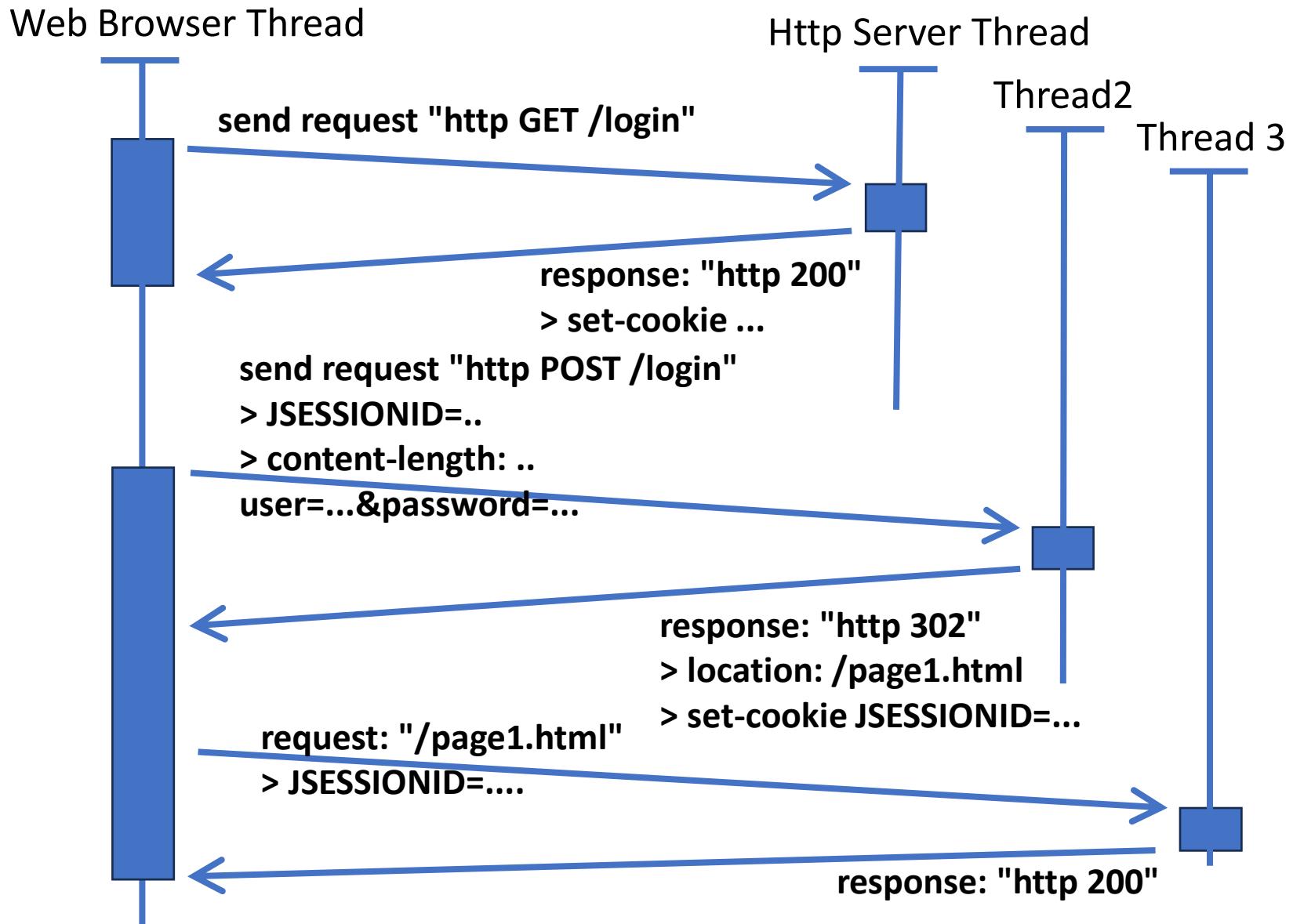
curl equivalent to http POST /login

```
Cmder
$ curl -vv 'http://localhost:8080/login'    -H 'Origin: http://localhost:8080'    -H 'Referer: ht
tp://localhost:8080/login' --data-raw 'username=user&password=32b43163-eb64-428b-b616-094903ae
17e6&_csrf=I3iPb2oPGITQyNKh6jnEro3WDnMTYE_eNInLiCRDyjoj34kQRq7Vgw6erb9q7CWiRTwn-nvI0p1VizzDLmv
6Rd0-A1G7rEk' -H 'Content-Type: application/x-www-form-urlencoded' -H 'Cookie: metabase.DEVICE=
7856a6fd-dd9c-495d-880e-0b2ed106aab5; _ga=GA1.1.576560884.1694723966; JSESSIONID=7A481C4907DBC1
DA5EBA75E4A53F5C40'
*   Trying [::1]:8080...
* Connected to localhost (::1) port 8080
> POST /login HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.4.0
> Accept: /*
> Origin: http://localhost:8080
> Referer: http://localhost:8080/login
> Content-Type: application/x-www-form-urlencoded
> Cookie: metabase.DEVICE=7856a6fd-dd9c-495d-880e-0b2ed106aab5; _ga=GA1.1.576560884.1694723966;
JSESSIONID=7A481C4907DBC1DA5EBA75E4A53F5C40
> Content-Length: 162
>
< HTTP/1.1 302
< Vary: Origin
< Vary: Access-Control-Request-Method
```

replaying EXACT SAME curl command twice => http 401

```
$ curl -vv 'http://localhost:8080/login' -H 'Origin: http://localhost:8080' -H 'Referer: http://localhost:8080/login' --data-raw 'username=user&password=32b43163-eb64-428b-b616-094903ae17e6&_csrf=I3iPb2oPGITQyNKh6jnEro3WDnMTYE_eNInLiCRDyjoj34kQRRq7Vgw6erb9q7CWiRTwn-nvI0p1VizzDLmv6Rd0-A1G7rEk' -H 'Content-Type: application/x-www-form-urlencoded' -H 'Cookie: metabase.DEVICE=7856a6fd-dd9c-495d-880e-0b2ed106aab5; _ga=GA1.1.576560884.1694723966; JSESSIONID=7A481C4907DBC1DA5EBA75E4A53F5C40'
* Trying [::1]:8080...
* Connected to localhost (::1) port 8080
> POST /login HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/8.4.0
> Accept: */*
> Origin: http://localhost:8080
> Referer: http://localhost:8080/login
> Content-Type: application/x-www-form-urlencoded
> Cookie: metabase.DEVICE=7856a6fd-dd9c-495d-880e-0b2ed106aab5; _ga=GA1.1.576560884.1694723966; JSESSIONID=7A481C4907DBC1DA5EBA75E4A53F5C40
> Content-Length: 162
>
< HTTP/1.1 401
< Vary: Origin
< Vary: Access-Control-Request-Method
```

Http POST Sequence



How did server remembered to redirect back from "/login" to "/page1.html" ?

Here, server is Statefull ...

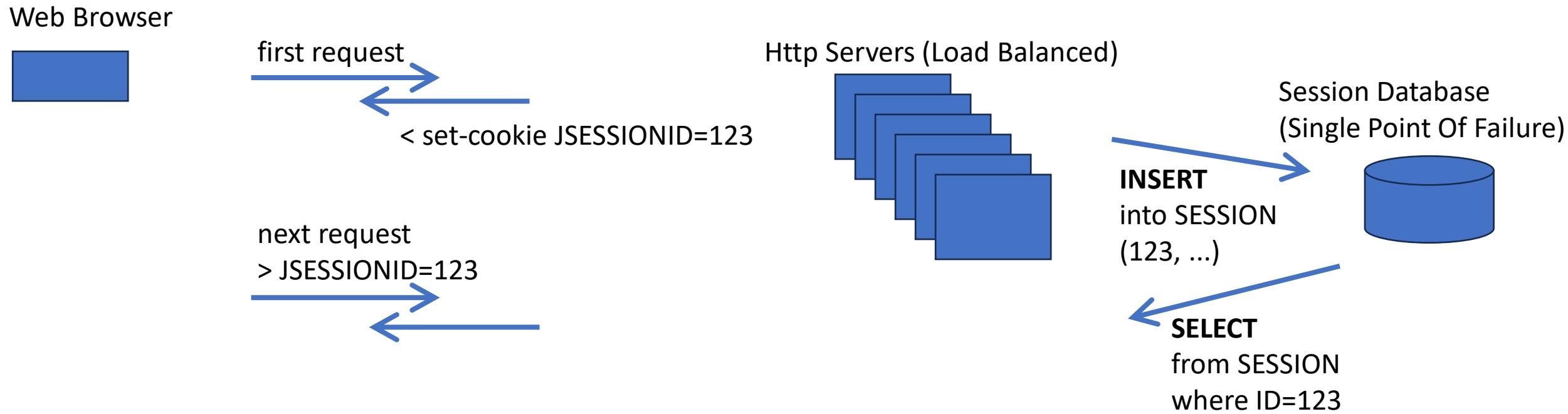
JSESSIONID are returned as "set-cookie" header from server to client

then passed back as header from client to server

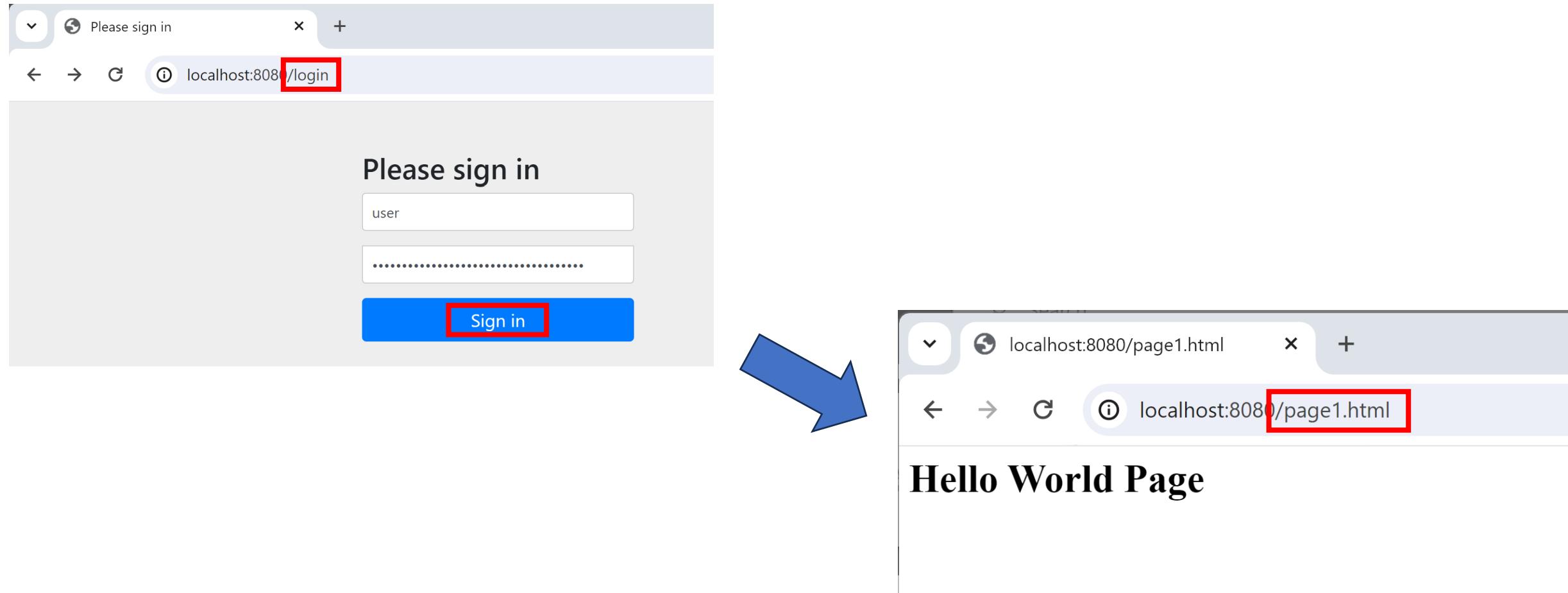
server lookup session in "database" (in memory) from JSESSIONID

Why is it "BAD" to use Cookie SESSIONID ?

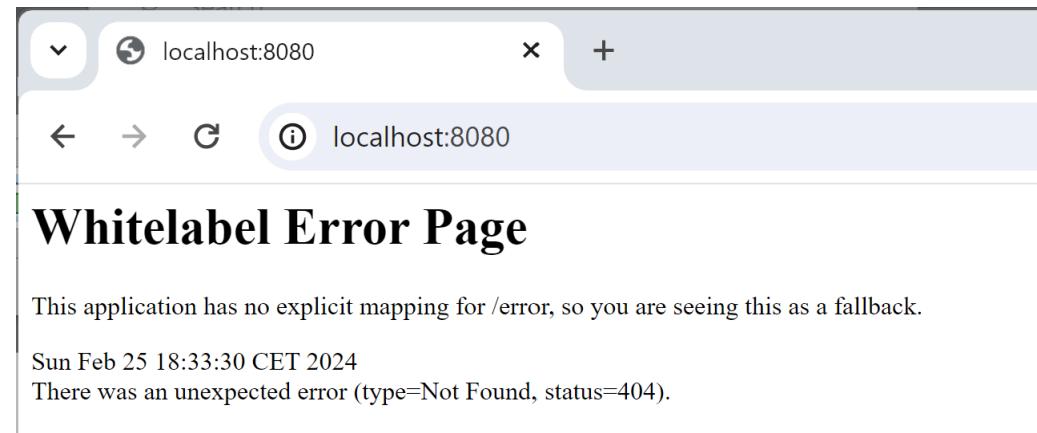
Statefull Server / Database ... SPOF



Click "Sign in" ... redirected to "/page1.html"



Browser Previous Page then Redo "Signin"
=> no redirect
"/" Not Found .. 404



Detailed Analysis of 4 http scenarios

Scenario 1/4 : No authentication

Scenario 2/4 : BAD user/password

Scenario 3/4 : user/password OK => auth success, set Cookie



Scenario 4/4 : clear cookie

Cookies in Browser - Developer Tools

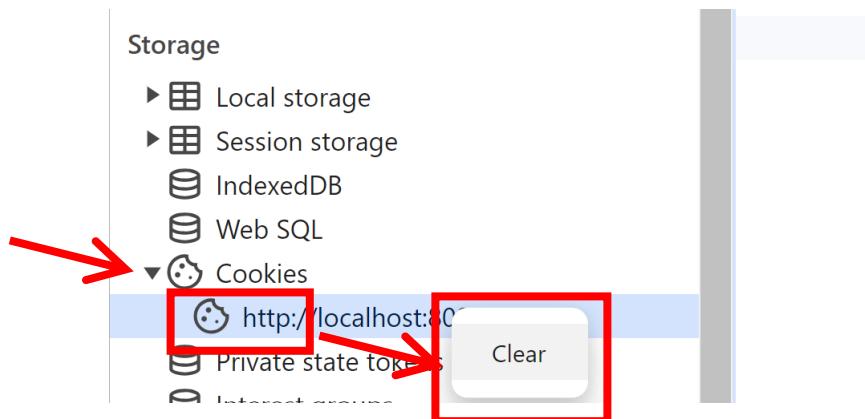
The screenshot shows the Google Chrome Developer Tools interface with the 'Application' tab selected. The left sidebar lists storage types: Manifest, Service worker, Storage, Local storage, Session storage, IndexedDB, Web SQL, Cookies, Interest groups, Shared storage, and Cache storage. The 'Cookies' section is expanded, showing a list of cookies for the domain http://localhost:8080. Three cookies are listed: JSESSIONID, _ga, and metabase.DEVICE. Red arrows point from the title 'Cookies in Browser - Developer Tools' to the 'Application' tab and the 'Cookies' section in the sidebar.

Name	Value	Dom...	Path	Expir...	Size	Http...	Secure	Same...	Partit...	Pri...
JSESSIONID	11E8975AD018694576690C5FE23532B2	local...	/	Sessi...	42	✓				Medi...
_ga	GA1.1.576560884.1694723966	local...	/	2024...	29					Medi...
metabase.DEVICE	7856a6fd-dd9c-495d-880e-0b2ed106aa...	local...	/	2024...	51	✓		Lax		Medi...

Select a cookie to preview its value

Delete Cookies [1/3]

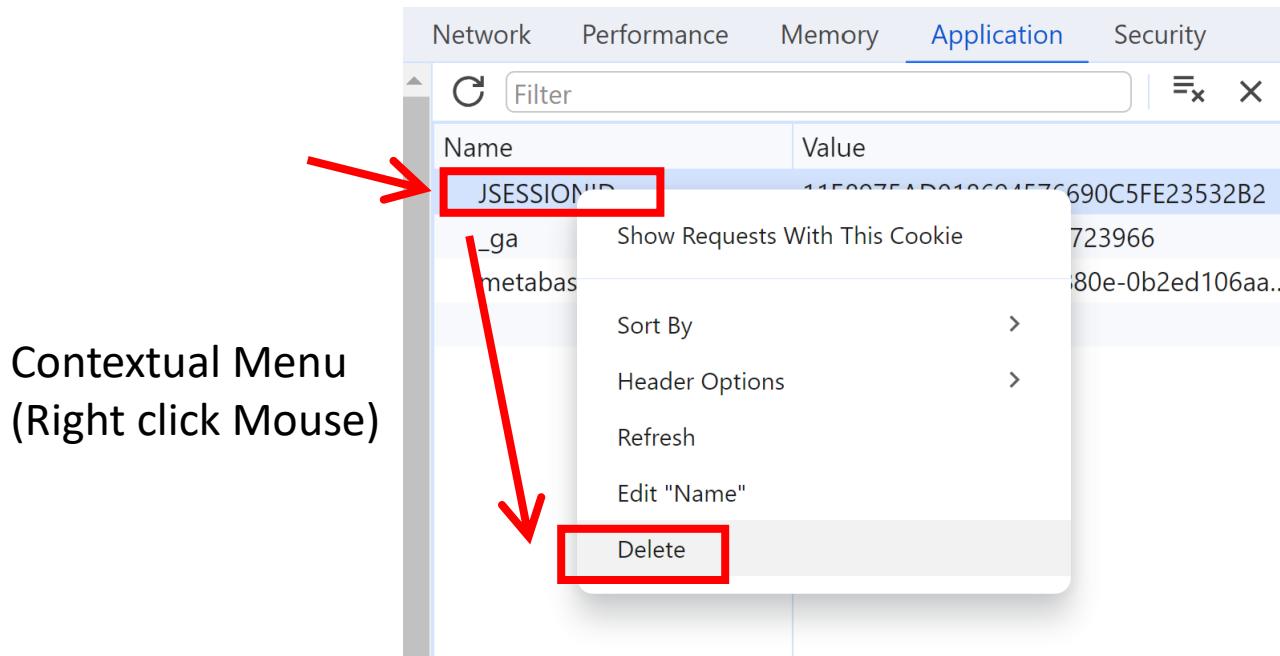
clear all Cookies of given URL site



Contextual Menu
(Right click Mouse)

Delete Cookies [2/3]

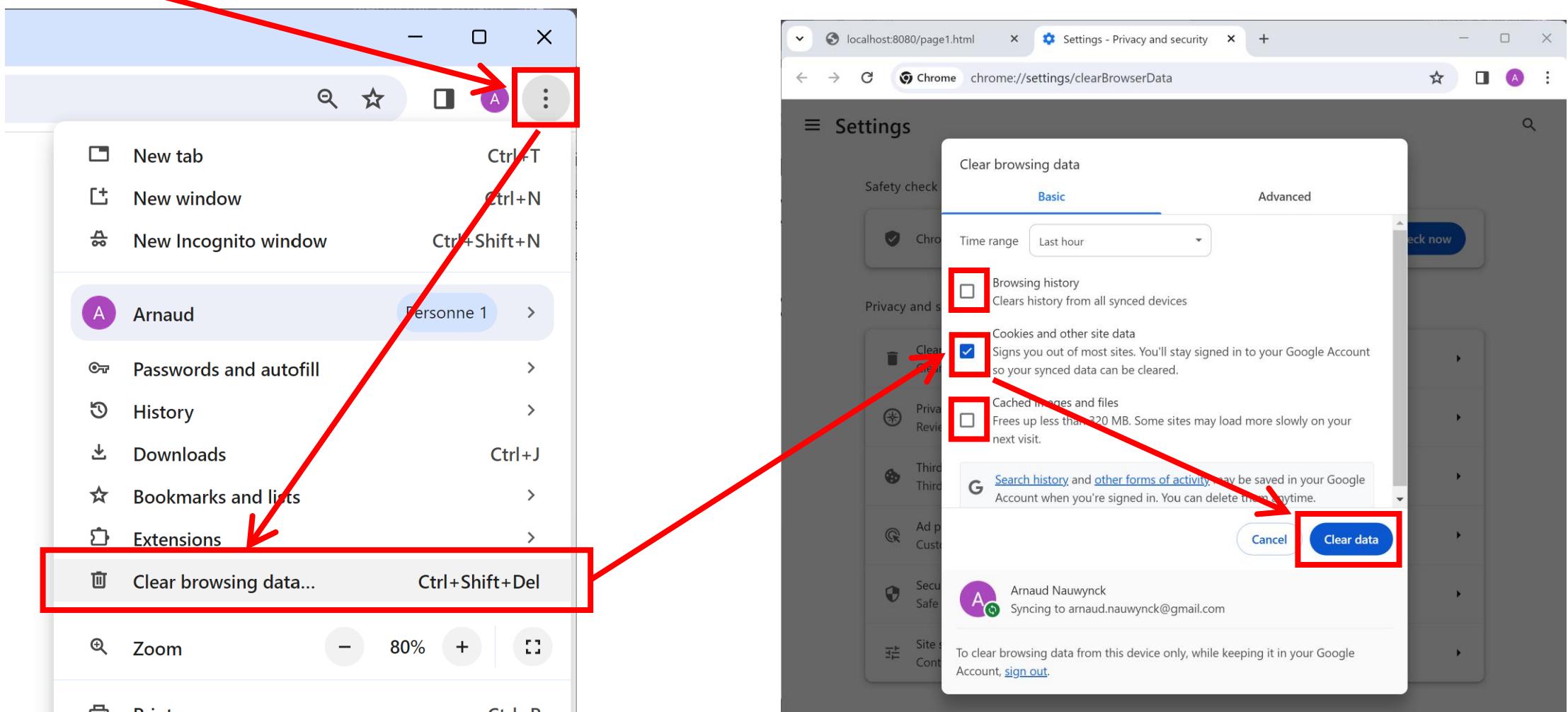
Delete explicitly a single Site-Cookie



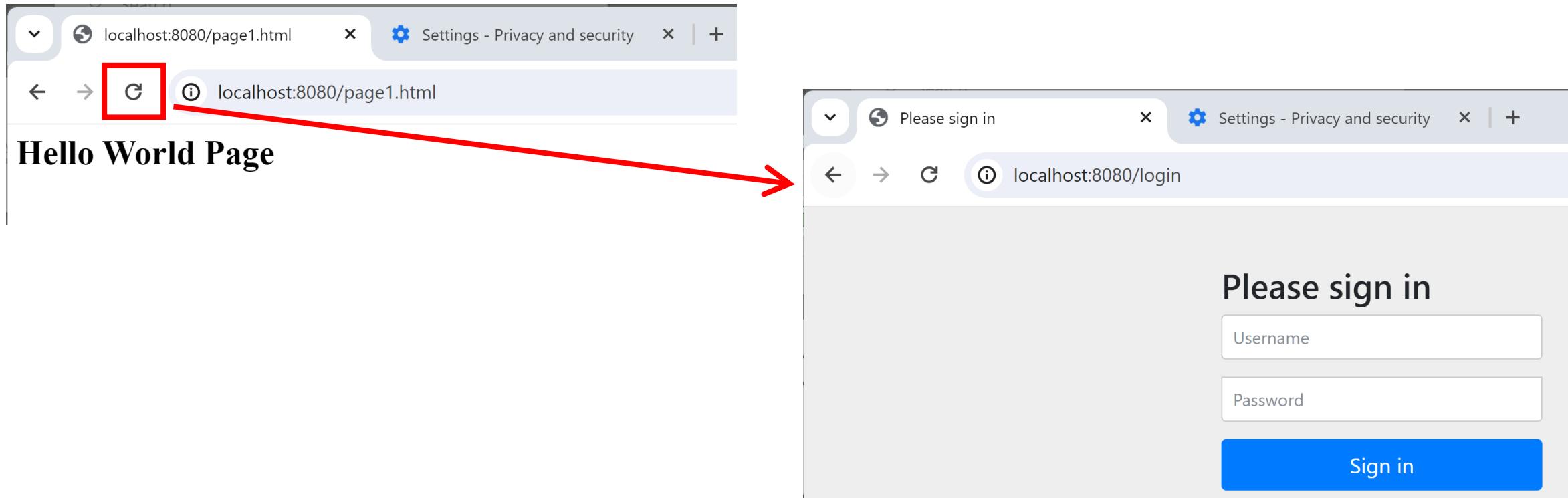
Delete Cookies [3/3]

Clear Browsing Data ...

delete all Cookies (and History) of All sites



After clear "JSESSIONID" cookie
Refresh (F5) /page1.html
=> redirect again to /login



Is there a Better way (Stateless, without cookie) to redirect back from "/login" to "/page1.html" ?

yes, using **http URL query parameter**

http://localhost:8080/login?redirect=page1.html

... but NON replay & CSRF JSESSIONID are more secure for /login

A You

why do springboot web security use set-cookie JSESSIONID in the /login html page ?



ChatGPT

In Spring Boot applications that use Spring Security for web security, the `JSESSIONID` cookie is typically set in the `/login` HTML page for session management purposes. Here's why it's done:

1. **Session Management:** When a user accesses the login page of a Spring Boot application, the server typically creates a session for that user. This session is used to maintain stateful information about the user's interaction with the application throughout their session.
2. **Authentication Flow:** In a typical authentication flow, when a user submits their credentials (username and password) via the login form, Spring Security processes these credentials and validates them. If the credentials are valid, Spring Security establishes an authenticated session for the user.
3. **Session Tracking:** The `JSESSIONID` cookie is a mechanism used by the server to track the user's session. It contains a unique identifier (the session ID) that associates the user's subsequent requests with their session data stored on the server.
4. **Secure Communication:** By setting the `JSESSIONID` cookie in the `/login` HTML page, the server ensures that subsequent requests, including the form submission with the user's credentials, are associated with the correct session. This helps maintain the integrity and security of the authentication process.
5. **Cross-Site Request Forgery (CSRF) Protection:** Additionally, setting the `JSESSIONID` cookie in the login page can also help mitigate CSRF attacks. When combined with CSRF tokens, it adds an extra layer of protection by ensuring that the session associated with the user's authentication is established before any sensitive actions, such as form submissions, are performed.

Overall, setting the `JSESSIONID` cookie in the `/login` HTML page is a standard practice in Spring Security for managing user sessions and ensuring the security of the authentication process.



Other ways to avoid "Basic" user:password credentials ?

using **Dynamic Challenges instead of static Credentials**

using **2FA or MFA (Multi Factor Authentication)**

using **OAuth2**

... then once authenticated, **using JWT Token**

(out of scope for this basic course)

OAuth2 Standard

one password to rule them all

... OAuth2 = internally using an extra indirection level
(but still another user:password at final step)

Using Another level of Indirection...

A screenshot of a Google search results page. The search bar at the top contains the query "any problem can be solved by addind another le". Below the search bar are several navigation links: Images, Videos, Job sites, News, Books, and Finance. A horizontal line separates this from the search results. Below the line, it says "About 1,270,000 results (0.33 seconds)". The main text area starts with "Showing results for any problem can be solved by **adding** another level of indirection". There is also a link to "Search instead for any problem can be solved by addind another level of indirection".

A famous aphorism of Butler Lampson goes: "All problems in computer science can be solved by another level of indirection" (the "fundamental theorem of software engineering"). This is often deliberately mis-quoted with "abstraction layer" substituted for "level of indirection".

Summary

only a very "short" introduction to "Basic Authentication for Web Apps" using Http status code + Http Headers (Cookie, redirect Location, ...)

much more to discover:

- https (TLS) & Certificates, using Public Key Cryptography
- password Hashes, Managed Identities, KeyVault
- CORS, CSRF
- OAuth2
- Authorizations
- Containers, Sql Injection, VPN, Firewall, DMZ, Dos Attacks, Virus, etc, ...

Questions ?

arnaud.nauwynck@gmail.com

APPENDIX

Remark on "_csrf" hidden field
in /login html page

Appendix: Csrf [1/5]

ChatGPT 3.5 ▾

A You

in springboot login html page, explain the input field "_csrf"

ChatGPT

In a Spring Boot login HTML page, the input field "_csrf" is a hidden field used for CSRF (Cross-Site Request Forgery) protection. CSRF attacks involve an attacker tricking a user into unintentionally performing actions on a web application that they are authenticated with. This hidden field helps to prevent such attacks.

Here's how it typically works:

1. When a user requests the login page, the server generates a unique token and stores it in the user's session.
2. This token is then included as the value of the "_csrf" input field in the login form. However, since it's hidden, the user won't see it or interact with it directly.
3. When the user submits the login form, the CSRF token is sent along with the other form data to the server.
4. Upon receiving the request, the server compares the CSRF token submitted with the one stored in the user's session. If they match, the request is considered legitimate, and the login process proceeds.
5. If the CSRF tokens don't match, the server can reject the request, assuming it might be a CSRF attack, thus protecting the user's session from being hijacked.

This mechanism helps to ensure that the form submission originates from the legitimate user and not from a malicious website or script attempting to perform unauthorized actions on the user's behalf.

Appendix: Csrf [2/5]

ChatGPT 3.5 ▾

A

You

suppose that there is NO _csrf field checked, can you show me an attack



ChatGPT

Sure, I can describe a potential CSRF attack scenario without the presence of a CSRF token (_csrf) field.

Let's say you have a Spring Boot login form without CSRF protection, and the form looks like this:

```
html
Copy code

<form action="/login" method="post">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username"><br><br>
    <label for="password">Password:</label>
    <input type="password" id="password" name="password"><br><br>
    <input type="submit" value="Login">
</form>
```

An attacker could create a malicious website with the following HTML:

Appendix: Csrf [3/5]

An attacker could create a malicious website with the following HTML:

```
html
Copy code

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Malicious Site</title>
</head>
<body>
    <h1>Click the Button!</h1>
    <form action="http://victim-site.com/login" method="post">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" value="attacker"><br><br>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password" value="password"><br><br>
        <input type="submit" value="Login">
    </form>
    <script>
        document.forms[0].submit(); // Automatically submits the form
    </script>
</body>
</html>
```

Appendix: Csrf [4/5]

Now, if a user who is authenticated on your application visits the attacker's website while still logged in, the form on the attacker's website will automatically submit, sending a POST request to your application's login endpoint (/login) with the user's credentials. Since there's no CSRF token validation, your application will accept the request, thinking it's a legitimate login attempt by the user.

As a result, the attacker can effectively trick the user into unintentionally performing actions on your application, such as logging in with their credentials without their knowledge or consent. This is the essence of a CSRF attack.



Appendix: Csrf [5/5]

A You

so in which attacks can it occurs ?



ChatGPT

CSRF (Cross-Site Request Forgery) attacks can occur in various scenarios where an attacker can manipulate a user's web browser to perform unintended actions on a different website where the user is authenticated. Here are some common scenarios where CSRF attacks can occur:

1. **Form Submissions:** CSRF attacks often involve crafting malicious forms on attacker-controlled websites. When a user who is authenticated on another site visits the attacker's site, the form submission can trigger actions on the authenticated site.
2. **Clickjacking:** In clickjacking attacks, attackers overlay transparent or opaque elements on top of legitimate buttons or links. When a user clicks on what appears to be a harmless element, they inadvertently trigger actions on a different site where they are authenticated.
3. **Image Tags:** Attackers can embed images in web pages with image tags () that have URLs pointing to actions on a different site. When a user loads the page containing the image, the browser sends requests to the specified URLs, potentially executing unauthorized actions.
4. **Script Injections:** If an attacker can inject malicious scripts into a vulnerable website, these scripts can then manipulate the user's browser to perform actions on other sites where the user is authenticated. This often occurs in the context of Cross-Site Scripting (XSS) vulnerabilities.
5. **Malicious Links:** Attackers can distribute malicious links via email, social media, or other channels. When a user clicks on such a link while authenticated on another site, the browser can execute actions on the authenticated site without the user's consent.
6. **Embedded Content:** Content embedded in iframes or other embedded elements can execute CSRF attacks if it contains scripts or forms that trigger actions on authenticated sites when loaded by a user's browser.

These are just a few examples of how CSRF attacks can occur. The common theme is that the attacker exploits the trust established between the user and the authenticated website to perform