Web Technologies

Leonardo Querzoni
querzoni@diag.uniroma1.it

Sapienza University of Rome

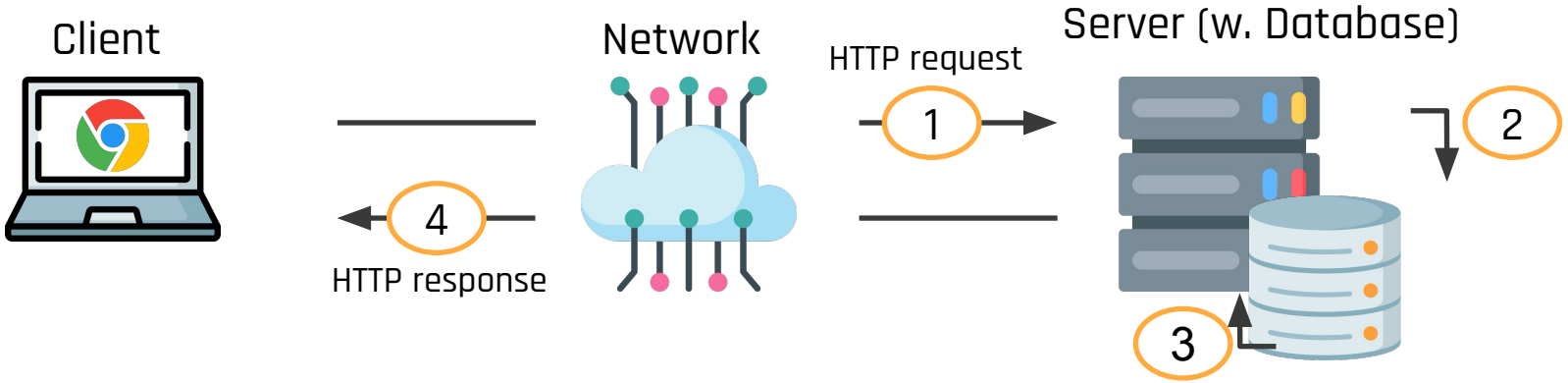
Credits

These slides have been designed by Emilio Coppa (coppa@luiss.it, Luiss University) on the basis of teaching material originally created by:

- Marco Squarcina (marco.squarcina@tuwien.ac.at), S&P Group, TU WIEN
- Mauro Tempesta (mauro.tempesta@tuwien.ac.at), S&P Group, TU WIEN
- Fabrizio D'Amore (damore@diag.uniroma1.it), Sapienza University of Rome

Introduction to HTTP

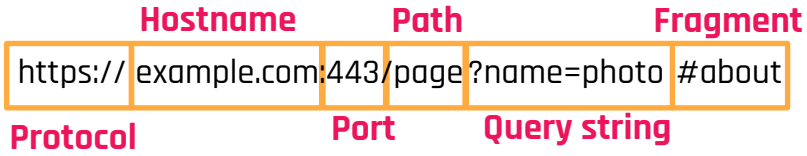
Anatomy of a Typical Web Application



1. The user request a webpage with dynamically generated content
2. The web application queries the database for user's data
3. The data from the database is used to generate page content
4. The page is rendered by the client's browser

Uniform Resource Locator (URL)

URLs are identifiers for documents on the Web



- Some elements are optional: port, query string, fragment
- When reserved characters (like space : ? /) need to be used in the URL, they must be URL-encoded:
  - %20 = space
  - %2F = /
  - ...

Example of encoding:

https://example.com/page?name=my%20page

The HTTP Protocol

- HTTP (Hypertext Transfer Protocol) defines the structure of the communication between client and web server
- Properties:
  - **Stateless**: different requests are processed independently from each other
    - Cookies are used to implement stateful applications on top of HTTP
  - **Not encrypted:** HTTP traffic can be read and modified on the network without the communication parties to notice it
  - Default port for HTTP is 80

The HTTPS Protocol

- HTTPS is the secure variant of HTTP:
  - Essentially, HTTP traffic delivered over a TLS connection
  - Default port is 443
- Security properties:
  - **Confidentiality**: content of the traffic cannot be inspected as it travels on the network
  - **Integrity**: content of the traffic cannot be modified as it travels on the network
  - **Authentication**: the client can verify that it is communicating with the expected server

HTTP Request

Most common HTTP Methods:

GET should have no side effects, used to retrieved data

POST possible side effect, used to insert/update remote resources

HEAD same as GET but without response body


```
POST /login HTTP/2
Host: example.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.16; rv:85.0)
Gecko/20100101 Firefox/85.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Content-Type: application/x-www-form-urlencoded
Content-Length: 71
Origin: https://example.com
Connection: keep-alive
Referer: https://example.com/login
Upgrade-Insecure-Requests: 1

user=ugo&csrf_token=IjljMjlkMDE4ODJmZWZlODhf
```

HTTP headers

Blank line

Optional request body (empty for GET)

HTTP Response

HTTP version — **Status code, where first digit defines the message type: 2: OK, 3: Redirect, 4: Client Error, 5: Server Error** — **Reason phrase**

```
HTTP/2 200 OK
Server: nginx
Date: Mon, 22 Feb 2021 15:38:46 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 10459
Vary: Cookie
Set-Cookie: session=apU8ig7aeonYoLtOKOC9R5D5fY; Secure; HttpOnly; Path=/
Strict-Transport-Security: max-age=63072000

<html>
   <body>login successful!</body>
</html>
```

Cookie

Blank line

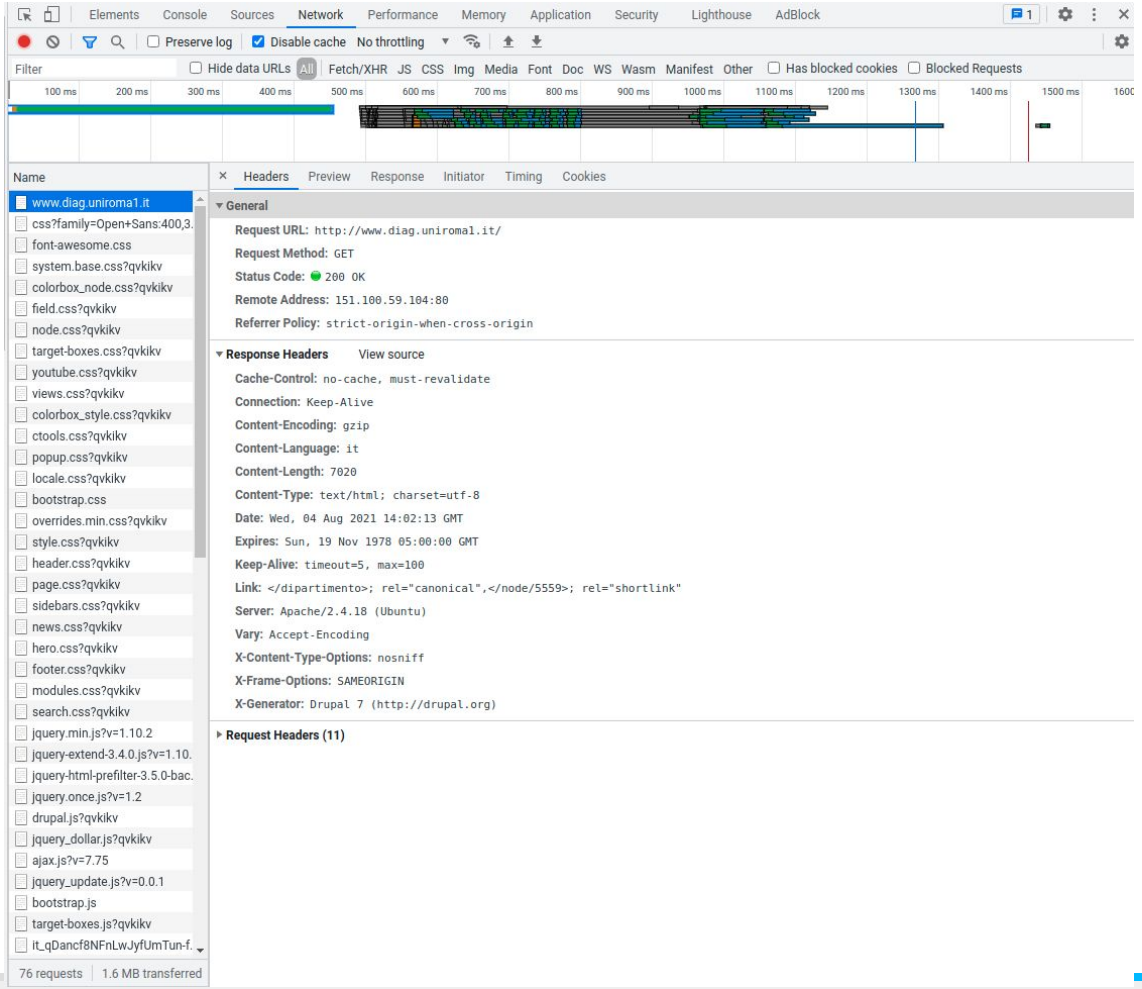
HTTP headers

(Optional) response body

Opening a page with Google Chrome

Google Chrome: Developers tools

1. Select Network
2. Refresh the page
3. Choose a request
4. Inspect request and response



General

Request URL: http://www.diag.uniroma1.it/

Request Method: GET

Status Code: 200 OK

Remote Address: 151.100.59.104:80

Referrer Policy: strict-origin-when-cross-origin

Request Headers View source

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=q=0.9

Accept-Encoding: gzip, deflate

Accept-Language: it-IT,it;q=0.9,en-US;q=0.8,en;q=0.7

Cache-Control: no-cache

Connection: keep-alive

Cookie: _ga=GA ; has_js=1; LtpaToken= saW8gQ29wcGEvT1U9RGlwLUluZm9ybWF0aWNhL0URpcGFydGltZW50aS9PVT1EaWRhdHRpY2EvT1U9QXRlbmVvL089VW5pcm9tY

DNT: 1

Host: www.diag.uniroma1.it

Pragma: no-cache

Upgrade-Insecure-Requests: 1

User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.107 Safari/537.36

Response Headers View source

Cache-Control: no-cache, must-revalidate

Connection: Keep-Alive

Content-Encoding: gzip

Content-Language: it

Content-Length: 7020

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

Date: Wed, 04 Aug 2021 14:02:13 GMT

Expires: Sun, 19 Nov 1978 05:00:00 GMT

Keep-Alive: timeout=5, max=100

Link: </dipartimento>; rel="canonical",</node/5559>; rel="shortlink"

Server: Apache/2.4.18 (Ubuntu)

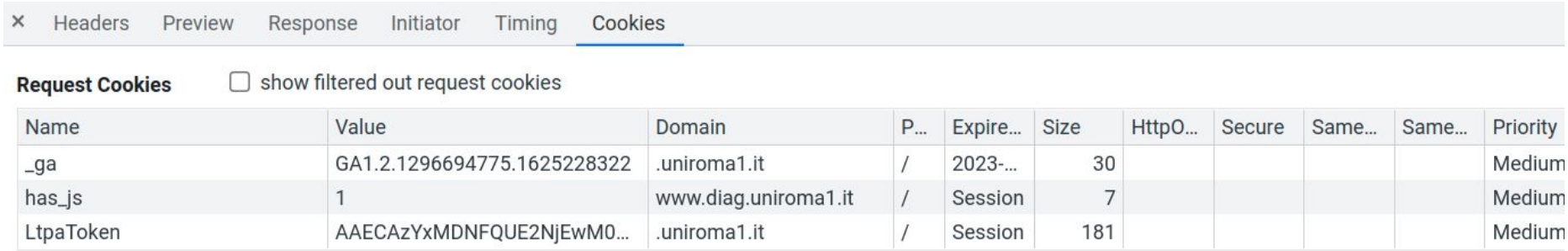
Vary: Accept-Encoding

X-Content-Type-Options: nosniff

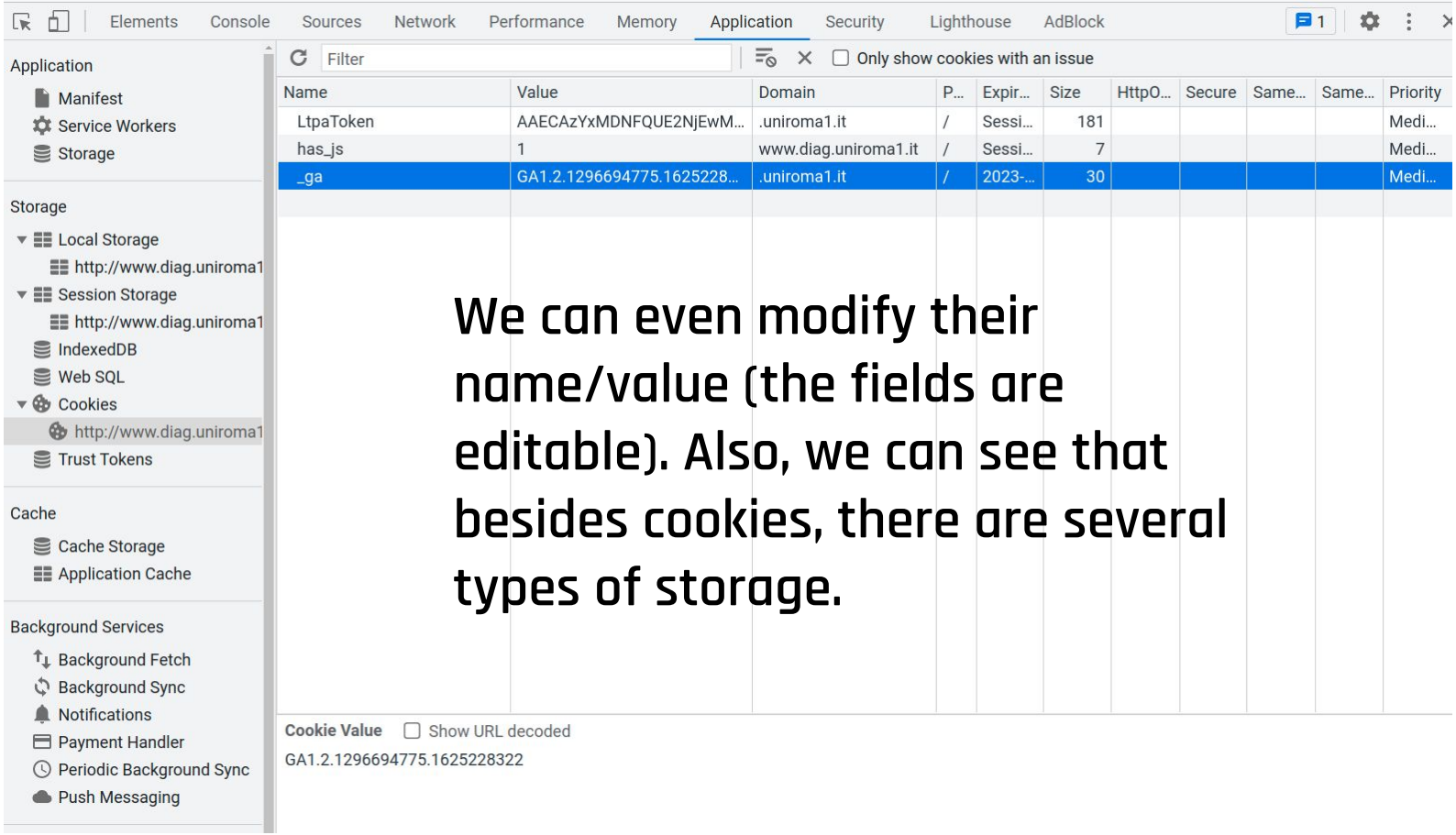
X-Frame-Options: SAMEORIGIN

X-Generator: Drupal 7 (http://drupal.org)

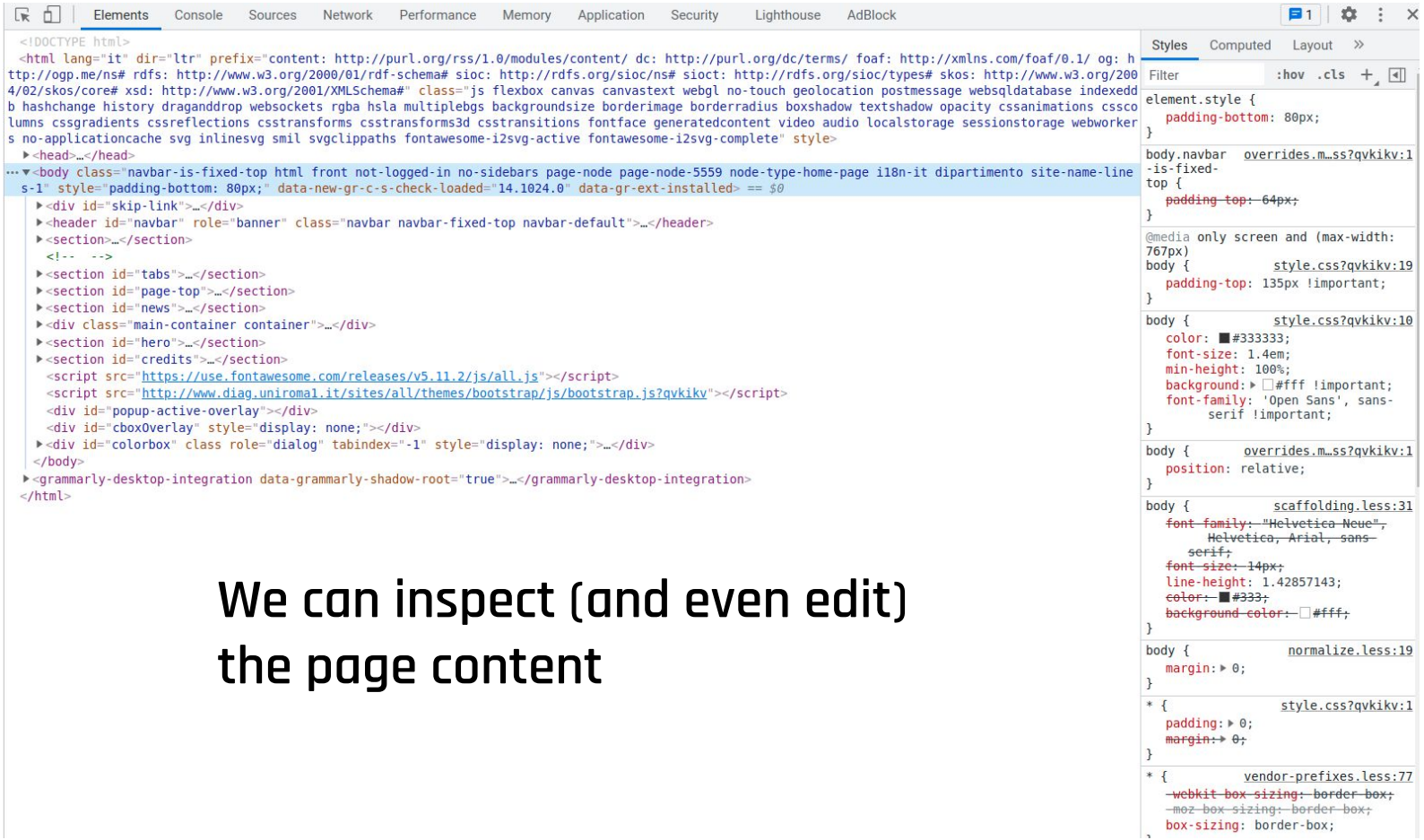
We can see which cookies are used by the page



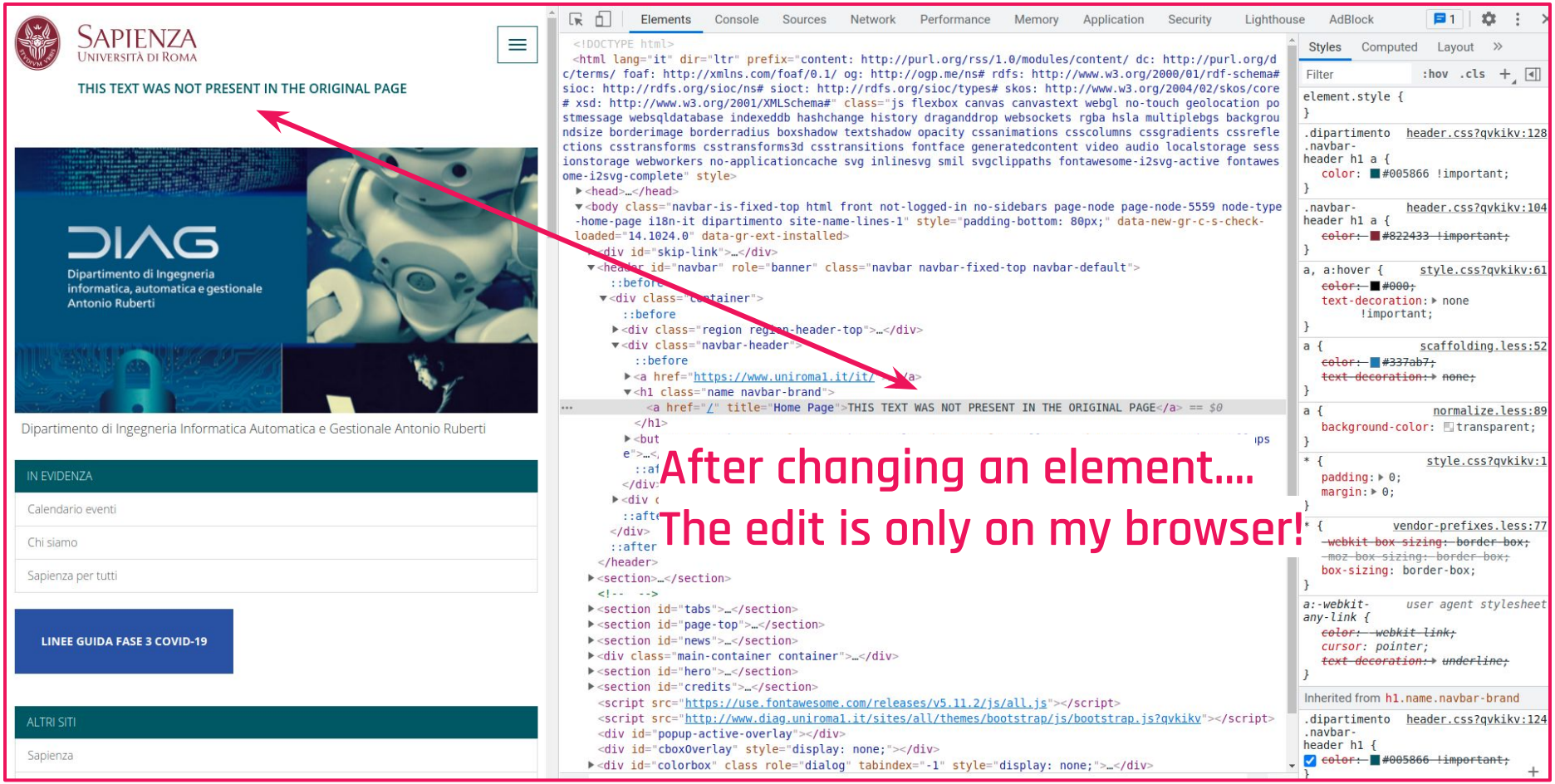
Name	Value	Domain	P...	Expire...	Size	HttpO...	Secure	Same...	Same...	Priority
_ga	GA1.2.1296694775.1625228322	.uniroma1.it	/	2023-...	30					Medium
has_js	1	www.diag.uniroma1.it	/	Session	7					Medium
LtpaToken	AAECAzYxMDNFQUE2NjEwM0...	.uniroma1.it	/	Session	181					Medium



We can even modify their name/value (the fields are editable). Also, we can see that besides cookies, there are several types of storage.



We can inspect (and even edit) the page content


SAPIENZA
UNIVERSITÀ DI ROMA
THIS TEXT WAS NOT PRESENT IN THE ORIGINAL PAGE
DIAG
Dipartimento di Ingegneria informatica, automatica e gestionale Antonio Ruberti
Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti
IN EVIDENZA
Calendario eventi
Chi siamo
Sapienza per tutti
LINEE GUIDA FASE 3 COVID-19
ALTRI SITI
Sapienza
Elements
Console
Sources
Network
Performance
Memory
Application
Security
Lighthouse
AdBlock
Styles
Computed
Layout
Filter
:hov .cls
THIS TEXT WAS NOT PRESENT IN THE ORIGINAL PAGE == $0
After changing an element....
The edit is only on my browser!

The Languages of the Web: Client-Side

- HTML
  - Defines the structure of the webpage
- CSS
  - Defines the styling of the page
- JavaScript:
  - Allows to add dynamic interactive effects to the webpage (e.g., react to user interactions)

```
<html>
  <body>
    <p>hello!</p>
  </body>
</html>
```

```
p {
  color: red;
}
```

```
let d = window.document;
let p = d.getElementsByTagName('p')[0];
p.addEventListener('click', function () {
  this.style.color = 'blue';
});
```

The Languages of the Web: Server-Side

- Virtually every programming language can be used on the server-side (even C!)

- Most common server-side languages in 2020:
  - **Python**, NodeJS (JavaScript), Java, C#, **PHP**

- The server-side language is used to implement your web application:
  - Session management of users
  - Interaction with the database
  - Generation of the response pages
  - ...

Quick and dirty HTTP server

A quick but **unsafe** way of spawning a HTTP server is:

> python3 -m http.server 8000

Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...

NOTE: the current working directory is the root for the web server

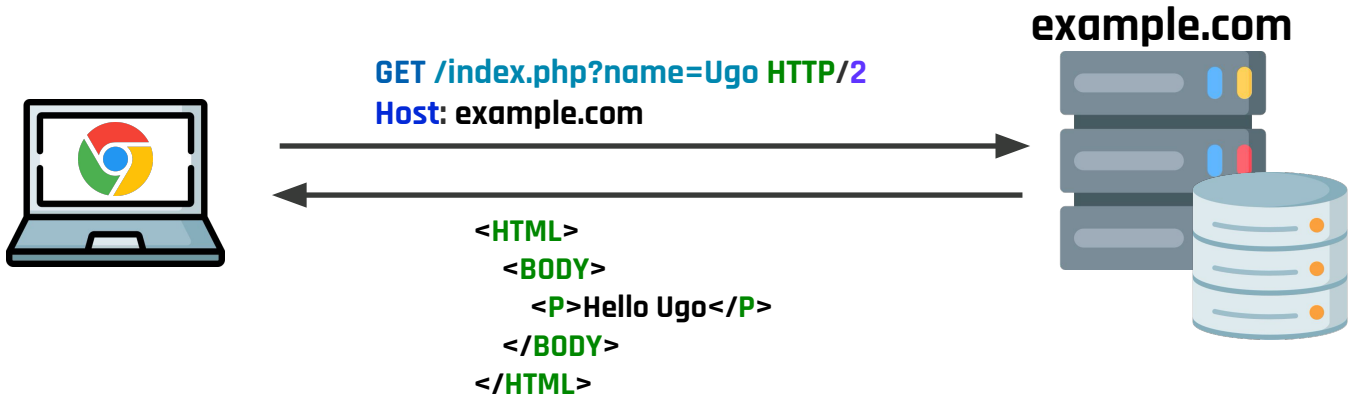
PHP: Hypertext Preprocessor - Basics

- We will use PHP in some of the examples
- It is a server-side scripting language with C-like syntax
- HTML and PHP code can be intermingled in the same file
- Variable names start with **$**
- Command **echo** can be used to print the value of an expression
- The operator **.** denotes string concatenation
- Important global associative arrays (i.e., dictionaries):
  - **$_GET**: parameters provided via the URL query string
  - **$_POST**: parameters provided in the body of a request
  - **$_SESSION**: parameters stored in a PHP session (preserved across multiple requests)

PHP: Hypertext Preprocessor - Example

```
<HTML>
  <BODY>
    <P><?php echo "Hello " . $_GET["name"]; ?></P>
  </BODY>
</HTML>
```

index.php



Quick and dirty HTTP+PHP server

A quick but **unsafe** way of spawning a HTTP/PHP server is:

> php -S 0.0.0.0:8000

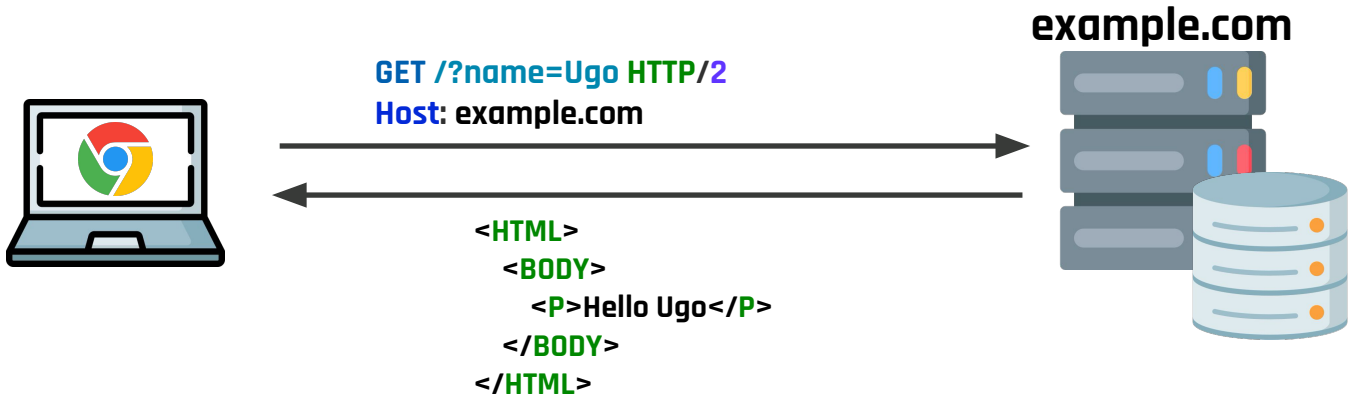
```
[Mon Oct 25 18:42:06 2021] PHP 7.4.3 Development Server (http://0.0.0.0:8000) started
[Mon Oct 25 18:42:28 2021] 127.0.0.1:37502 Accepted
[Mon Oct 25 18:42:28 2021] 127.0.0.1:37502 [200]: GET /
[Mon Oct 25 18:42:28 2021] PHP Notice:  Undefined index: name in index.php on line 3
[Mon Oct 25 18:42:28 2021] 127.0.0.1:37502 Closing
[Mon Oct 25 18:42:37 2021] 127.0.0.1:37506 Accepted
[Mon Oct 25 18:42:37 2021] 127.0.0.1:37504 [200]: GET /?name=ugo
[Mon Oct 25 18:42:37 2021] 127.0.0.1:37504 Closing
```

NOTE: the current working directory is the root for the web server

Quick and dirty HTTP/Python server

```
from flask import Flask, request

app = Flask(__name__)

@app.route("/")
def hello_world():
  return "<html>\n<body>\n<p>Hello %s</p></body></html>"
                % request.args.get('name')
```

app.py



Quick and dirty HTTP/Python server (2)

> pip3 install flask

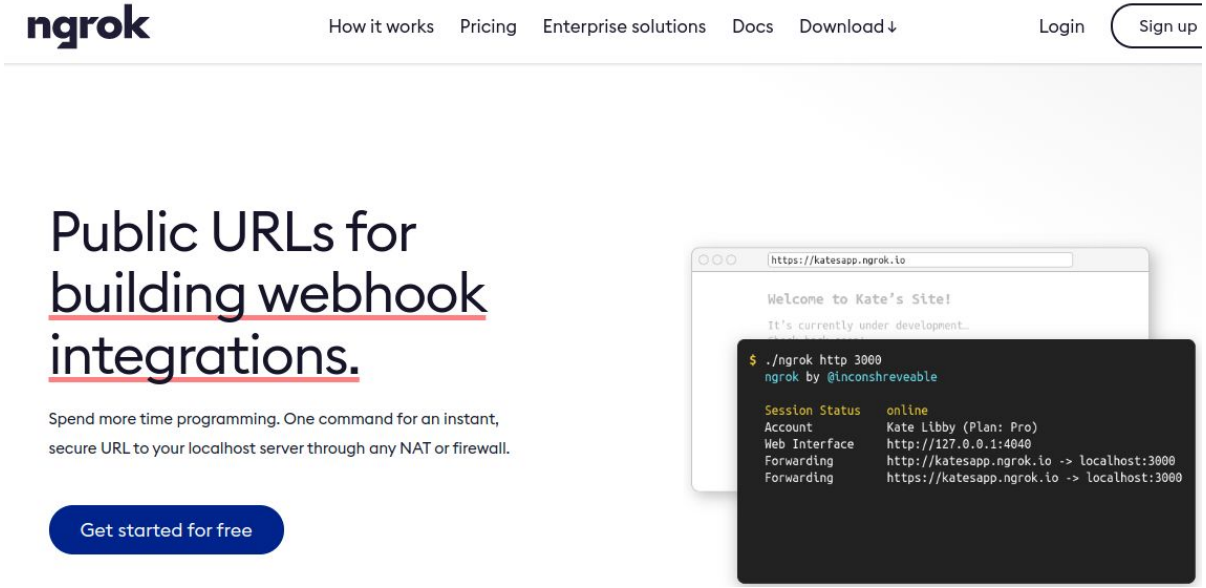
> python3 -m flask run

```
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [25/Oct/2021 18:57:17] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [25/Oct/2021 18:57:17] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [25/Oct/2021 18:57:29] "GET /?name=ugo HTTP/1.1" 200 -
```

How to make our server reachable from the internet?

Assuming that we are just talking about development/CTF deployment... we can use **ngrok** to make our server reachable (possibly even with HTTPS). This will work even without a firewall (port forwarding) and without a (dynamic) domain.

https://ngrok.com/



ngrok

1. Spawn your local HTTP server on port X

2. Download and install ngrok (available as a snap package!)

3. Register an account on ngrok.com and get the authtoken

4. Configure the authtoken:
   > ngrok authtoken <auth_token>

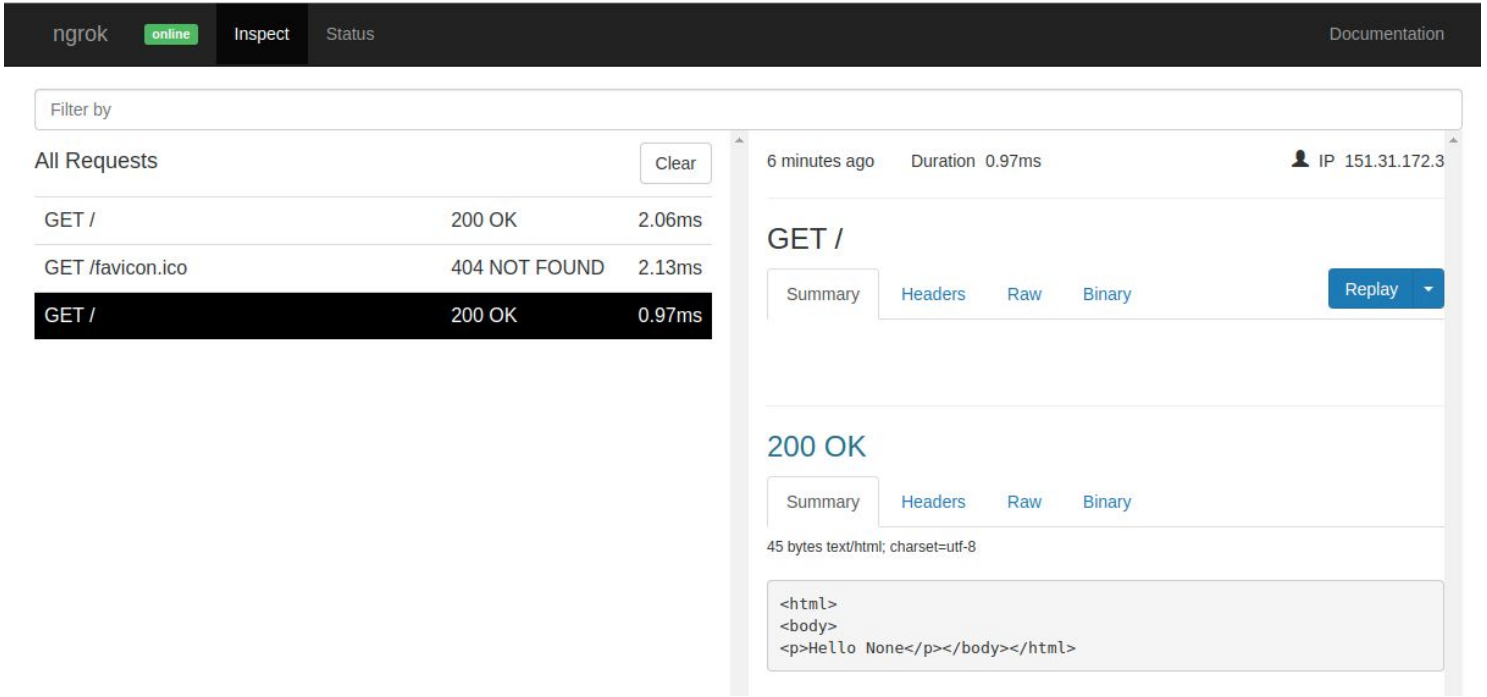
ngrok (2)

5. Run ngrok for http X: > **ngrok http X**

```
Session Status                online
Account                       ercoppa (Plan: Free)
Version                       2.3.40
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://2781-151-31-172-3.ngrok.io -> http://localhost:5000
Forwarding                    https://2781-151-31-172-3.ngrok.io -> http://localhost:5000

Connections                   ttl     opn     rt1     rt5     p50     p90
                              3       0       0.04    0.01    0.00    0.00

HTTP Requests
-------------

GET /                         200 OK
GET /favicon.ico              404 NOT FOUND
GET /                         200 OK
```

ngrok (3)

6. Get statistics from the ngrok web interface



Training challenge #12

URL: https://training12.webhack.it

NOTE: **THE CHALLENGE IS LIVE!**
TRY IT TO LEARN!

Description:

At WebHackIT we like ping pong! Can we play it for fun?

Ping Pong!

I expect to find "[redacted]" at page at:

https://your-ngrok.url

Fetch the page

WebHackIT

Analysis

- It is a web application that ask us a URL
- It is saying that, at the URL, we should serve a page with a specific content
- The url should be from ngrok.io

....let's try to serve a page with this content!

Solution

....too easy! Just look at the previous slides!

HTTP/2 and HTTP/3

HTTP2

- Major revision of the HTTP network protocol. It was derived from the earlier experimental SPDY protocol, originally developed by Google. RFC 7540.

- Main goals:
  - Provide a negotiation mechanism to select HTTP/1.1, 2.0, or other non-HTTP protocols.
  - High-level compatibility with HTTP/1.1
  - Decrease latency to improve page load speed:
    - data compression of HTTP headers
    - HTTP/2 Server Push
    - pipelining of requests
    - fixing the **head-of-line blocking problem** in HTTP 1.x
    - multiplexing multiple requests over a single TCP connection
  - Support common existing use cases of HTTP, such as desktop web browsers, mobile web browsers, web APIs, web servers at various scales, proxy servers, etc.

Reference: HTTP2 Explained

Head-of-line blocking problem in HTTP 1.x

HTTP pipelining is a way to send another request while waiting for the response to a previous request. The idea is to avoid to perform several parallel requests since the setup time for each request could be huge.

Problem:

- suppose request B is “queued” (in the pipeline) after request A
- what happens if request A requires a long processing time?

Hard to choose how to “queue” the requests. Modern browsers disable pipelining...

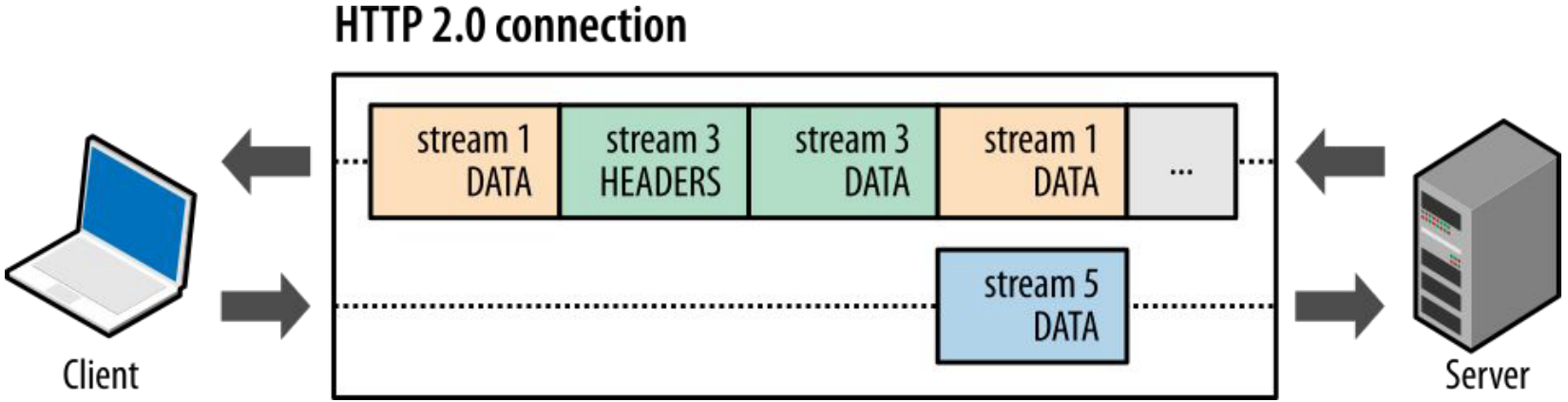
HTTP2: some key ideas

- The client can request an upgrade of the connection using the HTTP/1 Request Header. If the server speaks HTTP/2, it sends a “101 Switching” status and from then on it speaks HTTP2 on that connection.

- HTTP2 is a binary protocol, while HTTP is “based” on ASCII. Web applications do not require changes, everything is done by the browser/server.

- The binary protocol allows to efficiently perform multiplexing within one connection:
  - **Stream**: bidirectional bytes flow within a connection, carrying one or more messages.
  - **Message**: sequence of frames that map to a logical request or response message.
  - **Frame**: smallest unit of communication in HTTP/2, each containing a frame header, which at a minimum identifies the stream to which the frame belongs.

HTTP/2: request and response multiplexing



[image credits]

HTTP/3 and QUIC

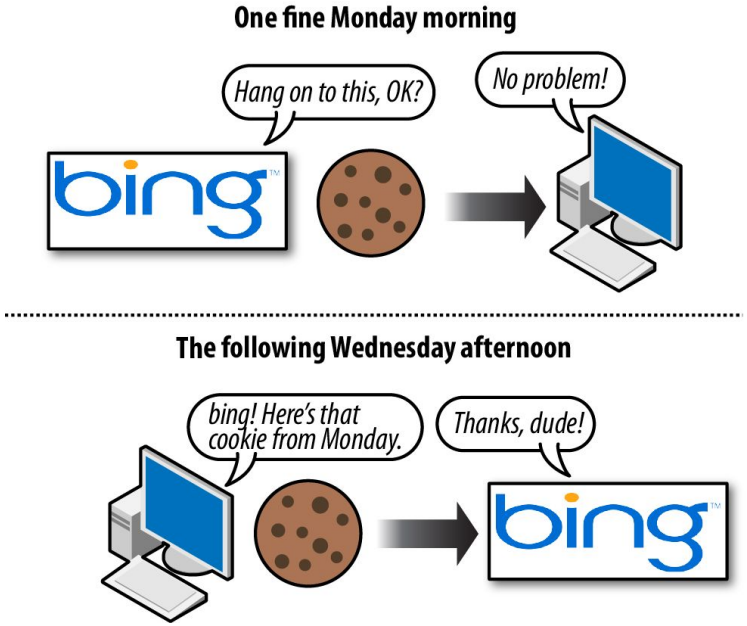
- HTTP/2 addresses **head-of-line blocking** (HOL) through request multiplexing, which eliminates HOL blocking at the application layer, but **HOL still exists at the transport TCP layer**!
- Third revision of HTTP, officially supported since 2022 . Backward compatible: same request methods, status codes, and message fields.
- It is based on QUIC, a general-purpose transport layer network protocol designed by Google, which come with two main features:
  - **Switch from TCP to UDP**: this significantly the overhead from the protocol stack, however, now it is up to protocol to recover from transmission errors (a packet is lost). This helps with HOL.
  - **Integrate into the protocol exchange of setup keys and supported protocols part of the initial handshake process**: this reduces overhead for the setup of TLS, which was not optimal in HTTP/2.

Cookies

HTTP(S) Sessions

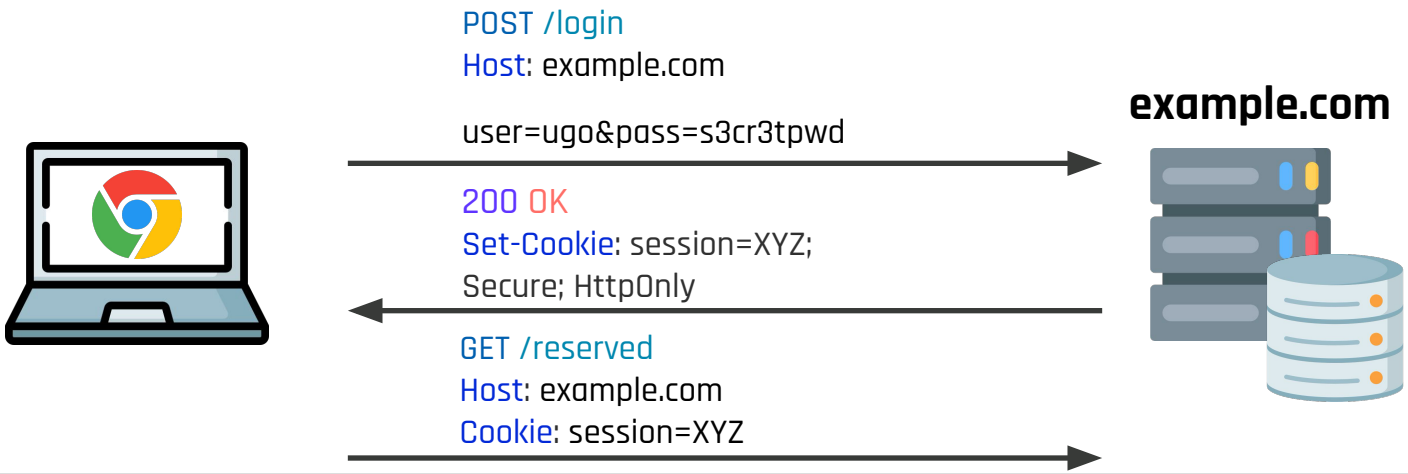
- HTTP(S) is a stateless protocol
  - Requests are independent from each other
  - What if user wants to stay logged in?

- Session concept
  - Session data is stored on the server with a unique session ID
  - Client attaches the session ID to each request
  - Attacker can hijack an (in)active session and impersonate user if session tokens are not properly protected!



Storing Info in Browser with Cookies

- Sessions are typically implemented on top of cookies
- Cookies set by websites are automatically attached by the browser to subsequent requests to the same website
- Cookie attributes (e.g., Domain, Path, Secure, HttpOnly) can be used to customize the cookie behavior
- A cookie is identified by the triplet (name, domain, path)



Cookies in practice

Setting a new cookie (client-side with Javascript):

document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";

where:

- **username** is the key (string)
- **John Doe** is the value (string) associated with the key
- **Thu, 18 Dec 2013 12:00:00 UTC** is the expiration date. When missing, the cookie expires at the end of the session
- **path=/** is the path the cookie belongs to. By default, the cookie belongs to the current page.

Cookies in practice (2)

Read all cookies:

```
let x = document.cookie;
```

Delete a cookie:

```
document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";
```

...you just set the expiration date to a past date.

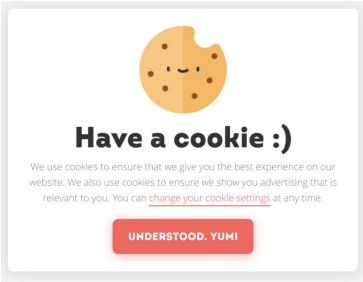
Cookies in practice (3)

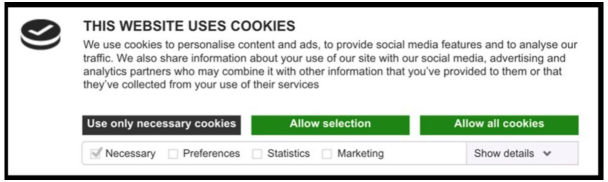
Handling cookies in PHP (server side):

```
setcookie("user", "John Doe", time() + (86400 * 30), "/"); // we set a cookie

echo $_COOKIE["user"]; // we get the value for the cookie

setcookie("user", "", time() - 3600); // delete the cookie
```

What are Cookies used for?

- **Authentication**
  - The cookie proves that the client previously authenticated correctly
- **Personalization**
  - Helps the website recognize the user from a previous visit
- **Tracking**
  - Follow the user from site to site; learn his/her browsing behavior, preferences, and so on

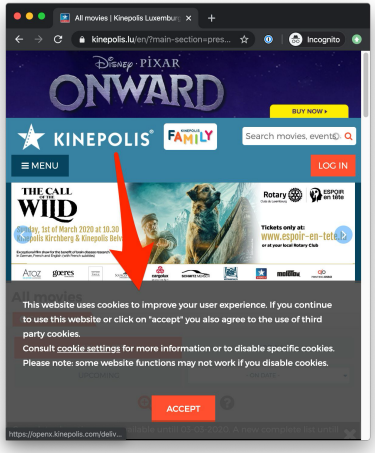


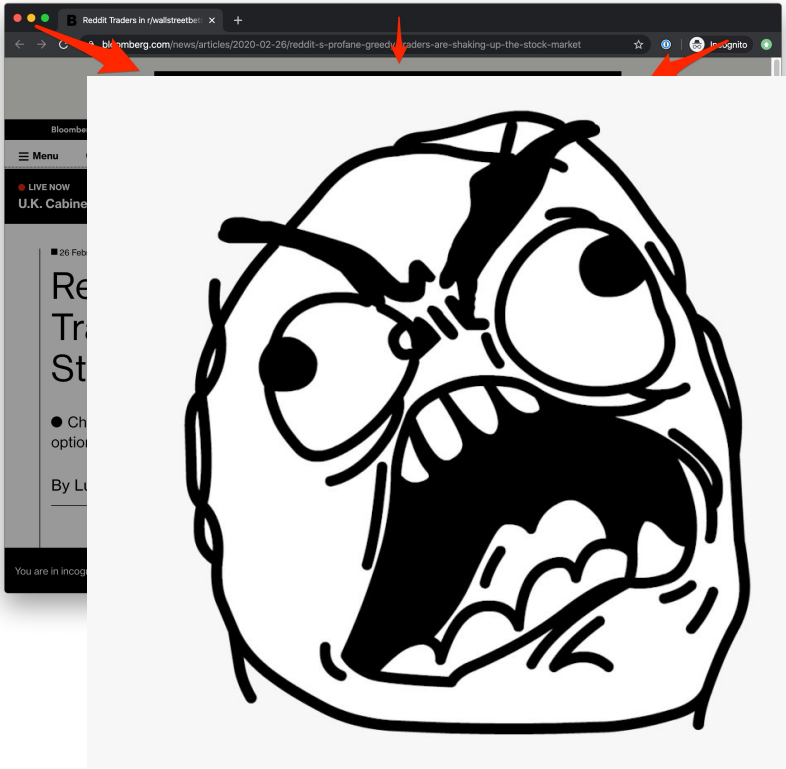


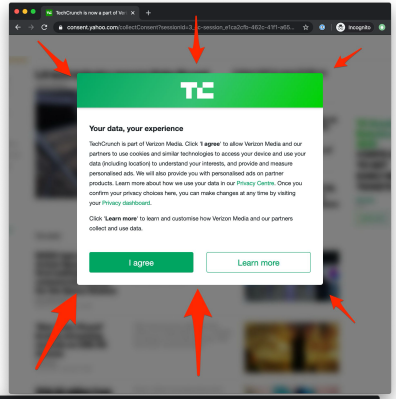
Third-party cookies

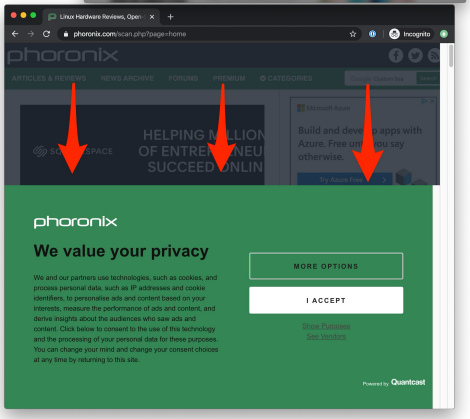
- a page can host contents coming from other web servers
- cookies that are sent by these servers are named **third-party cookies**
- there are organizations operating in the advertisement that use third-party cookies for tracking users across different sites, allowing ads consistent to user profile
- This may be a huge privacy concern.
- Several countries have issued laws on the topic. UE have regulated this matter...

Cookie Banners









Source: https://ma.ttias.be/loophole-cookie-notices/

Storage

Web Storage (or DOM Storage)

Modern browsers allow a web application to store (key, value) pairs on the client:

- **Session Storage**: kept only for the current session
- **Local Storage**: permanent across sessions

The key and value can only be strings. The maximum size for the whole storage is typically 5MB, however, it depends from the browser implementation.

Web Storage is NOT encrypted: e.g., Firefox uses a SQLite file.

Cookie vs Web Storage

They are similar but different:

- Cookies keep track of data in the client for the server (they are attached to each request!). Web Storage keeps track of data only for the client: the server cannot access it (however, the client may still send its content to the server...).

- Cookies have a maximum size of 4KB. Web Storage is designed with a larger capacity in mind.

- Not always clear what happens if two tabs edit the same cookie at the same time. Web Storage uses DB transactions to deal with concurrent operations.

- Cookies come with very old API, which may lead to some security risks. API for Web Storage were designed later and “should” be better.

Web Storage in practice

Javascript (client-side):

```
localStorage.setItem("lastname", "Smith");

console.log(localStorage.getItem("lastname"));

localStorage.removeItem("lastname");
```

Same API for **sessionStorage**.

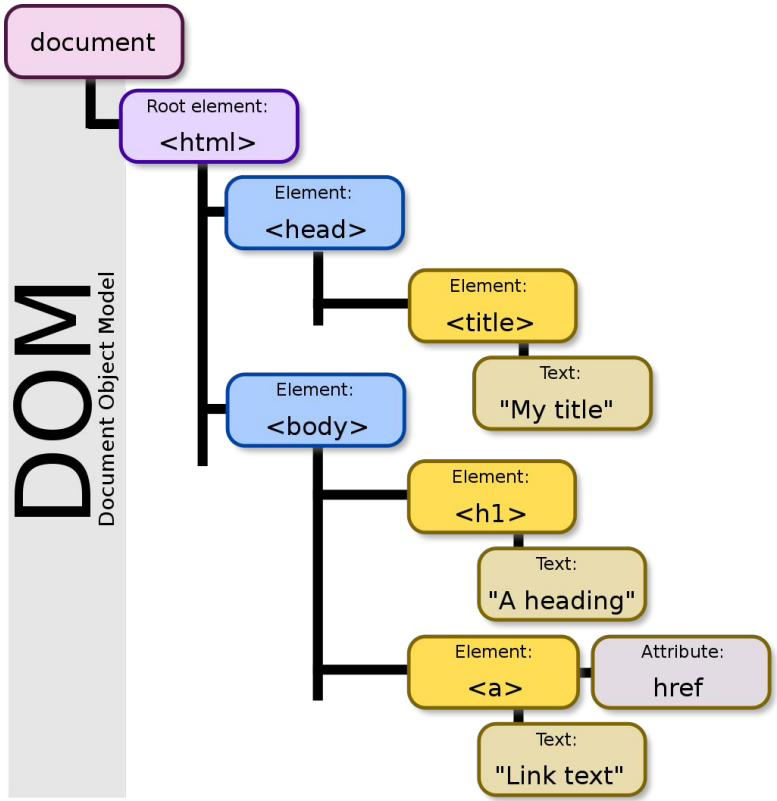
Document Object Model (DOM)

Document Object Model (DOM)

A cross-platform and language-independent interface that treats an XML or HTML document as a tree structure. Each node is an object representing a part of the document.

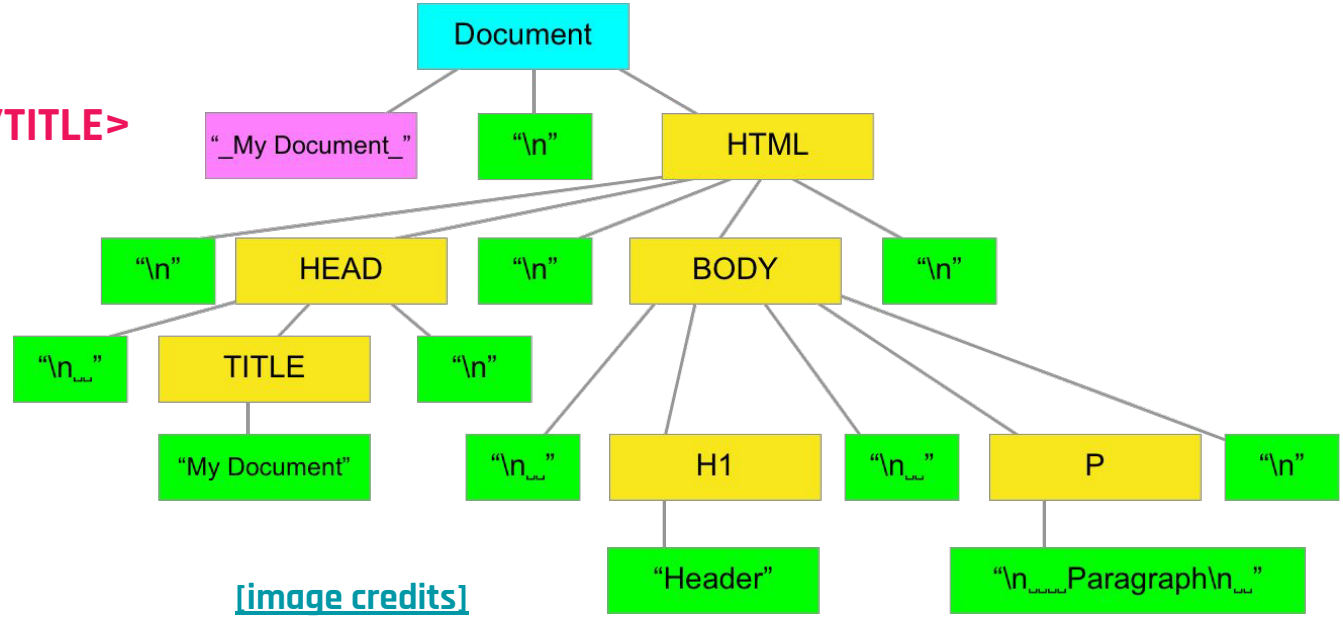
Javascript can thus easily modify the a page by modifying the HTML DOM.

[image credits]



HTML DOM in practice

```
<!-- My document -->
<HTML>
<HEAD>
 <TITLE>My Document</TITLE>
</HEAD>
<BODY>
 <H1>Header</H1>
 <P>Paragraph</P>
 <P>Paragraph</P>
</BODY>
</HTML>
```



[image credits]

HTML DOM in practice (2)

Javascript (client-side):

- Elements can be retrieved with: document.getElementById(id), document.getElementsByTagName(name), document.getElementsByClassName(name). E.g.,:

```
document.getElementById("demo").innerHTML = "Hello World!";
<p id="demo"></p>  ⟶  <p id="demo">Hello World!</p>
```

- Given an element, it can be modified using **element.innerHTML** (element content, see example above), **element.<attribute>** (modify an attribute), **element.style.<property>** (modify a CSS property).

HTML DOM in practice (3)

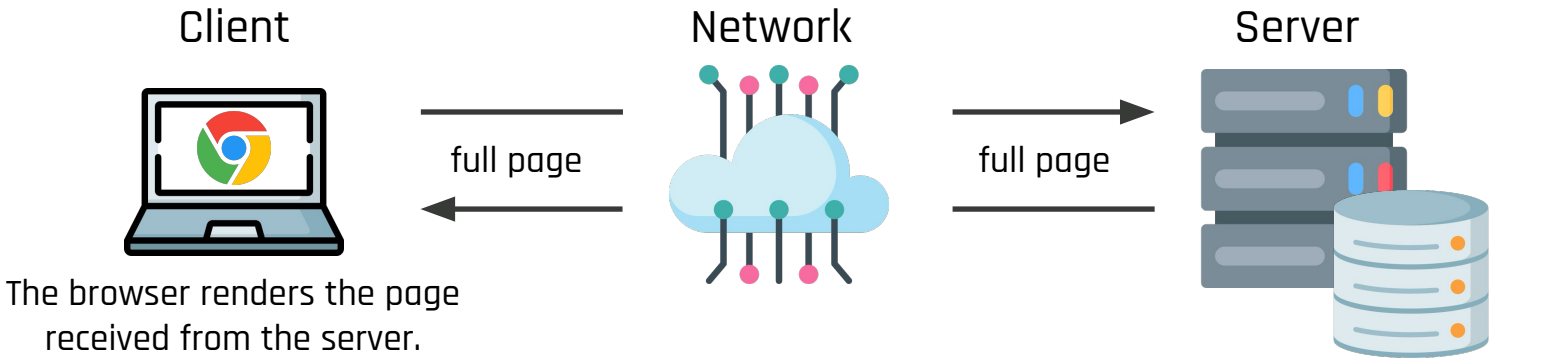
- Given an element, we can modify its subtree with: **element.appendChild(element2), element.replaceChild(old, new), element.removeChild(element2)**

- We can create a new element with **document.createElement(<tagname>)**

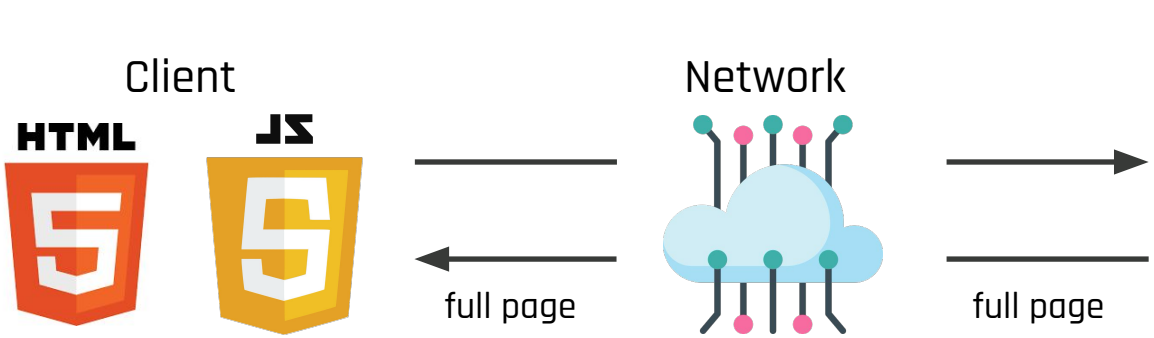
Additional details and example can be found here.

MODERN WEB APPLICATIONS

Web Applications: old but still common approach



Web Applications: old but still common approach (2)



+

Several template endines

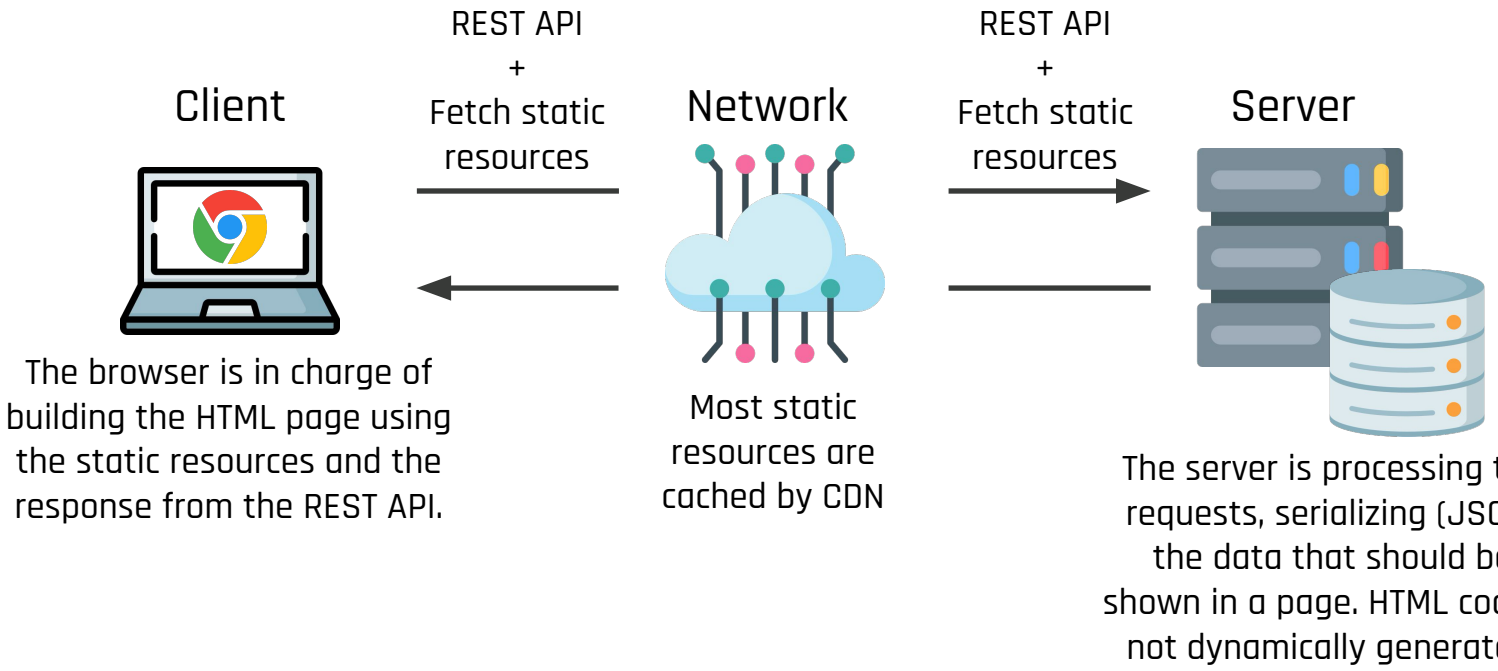
Pros:

- browser is doing little work (it is mainly a “viewer”)
- Simple logic: most things are done by the server

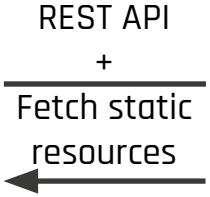
Cons:

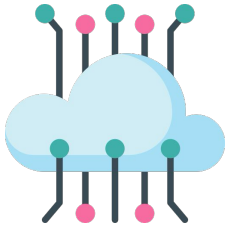
- Hard to scale: large load on the server
- Decoupling: what if want to support a mobile app that has its own way of rendering the content?

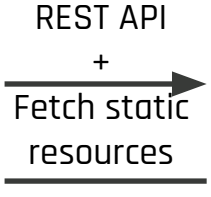
Web Applications: modern approach



Web Applications: modern approach (2)







Pros:

- the server is mainly serving static resources and computing the minimal data required for a response
- Clear separation between frontend and backend, making easier to support other client platforms (e.g., mobile app)

Cons:

- Extremely advanced client frameworks

Web Applications: modern approach (3)

Modern client web frameworks propose the Single-Page Application (SPA) paradigm:

- there is only a single page that is doing all the work. Depending on the URL, the page is built and rendered in different ways.
- when the user clicks something, the page performs a REST request, waits for the response and then renders the new content
- the client framework dynamically modifies the DOM
- there is no need thus reload from scratch the page for each user interaction
- better response time and better user experience
- It is very hard to inspect the code for a human or a bot (e.g., a search engine)

Web Applications: modern approach (4)

HTTP was not designed for a continuous interactions and push notifications.

Modern browsers support **WebSocket**:

```
const socket = new WebSocket('ws://example.com:1234/updates');

// Fired when a connection with a WebSocket is opened,
socket.onopen = function () {
  setInterval(function() {
    if (socket.bufferedAmount == 0)
      socket.send(getUpdateData());
  }, 50);
};

// Fired when data is received through a WebSocket,
socket.onmessage = function(event) {
  handleUpdateData(event.data);
};
```

WEB AUTHENTICATION

HTTP Basic Authentication

HTTP defines with RFC 7617 an HTTP transaction for basic authentication:

- The client request a page, e.g.:

  GET / HTTP/1.1
- The server replies with:

  HTTP/1.1 401 Unauthorized

  WWW-Authenticate: Basic realm=<name-realm>
- The client gets the username/password from the user with, e.g., a popup and sends:

  GET / HTTP/1.1

  Authorization: Basic <BASE64(username:password)>

HTTP Digest Authentication

HTTP defines with RFC 7616 an HTTP transaction for basic authentication:

- The client request a page, e.g.:

  GET / HTTP/1.1

- The server replies with:

  HTTP/1.1 401 Unauthorized

  WWW-Authenticate: Digest realm=<name-realm> nonce=<nonce>

  opaque=<opaque> algorithm=<algorithm> qop=auth

algorithm is a cryptographically secure hash function (default: MD5)

HTTP Digest Authentication (2)

HTTP defines with RFC 7616 an HTTP transaction for basic authentication:

- The client gets the username/password from the user with, e.g., a popup and sends:

  GET / HTTP/1.1

  Authorization: Digest realm=<name-realm> nonce=<nonce> user=<user>

  opaque=<opaque> response=<digest> nc=<counter> cnonce=<cnonce>

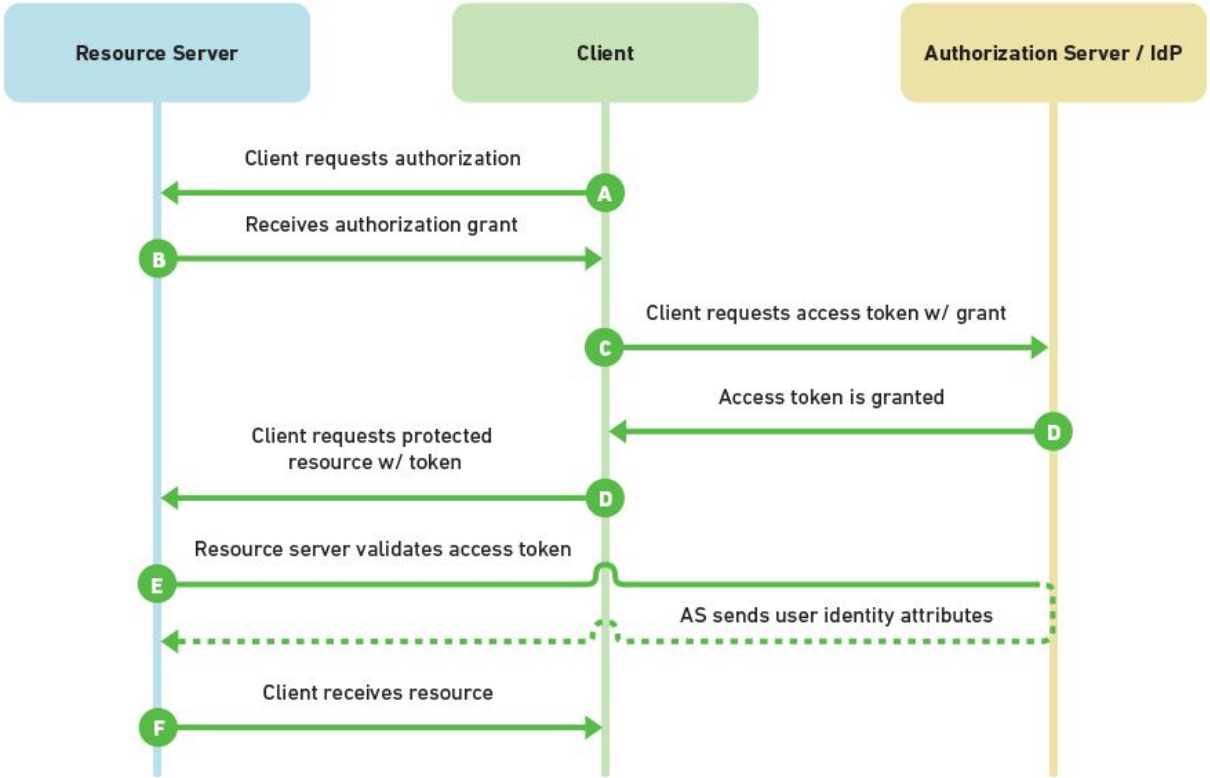
  where

  digest = HASH(HASH(<username>:<name-realm>:<passwd>):

  <nonce>:<counter>:<cnonce>:auth:HASH(GET:/))

OAuth and Single Sign On (SSO)

- The user has an account on a service provider (Google, Facebook, Linkedin, etc.), also called Identity Provider (IdP)
- The client wants to access a protected resource on a server.
- The server generates a grant code for the client
- The client interacts with IdP, obtaining an access token, which is sent to the server
- The server validates by interacting with IdP, getting also user attributes.



BURP SUITE

Burp Suite by PortSwigger

- Platform for performing security testing of web applications
- Written in Java. Proprietary and closed source :(
  - Luckily, there is a (free) community edition for Linux, Windows, and Mac OS X **[DOWNLOAD]**
- Several functionalities:
  - HTTP(S) Interceptor
  - HTTP(S) repeater
  - Request comparer

NOTE: THERE IS NO MAGIC BEHIND THIS TOOL. HENCE YOU CAN DO THE SAME THINGS WITH OTHER (100% OPEN SOURCE) TOOLS. HOWEVER, THIS TOOL CAN BE VALUABLE AT THE BEGINNING WHEN YOU ARE JUST STARTING LEARNING ABOUT THE WEB.

Another valuable resource from the same company...

Web Security Academy

Boost your career

The Web Security Academy is a strong step toward a career in cybersecurity.

Flexible learning

Learn anywhere, anytime, with free interactive labs and progress-tracking.

Learn from experts

Produced by a world-class team - led by the author of The Web Application Hacker's Handbook.

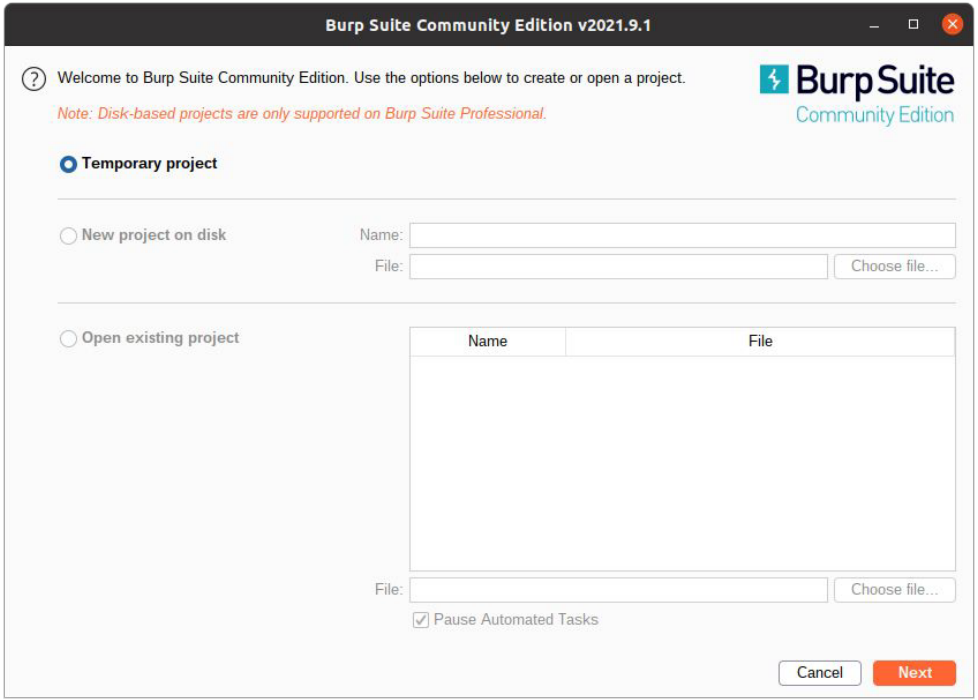
It nicely covers the topics that we will see in the upcoming lectures. The labs may help you prepare for the CTF.

Reference: https://portswigger.net/web-security

Tutorial - Burp Suite (1)

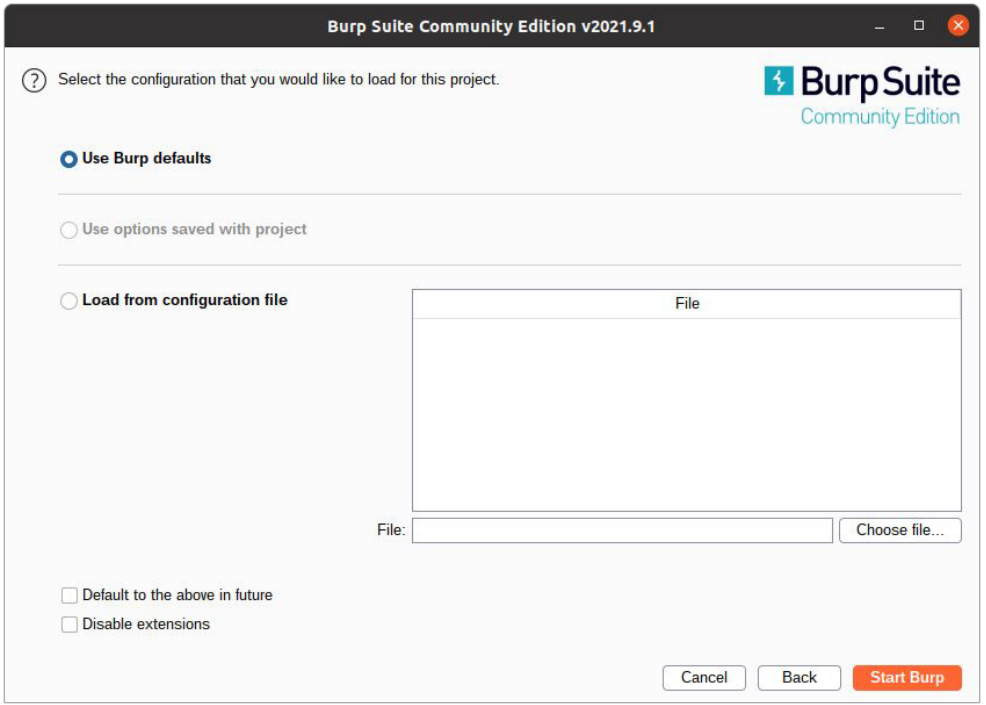
Tutorial - Burp Suite (2)

A temporary project is fine for our goals.



Tutorial - Burp Suite (3)

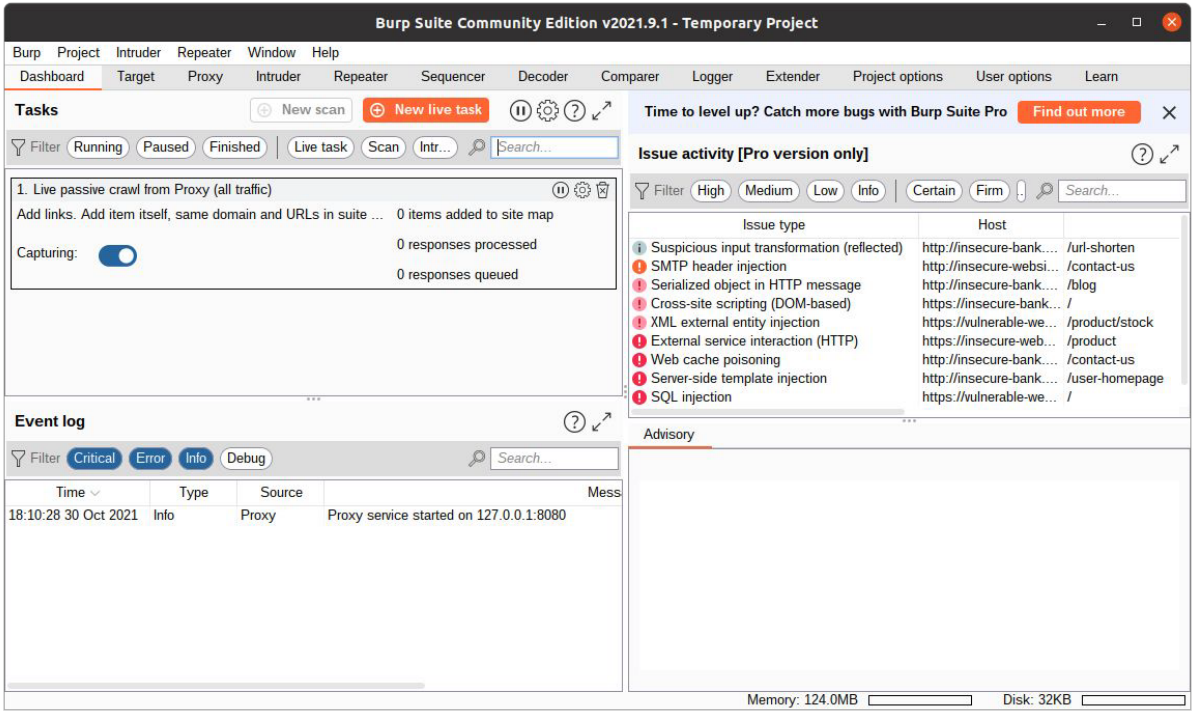
Default settings are fine for our goals.



Tutorial - Burp Suite (4)

Dashboard...

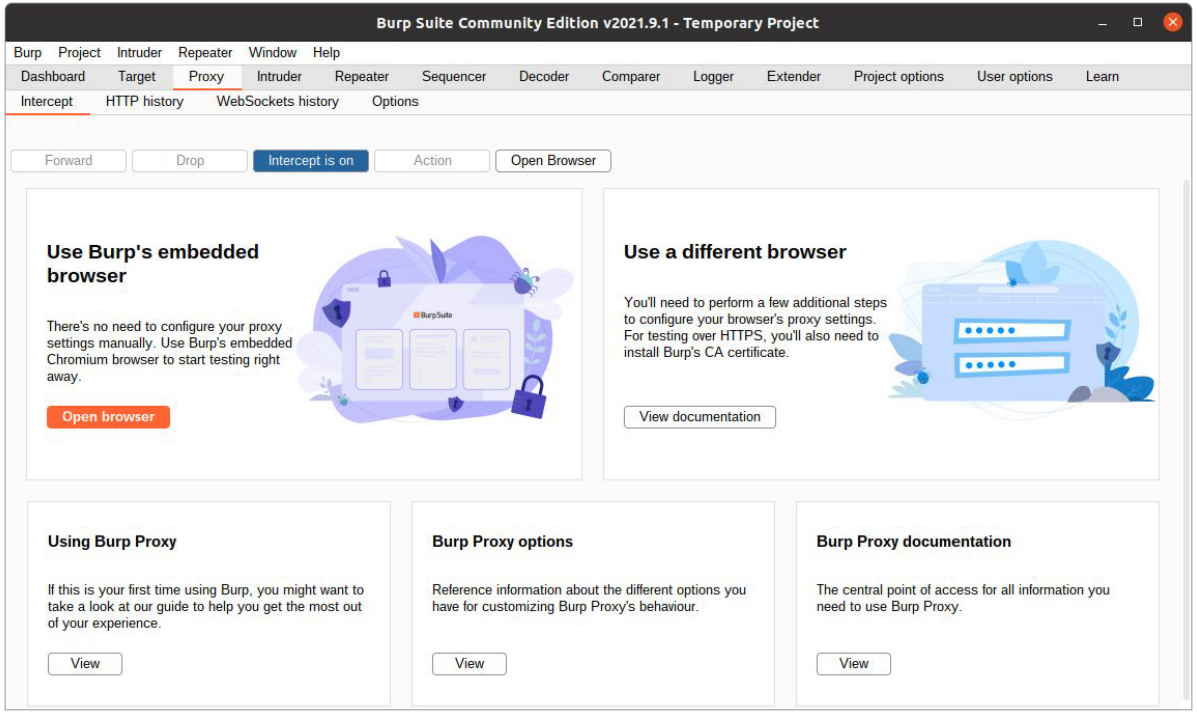
Use the tabs to switch to specific functionalities



Tutorial - Burp Suite (5)

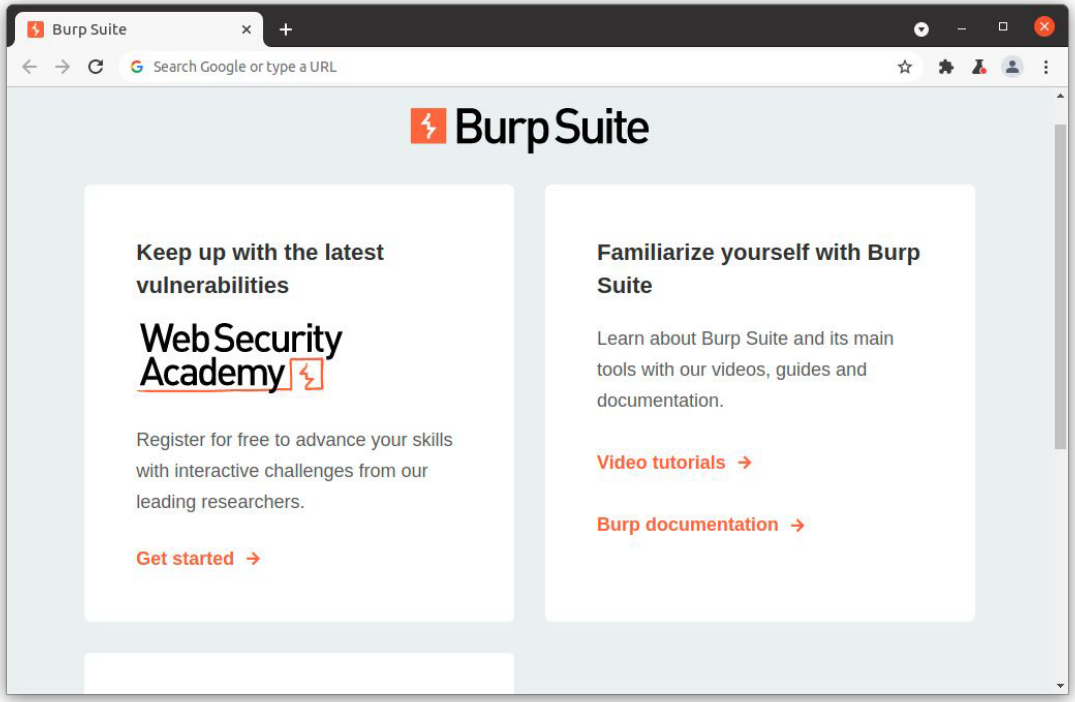
Tab: **Proxy > Intercept**

Use *Open Browser* to launch the embedded browser



Tutorial - Burp Suite (6)

The embedded browser is based on Chromium. This a (clean) browser: use it for the challenges.

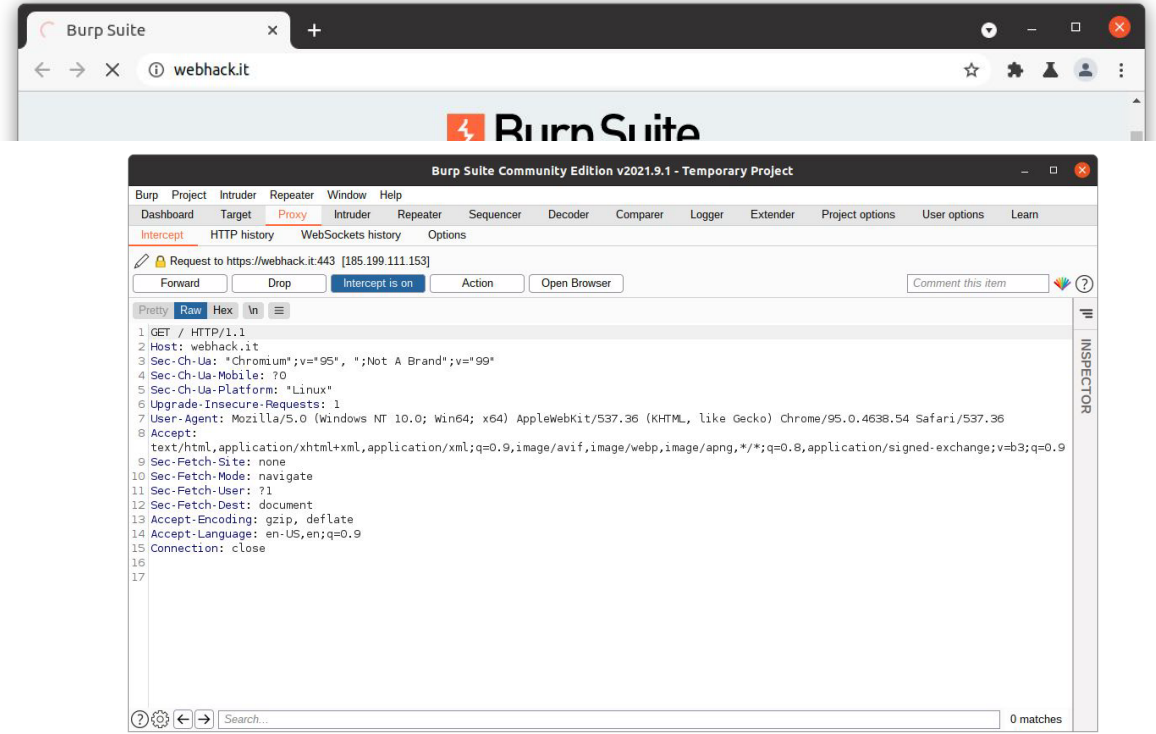


Tutorial - Burp Suite (7)

When doing a HTTP request, Burp will intercept it and wait for your instruction:

- forward
- drop

You can edit the request before forwarding it.



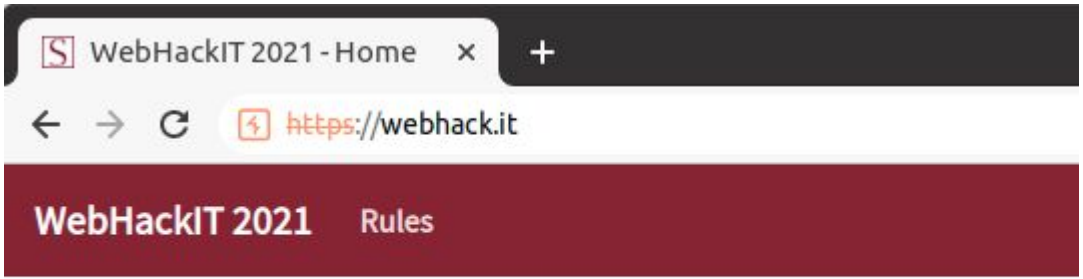
Tutorial - Burp Suite (8)

Intercepting each request gives you a lot of control but it is a mess when a site requires a lot of requests... e.g., for images, css, js, etc.

Hence, we can disable this function: click *intercept is on*. Now, we can browse without stopping each request.

Tutorial - Burp Suite (9)

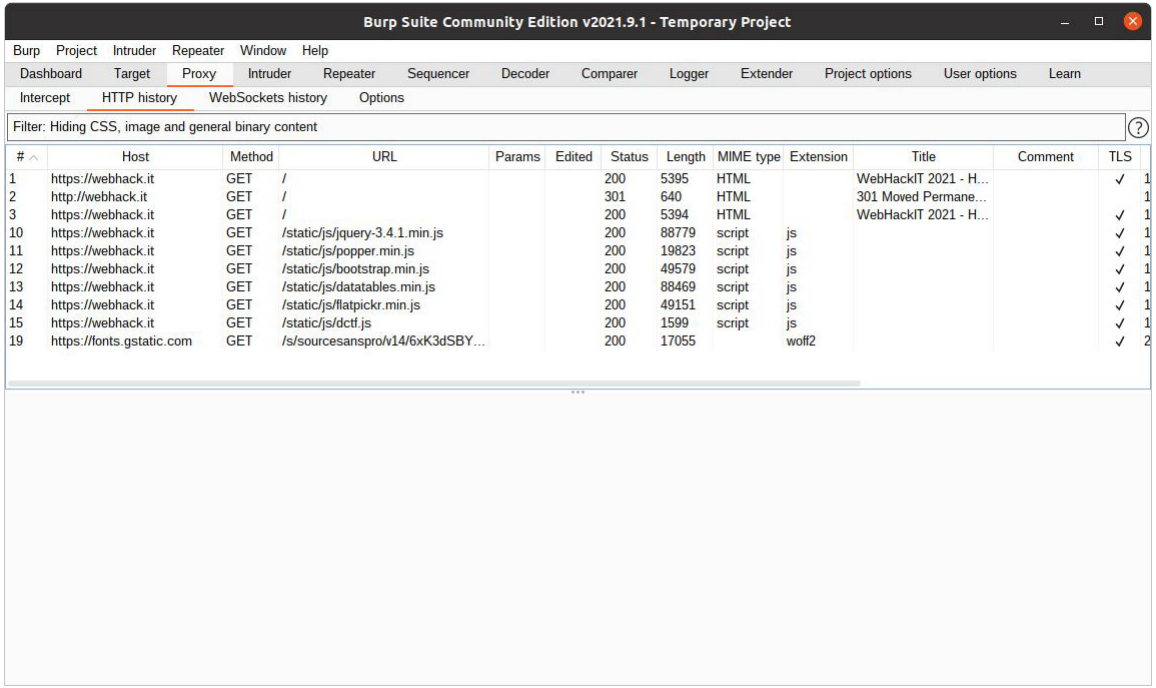
Burp Suite is messing up with the certificates! This is done to correctly intercept our HTTPS traffic.



Tutorial - Burp Suite (9)

Tab: **Proxy > HTTP History**

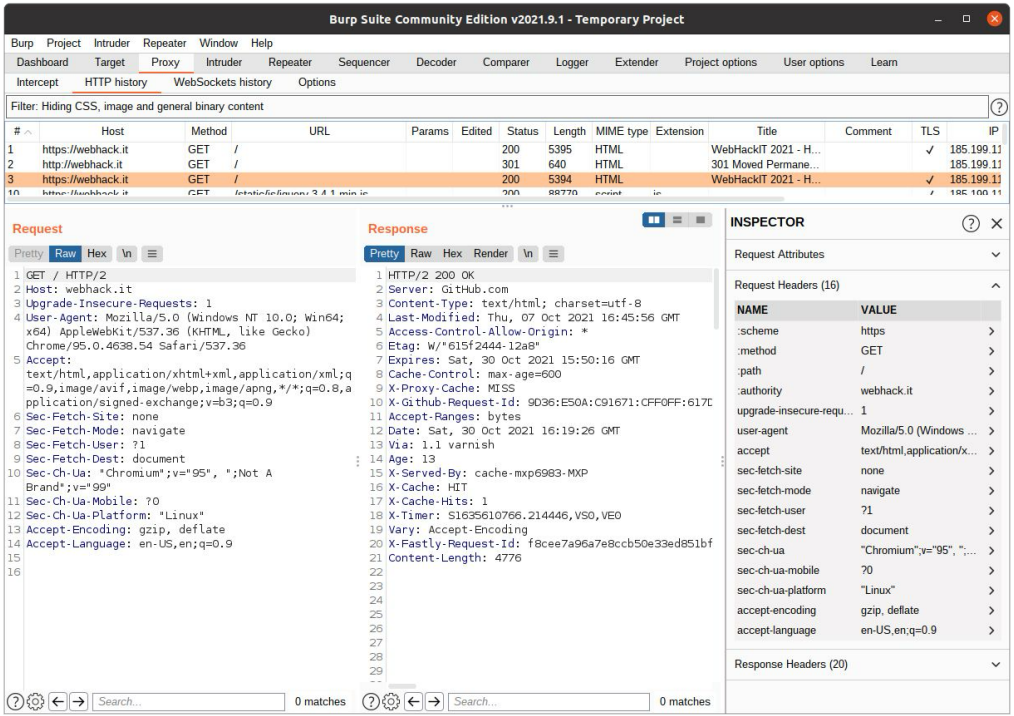
We get a history of all network requests. We can easily inspect them...



Tutorial - Burp Suite (10)

Tab: **Proxy > HTTP History**

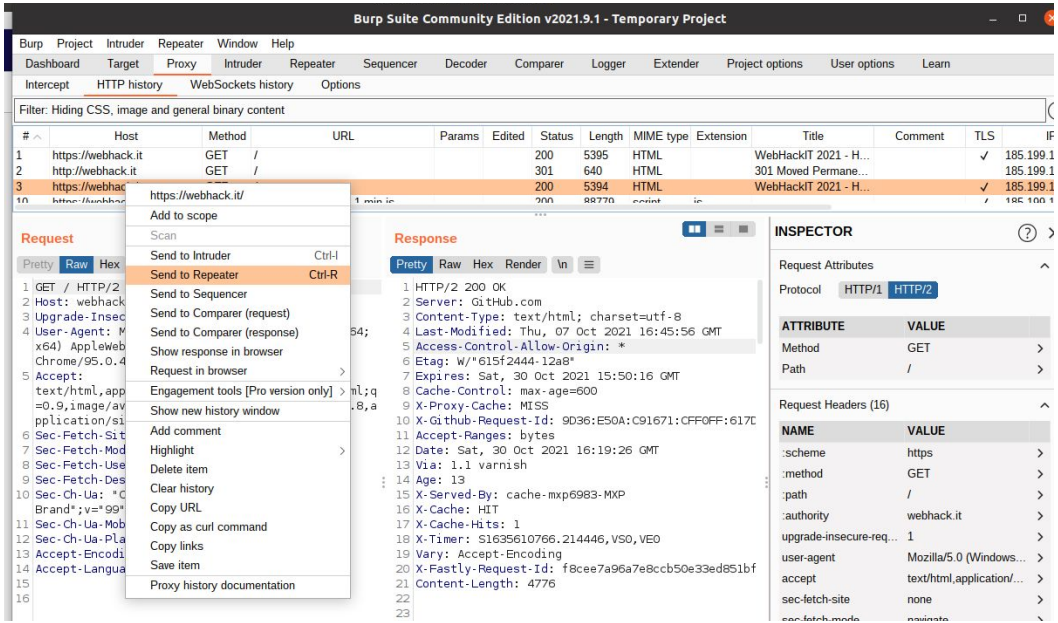
We can see the request and the response.



Tutorial - Burp Suite (11)

Tab: **Proxy > HTTP History**

If we want to repeat a request, then can use *Send to Repeater*



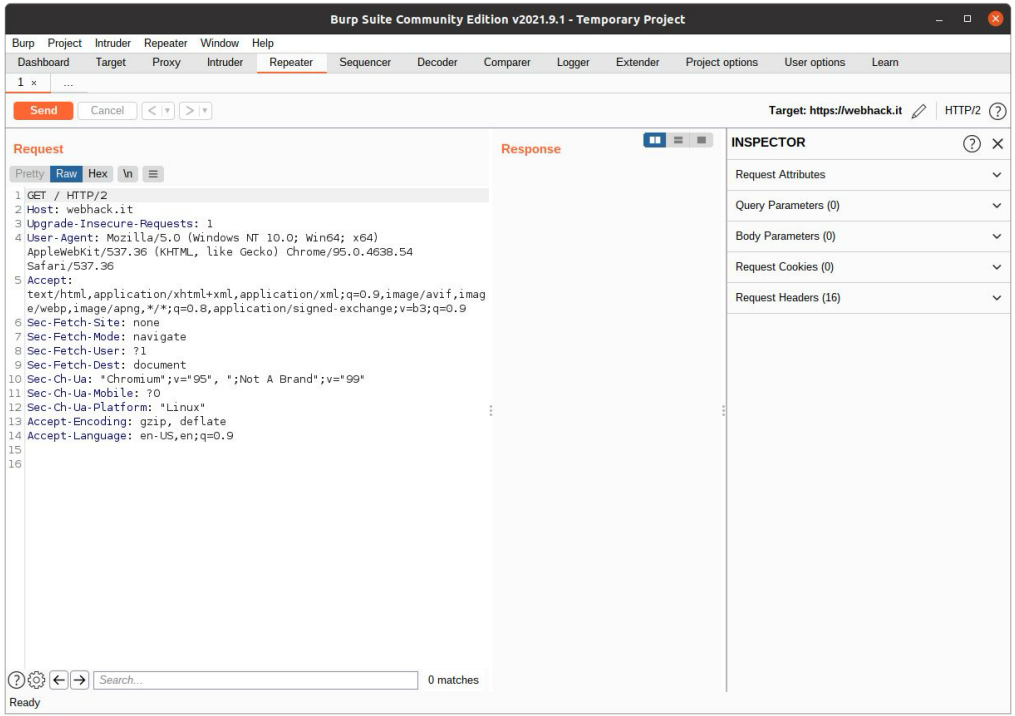
Tutorial - Burp Suite (12)

Tab: **Repeater**

Now we can modify the request (headers and/or the body). E.g.:

- change the URL
- change Accept-Language
- change User-Agent

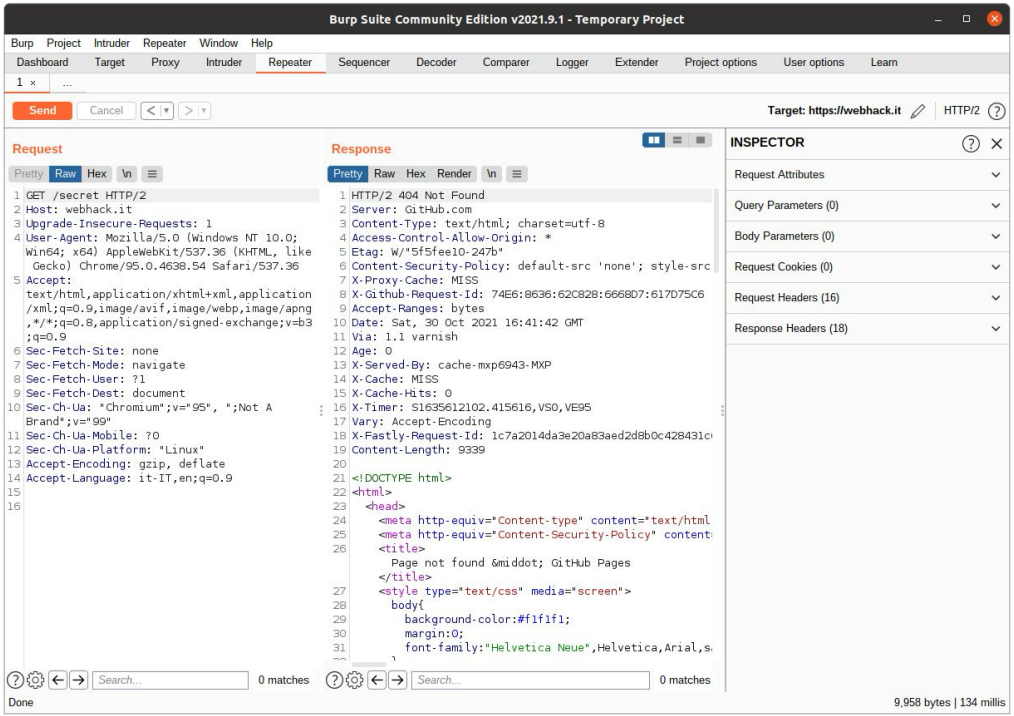
After we can *Send* it.



Tutorial - Burp Suite (13)

Tab: **Repeater**

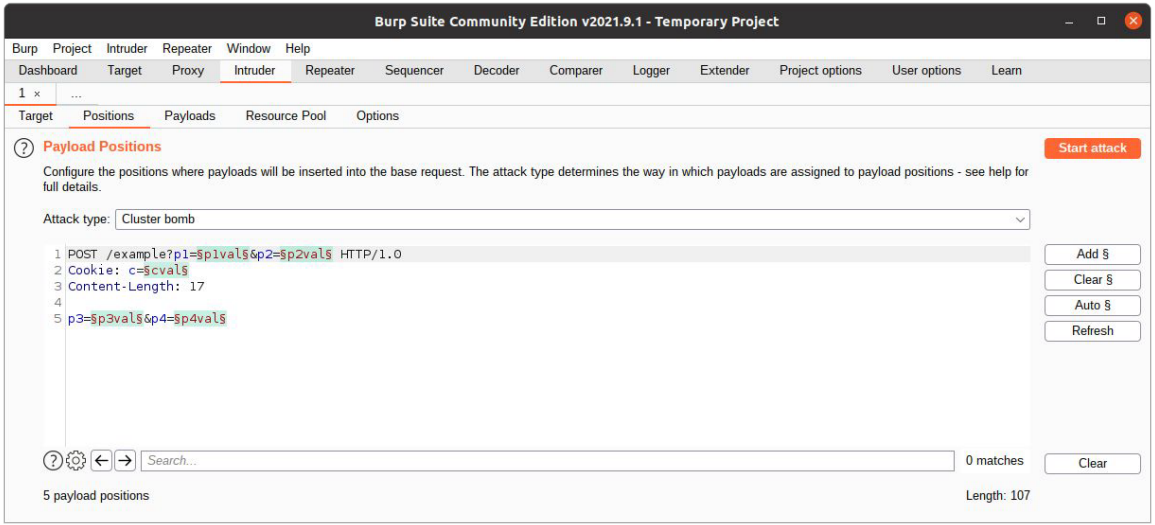
We get back the response. We can use *Render* to view the rendered page.



Tutorial - Burp Suite (14)

Tab: **Intruder**

This can be used to perform brute-force attacks. For instance, you can test the value of a GET/POST/COOKIE picking values from a list (e.g., a dictionary).



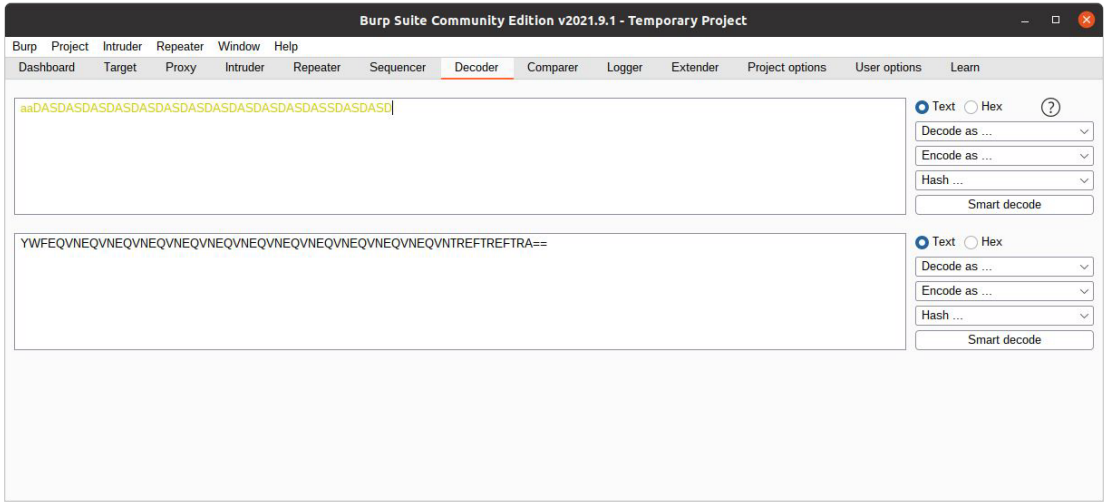
THERE IS NO NEED TO USE THIS FUNCTIONALITY IN OUR CTF

Tutorial - Burp Suite (14)

Tab: **Decoder**

Quick way of {de,en}coding data.

E.g., In the example, we can encode on the fly a plain text in base64



Training challenge #13

URL: https://training13.webhack.it

NOTE: **THE CHALLENGE IS LIVE!**
TRY IT TO LEARN!

Description:

Getting into a system is not always easy... unless... the page leaks crucial information!

Byte Information Exchange

username

password

Enter

WebHackIT

Analysis

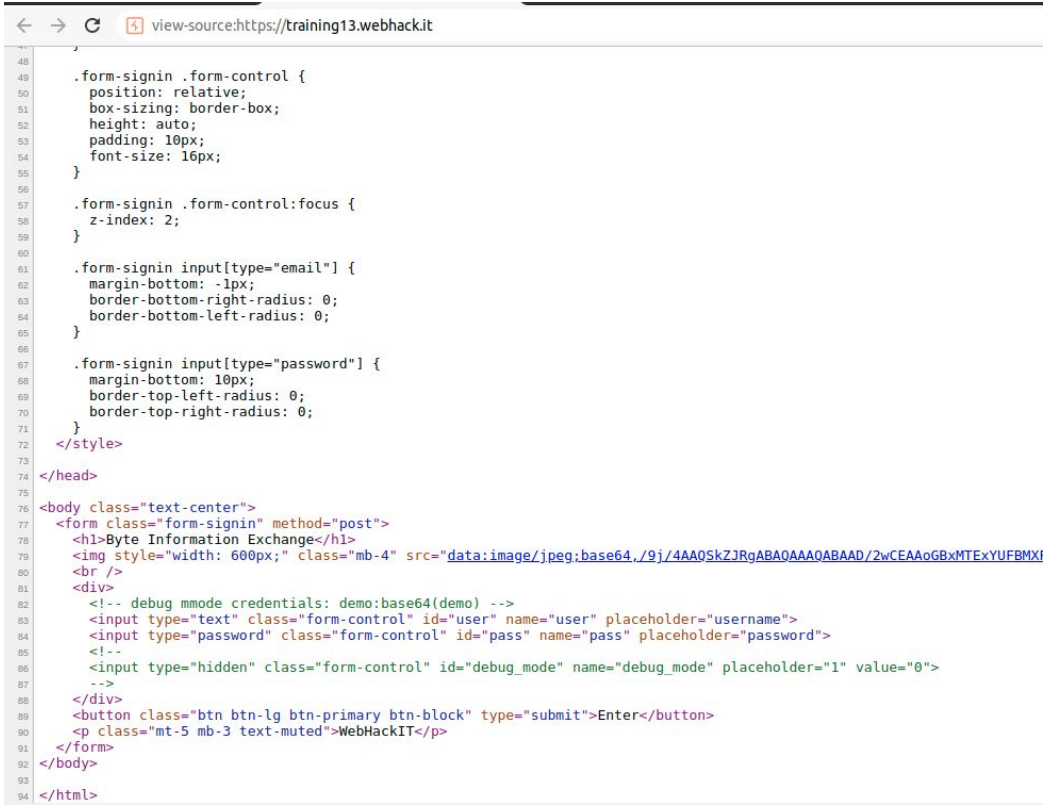
- It is a web application that asks username/password
- The description is hinting that the page is leaking some crucial information

....let's try to carefully check the page!

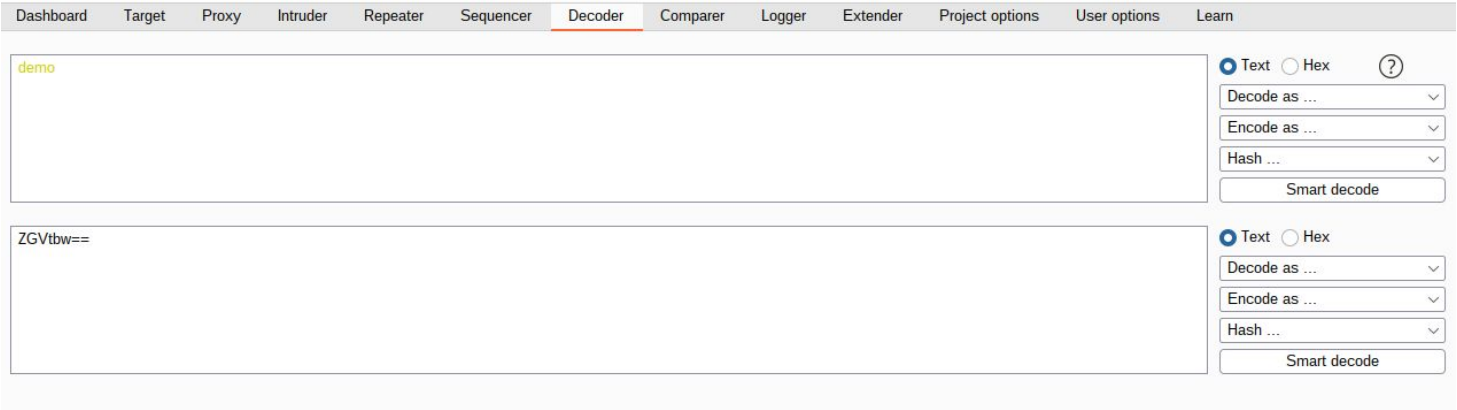
Solution (1)

We see two comments:

- the first one is suggesting a username/password
- the second one is exposing a hidden POST key/value



Solution (2)



Using Decoder in Burp Suite, we can quickly get base64("demo")

Solution (3)

Using the Repeater in Burp Suite, we can forge a new request, using the computed password and adding the additional POST key/value