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...

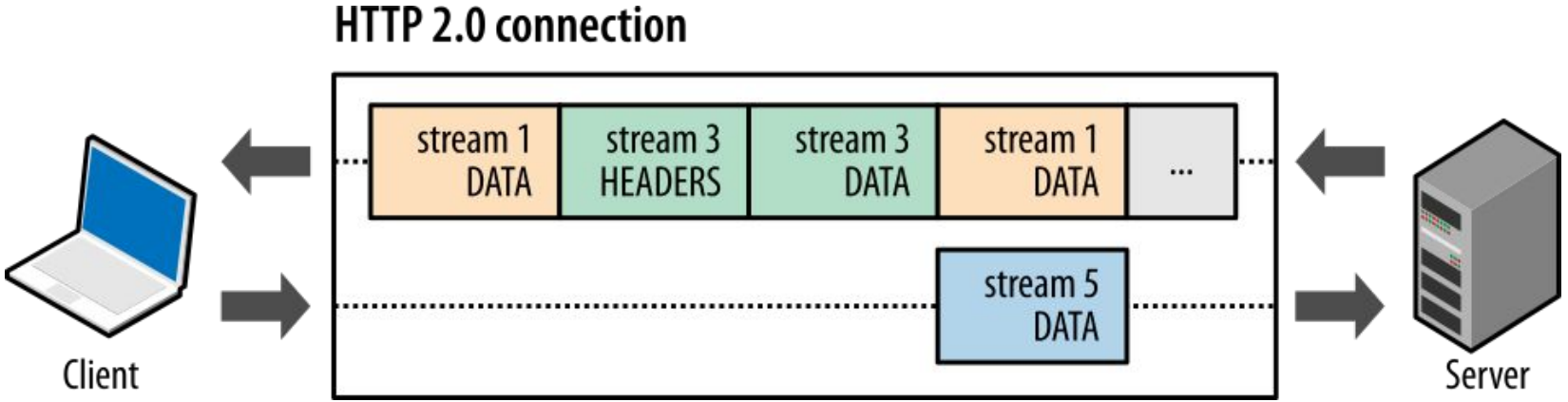
HTTP2: some key ideas

- 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

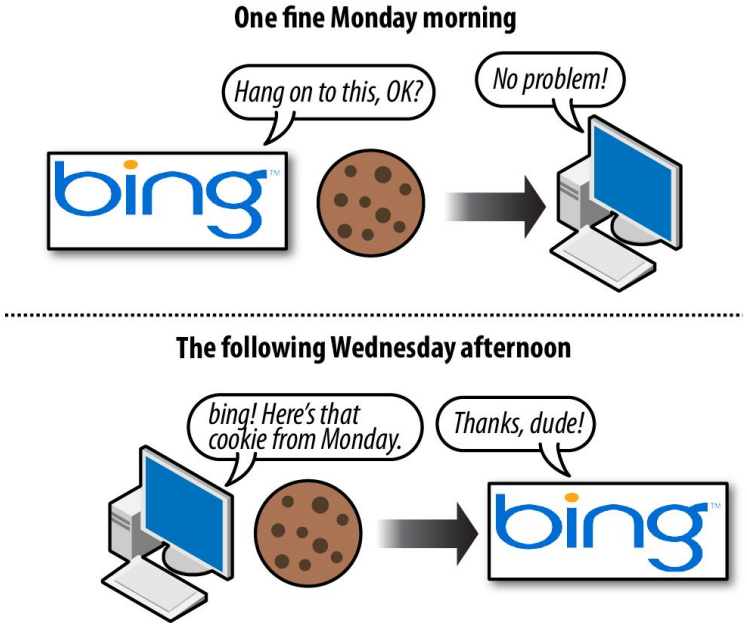
- HTTP/2 addresses **head-of-line blocking** (HOL) through request multiplexing, which eliminates HOL blocking at the application layer, but **HOL still exists at the transport TCP layer**!
- Third revision of HTTP, still in draft. Backward compatible: same request methods, status codes, and message 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

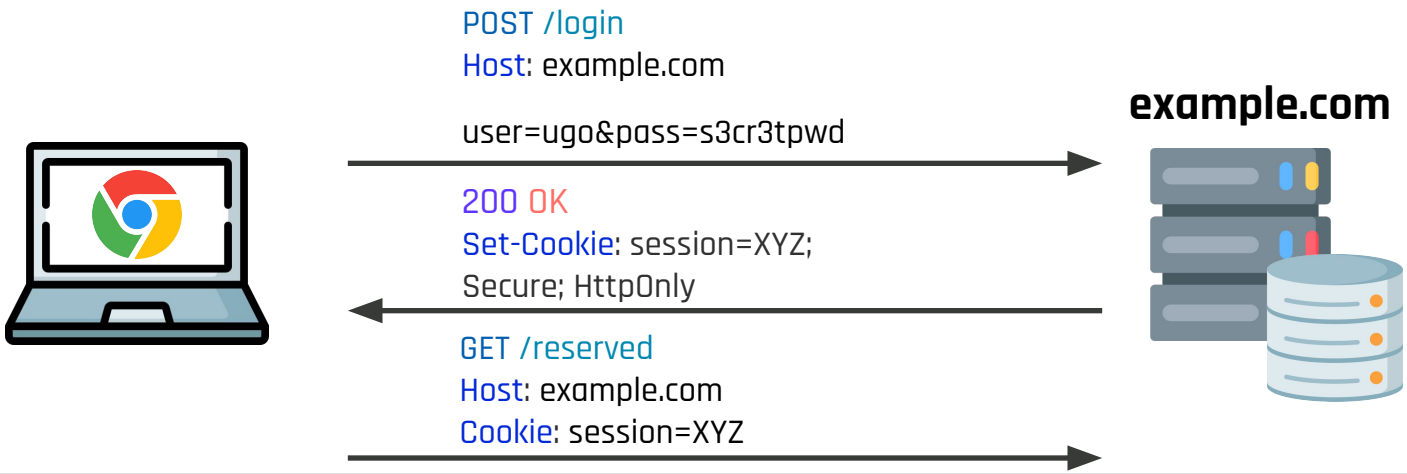
- HTTP(S) is a stateless protocol
  - Requests are independent from each other
  - What if user wants to stay logged in?

- 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.

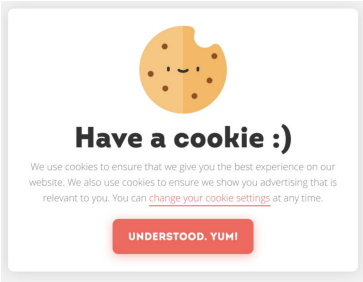
Cookies in practice (3)

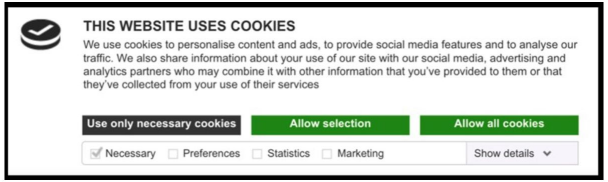
Handling cookies in PHP (server side):

```
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

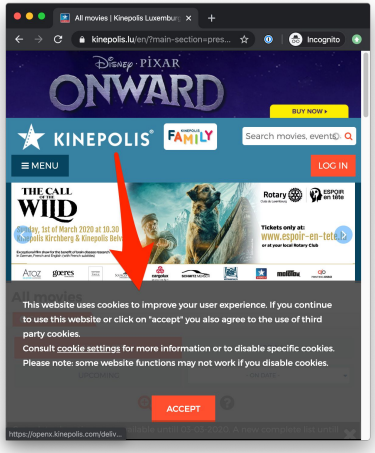


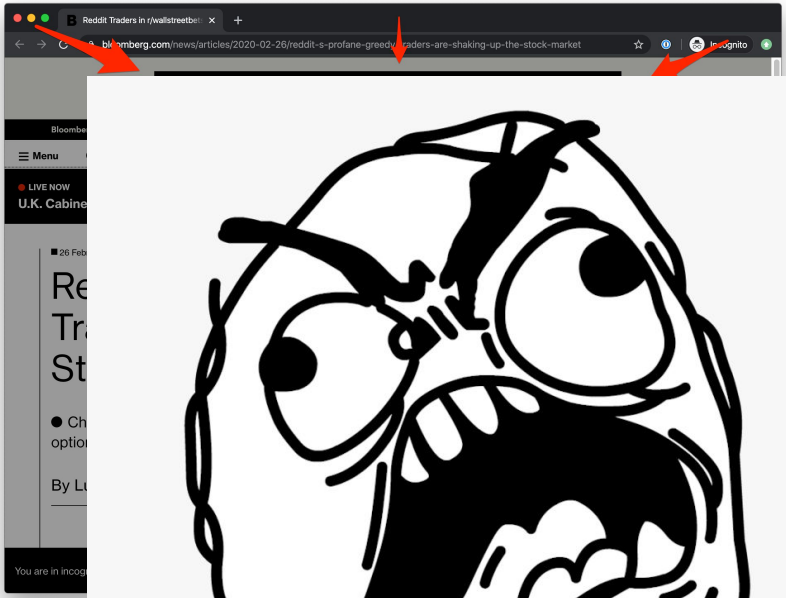


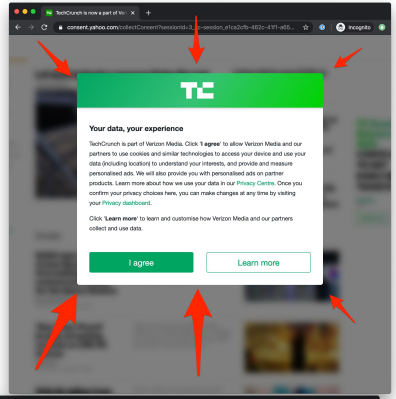
Third-party cookies

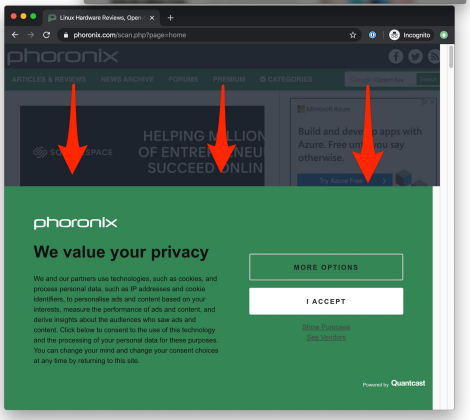
- a page can host contents coming from other web servers
- cookies that are sent by these servers are named **third-party cookies**
- there are organizations operating in the advertisement that use third-party cookies for tracking users across different sites, allowing ads consistent to user profile
- This may be a huge privacy concern (we talk about this later in the course)
- 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**.

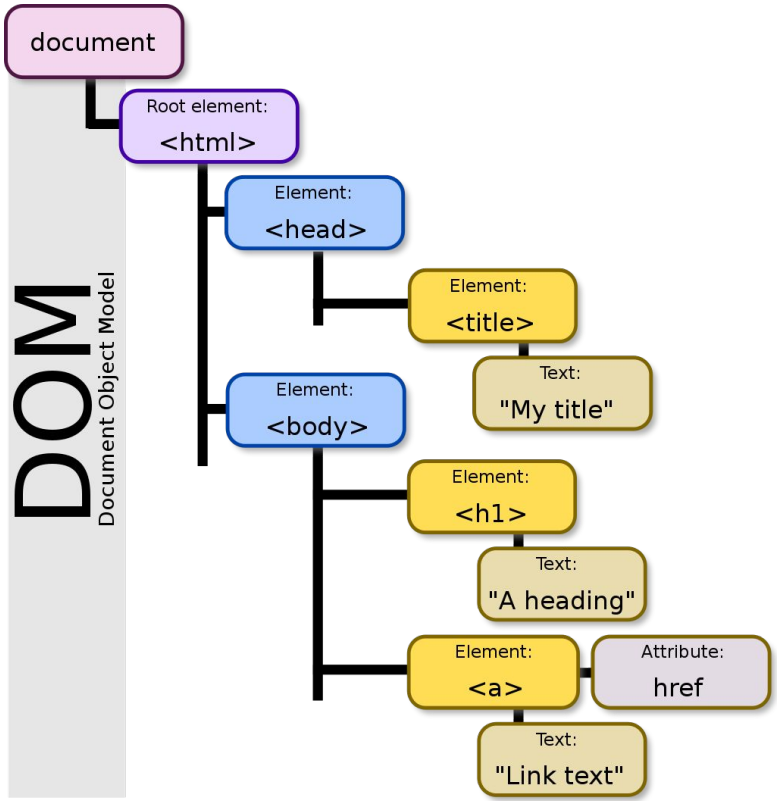
Document Object Model (DOM)

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.

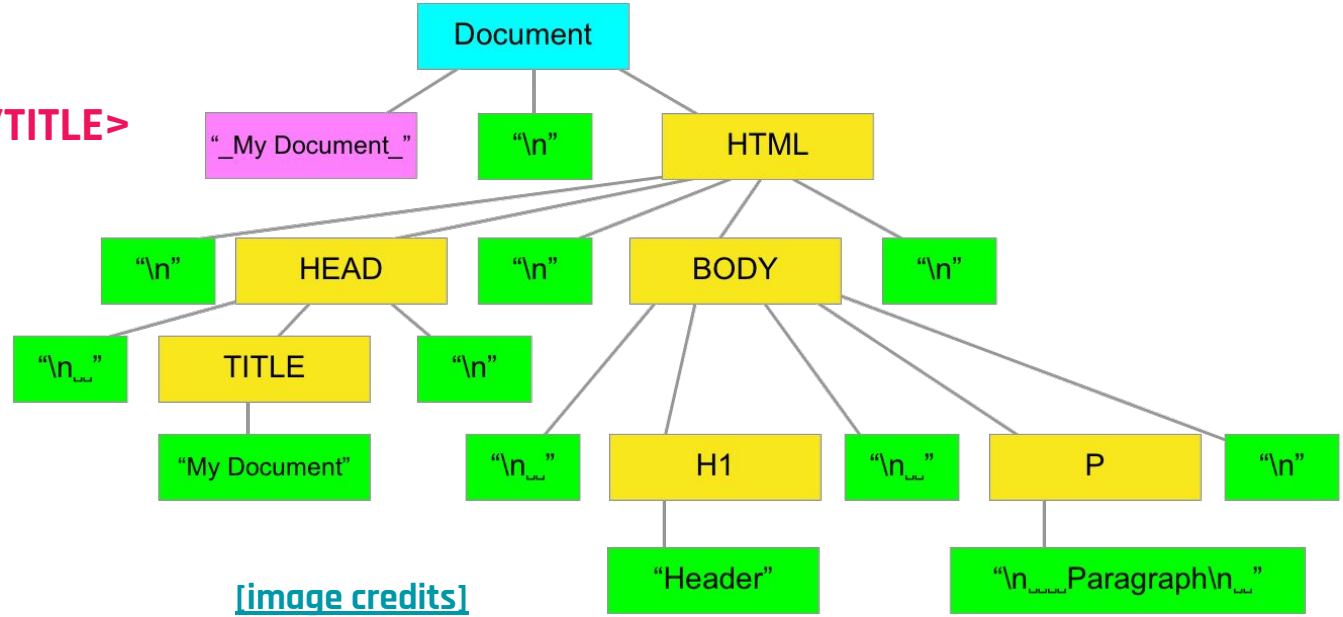
Javascript can thus easily modify the a page by modifying the HTML DOM.

[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!";
```

** ** → ** Hello World! **

- 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)

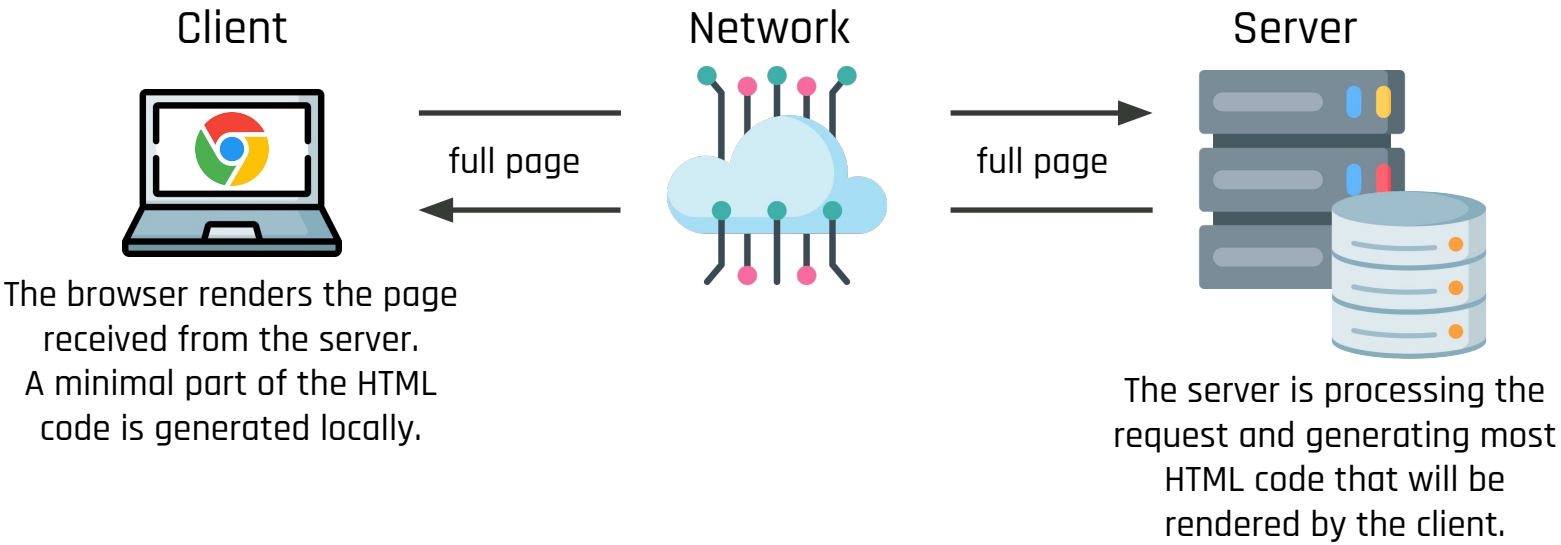
- Given an element, we can modify its subtree with: **element.appendChild(element2), element.replaceChild(old, new), element.removeChild(element2)**

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

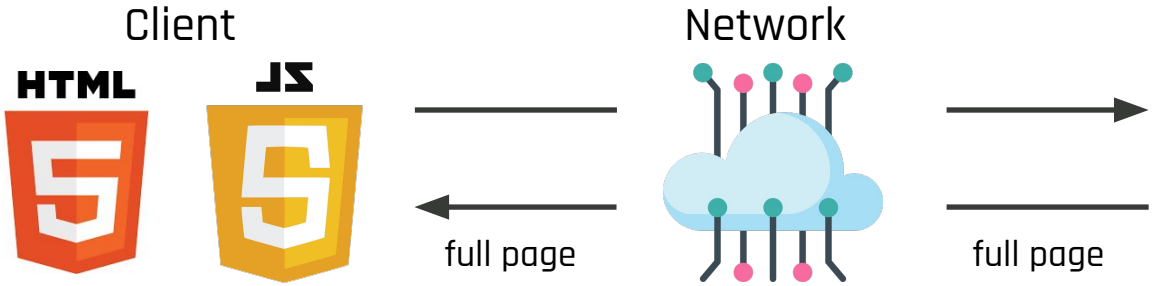
Additional details and example can be found here.

MODERN WEB APPLICATIONS

Web Applications: old but still common approach



Web Applications: old but still common approach (2)



+

Several template endines

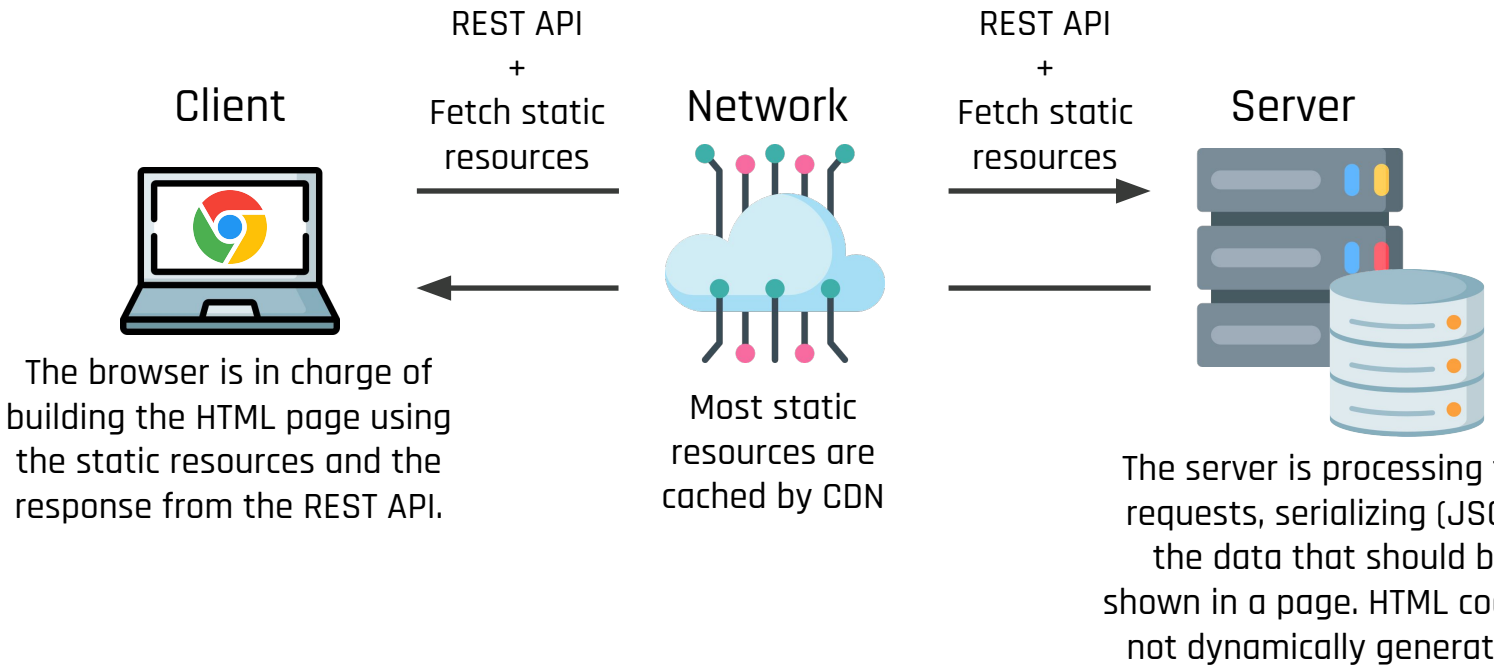
Pros:

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

Cons:

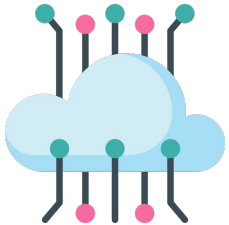
- Hard to scale: large load on the server
- Decoupling: what if want to support a mobile app that has its own way of rendering the content?

Web Applications: modern approach



Web Applications: modern approach (2)

REST API
+
Fetch static resources



REST API
+
Fetch static resources

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>

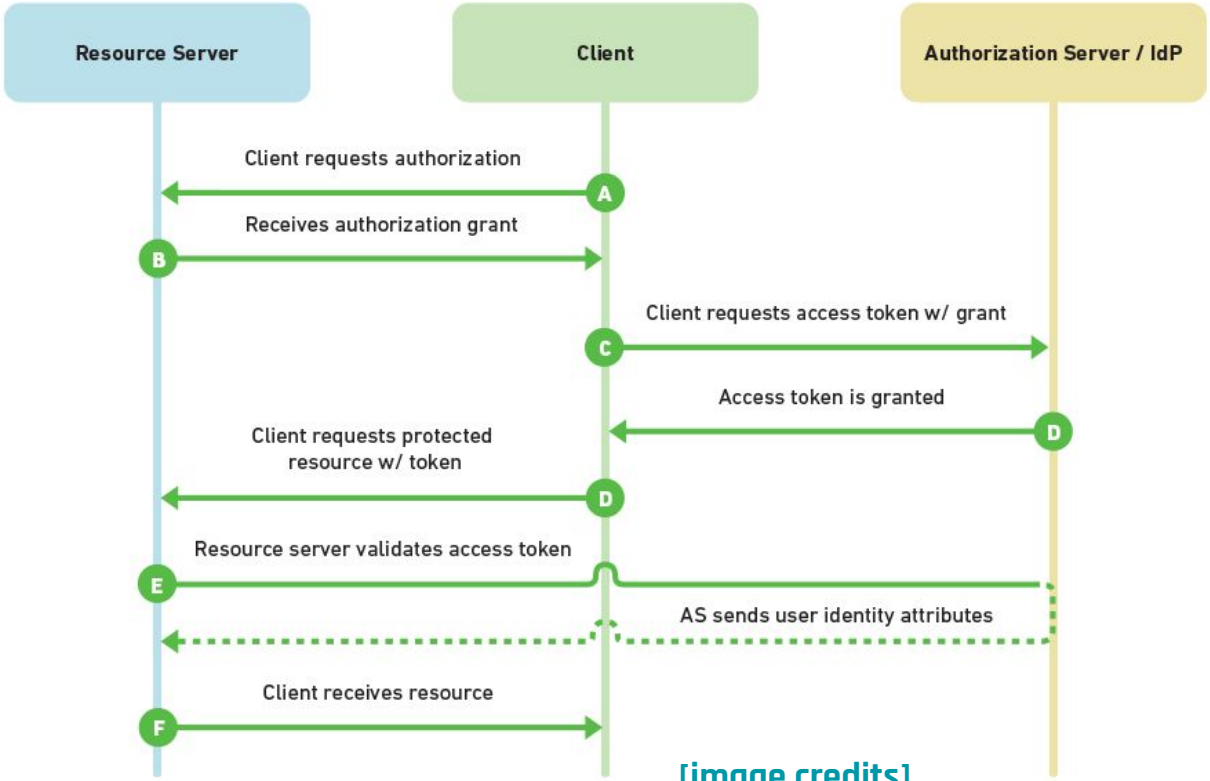
  where

  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.



[image credits]