#1. TANANYAG: HTML 2025.02.20.

Fullstack = Frontend + Backend

Frontend: HTML + CSS + JS
HTML -> "váz" struktúrális leírása
CSS -> kinézet
JS -> kliens oldali dinamizmus

Frontend és Backend közötti kommunikáció pl interneten (HTTP) keresztül történik. A Backend-en van egy API réteg is. A kettőt el lehet szeparálni.
API first approach (API első megközelítés) Fejlesztés kezdetekor a Frontend és Backend fejlesztő megbeszélik hogy milyen API hívások fognak kelleni.
*pl: getAllStudents(x)
* '-> Student
* '->name, age,...
Ha az API előre el van tervezve, akkor egyszerre el tudnak kezdeni dolgozni.

HTTP -> *[OSI modell alkalmazás réteg]*
 |->POST
 |->GET
 |->PUT
 '->DEL...

HTML -> HyperText Markup Language
HTTP -> *HyperText* Tranfer Protocol
  '-> Sir Tim Berners Lee nevéhez fűződnek ezek

VS Code-ot használunk
VS CODE Live Sever Extension otthonra

"Default preset" az alap kinézete a html-nek

html dokumentumnak fix szabványokat kell követnie: "! + tab"

A <head> _*metaadatokat* tartalmaz
  '-> adatról szóló adat

Napjainkban:
HTML 5-ös verzió (fontossabb frissítések: canvas, video, zene)
CSS 3-as verzió
JS ?

alapvető visszaadás: főoldalt index.html-nek nevezzük el és akkor ez lesz az alapértelmezett főoldal.

Napjainkban az iframe használata nem feltétlen ajánlott/elfogadott, biztonsági kockázatok miatt

#2. TANANYAG: HTML + CSS 2025.02.27

**Leíró nyelvek (markup language)
  |-> HTML
  |-> Tex
  |-> XML
  |-> JSON
  |-> `markdown`
  '-> YAML

markdown vagy Tex típusú dokumentációkat lehet verziókezelni, nem úgy mint egy word dokumentumot

---

#HTML alapok folytatás

svg vs raster képek. Svg képek vektorgrafikusak mig a raster képek nem. Azok pixelesek ha belezoomolsz.
A szöveges dolgok tekinthetők svg-nek mert azok nem pixelesednek ha belezoomolsz.

HTML-ben is van már svg:

```
<svg width="100" height="100">
  <circle cx="50" cy="50" r="40" stroke="green" stroke-width="4" fill="yellow" />
</svg>
```

Ez pl akkor jó ha használunk bootstrap ikonokat.

Van egy JS megoldás -> Használják e az emberek? -> Ha igen, akkor HTML natív támogatás.

Mi form-os verziókat nem nagyon fogunk használni. A form azért jó mert detektálja hogy összetartoznak mert egy formon belül vannak.
Form-nál a gombnyomás hatására (input type="submit") lefut az "action"

Mi input type="submit" helyett button-t fogunk használni form nélkül

ha megjleníteni szeretnék kacsacsőrt akkor speciális karaktereket kell hasznánlni: < >

##CSS alapok:

CSS -> Cascading Style Sheets `language`

CSS azért kell mert a HTML-nek van egy Defual Presetje és a CSS-sel felül tudjuk írni a HTML stílusát/kinézetét.

(érdekesség -> CSS tailwind)

1.###inline CSS

'-> html tag-en belül megadott css -> ez nem annyire preferált megoldás

2.###internal CSS

'-> html <head> részben van írva

CSS 3 fontos újdonsága:
- gradients -> színátmenetek,
- shadow,
- meadia query, transzfromációk -> különböző dolgok mozgatása forgatása

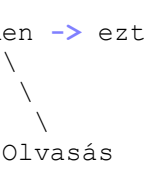
ameddig nem voltak ilyenek addig trükközni kellett.

** **PURE CSS -> kizárólag CSS**** az az hogy ne kelljen ügyeskedni pl photoshoppolt "letöltés" feliratú képekkel, meg átmenetekkel
JQUERY -> függvénykönyvtár. EZ akkor volt jól, amikor még a CSS nem volt ilyen szinten mint most.

`Class` és `Id` között vannak fundamentális különbségek, de ****nincs működésbeli különbség**
De elvárás lehet hogy id csak 1szer forduljon elő per oldal
Nem is nagyon használunk id-kat, hanem inkább csak classokat

Sok kicsi osztály van.
Keretrendszereknél az a fontos hogy ismerjük majd az osztályokat.

SOLID elvek: I - Intefrace szegregáció:



```
          /  |  \
         /   |   \
        A    B    C
```

3.### external CSS -> külső css fájl behivatkozása
'-> head-en belül: <link rel="stylesheet" href="../style.css">

#3. TANANYAG: CSS keretrendszerek (Bootstrap) 2025.03.06

HTML elemek két kategóriába sorolhatók: inline és block-level.

1. A blokk-level elem mindig új sorban kezdődik, és a böngészők automatikusan hozzáadnak egy margót az elem elé és után.
A blokkszintű elem mindig a teljes rendelkezésre álló szélességet foglalja el (balra és jobbra nyúlik, ameddig csak tud).
Két gyakran használt blokkelem: <p> és .

2. Egy inline elem nem kezdődik új sorban.
Egy inline elem csak annyi szélességet foglal el, amennyi szükséges.
Inline elem: <a>

Megesik hogy vannak elcsúszások a megjelenítésben, ezért szokás `reset css`-t használni. Amihez hasonló a `bootstrap reboot`.

Hasznos oldal gombok készítéséhez: https://www.bestcssbuttongenerator.com/

- Tipográfia

10rem -> root element
10em -> element (szülő)
10%
10px

- Webergonómia
"mobiel usability" https://miro.medium.com/v2/resize:fit:640/format:webp/1*j1lF_H9bdPN-g78oPKqQtQ.jpeg

ide tartozik ez a két kifejezés is:
-ux -> user experience
-ui ->user interface
összefügnek ezek de mást jelentenek
ui a nézet/felület
ux pedig az hogy ezen a felületen mennyire jó érzés létezni, mozogni. "mennyire jó ezeket használni"

`reszponzivitás`: ***optimális megjelenés biztosítása különböző kijelzőkön

van mobile-first design és desktop-first design

***css media query
https://www.w3schools.com/css/css_rwd_mediaqueries.asp
https://www.w3schools.com/css/css3_mediaqueries_ex.asp
a css media query-nél az adott stílus csak akkor lép érvénybe ha a megadott feltétel igaz.
Például ha a kijelző mérete kisebb lesz mint 600px, akkor átrendezi a kinézetet

fejlesztésnél hasznos dolgok:

!important:
p {

```
  background-color: red !important;
}
```
ez például felülirja a p tag-eken lévő stílusokat. Esetleg ha az hozzá van adva egy class-hoz vagy van egy id-ja, akkor is ez fog érvényesülni

```
*{
    border: 2px solid red;
    background-color: beige;
}
```
Az összes elem-et kiválasztja

*pagination -> lapozás
*infinite scrolling / doomscrolling -> ilyen pl a facebook is
https://addyosmani.com/assets/images/infinite-scroll.png

#oldal struktúrák kialakítás
|-> table (pl:totalcommander https://www.ghisler.com/) (ez rossz megoldás)
'-> div
|-> flexbox (1 dimenziós) https://miro.medium.com/v2/resize:fit:880/1*I6QO_eTwocQE3X77KFfUXw.png
'-> grid (2 dimenziós)

html tree
https://www.openbookproject.net/tutorials/getdown/css/images/lesson4/HTMLDOMTree.png

#flexbox
```
div class="container"
.container{
    display: flex
    flex-wrap: wrap
}
```

#grid
```
div class="grid-container"
p class="grid-item"
```

nem annyira egyszerű ezért van grid generátor: https://cssgrid-generator.netlify.app/

#block-level vs inline

Block level elfoglalja a teljes képernyő szélességét, így több ilyen, egymás alatt helyezkednek el. Ilyen pl a `<p>` tag

Inline pedig egymás mellett fognak elhelyezkedni, ilyen pl a `<span>`

Sok olyan dolgot vettünk most át amikkel sokat lehet szívni.
Ezért fogunk Bootstrap keretrendszert használni.

CDN -> content delivery network
A CDN földrajzilag elosztott szerverek csoportja, amelyek felgyorsítják a webtartalom kézbesítését azáltal, hogy közelebb hozzák a felhasználók helyéhez.
https://www.cloudns.net/blog/wp-content/uploads/2023/04/CDN.png

#5. TANANYAG: JavaScript 2025.03.20

***Történelem
www -> 1990-es évek eleje

"böngésző háború"
js -> 1995-ben készült el Brendan Eich nevéhez kötődik
a "js egy szar nyelv" ezt a kitalálója mondta
Sok probléma -> erre megoldás a TypeScript

hogy kapcsolódik ide a `Java`?
Azért van benne a nevében mert 95-ben Java volt az egyik legnépszerűbb programozási nyelv
Eredetileg LiveScript lett volna a neve. Üzleti okokból lett JS a neve.

Nyelvi jellemzők
- high-level -> sok más nyelvben is megtalálható ez, magyarul: absztarkciós szintű nyelv => nem kell foglalkozni a cpu-val és a memóriával
        '-> nem kell foglalkozni sem a processzor sem a memória szintű dolgokkal (pl a c# és php is ilyen)

        ez jónak tűnik de nem feltétlen az

            ^
          __|__
          __|__
          __|__   absztarkciós szintek
            |
         gépi kód

         minnél több absztarkciós szint annál nagyobb a hibalehetőség.

    ellentéte a low level

- garbage collected (kapcsolódik a high-level-hez)
        '-> háttérmehanizmus ami a felszabadítást elvégzi; a referenciával nem ellátott területeket felszabadítja

- interpreted (vagy just in time compiled *JIT*)
        '-> interpretált / értelmezett (js, php)
        sorról sorra futnak le a kódok. ez olyan hogy amikor elér egy sort már akkor el is kezdi végrehajtani
        hátrány hogy nem igazán tudjuk kioptimalizálni a kódot.

    ellentéte a compiled
                    '-> fordított (c#)
                    itt a cs állományból létrejön egy köztes állomány:
                    .cs -> compiled -> .exe
                    itt figyelembe veszi a teljes kódot

    a *JIT* megpróbálja a kettőt ötvözni

- multi-paradigm | paradigma -> nézőpont, megközelítés, értelmezési nézet/szemüveg
    '-> procedurális (szekvenciális, sorrend alapján halad végig)
    '-> OOP
    '-> impreatív (C, C#, java, php) -> azt írom le hogy HOGYAN szeretném hogy valami megtörténjen
    '-> dekleratív (SQL, Linq)   -> itt azt írom le, hogy MIT szeretnék hogy megtörténjen
                                                                '-> (pl. szeretném a legnagyobb elemet)

- prototype-based (object oriented)
    ősosztály: `Prototype`
        olyan mint pl C#-ban:
        class kutya : **: Obejct**
        {

        }

- first class functions | első osztályú függvények jellemzik
         '-> függvények használhatók mint változók

            apró példa:
            function Alma(){...}
            function X (param) {...}
            x(alma)

ez olyan mint c#-ban a delegáltak

- dynamic | dynamically typed language -> dinamikusan típusos nyelv

    (erősen vagy gyengén típusos is egy módja a nyelvek csoportosításánka)
    erősen típusos: c#
    gyengén típusos: JS, php ($alma=...)

    c#: string a = "alma"
    js: let a = "alma"
        a = false
        a = 1002
    dinamikusan típusos nyelvben egy változó változtathatja a típusát is
    ez nem azt jelenti hogy a változónak nincs típusa

    Előnye:
        - Gyors prototipizálás
        - Leegyszerűsíti a kódolási folyamatot: gyorsabb kódolás és flexibilitás.
    Hátránya:
        - A típushoz kapcsolódó hibákat a rendszer csak a végrehajtás során észleli, ami váratlan hibákat okozhat, és bonyolítja a hibakeresést.
        - könnyebb véletlenül hibásan viselkedő kódot írni, amelye gyengeségeket a támadók kihasználva váratlan műveleteket hajthatnak végre, például rosszindulatú kódokat juttathatnak be vagy megkerülhetik a biztonsági ellenőrzéseket.

- single threaded (non-blocking event loop)
    '-> a JS egy szálas programozási nyelv. Aszinkron de egy szálas.

**** Használata:**
A <body> legaljára kell beszúrni. Lehet akár többet is.
<script src="app.js"></script>

****JS-ben a pontosvesszők elhanyagolhatóak**
használható ' és " is van aki azt mondja a ' jobb

****három mód változó deklarálásához:**
    - var   -> ezt soha ne használjuk!
    - let   -> változó értéke később módosulhat
    - const -> konstans

c#-ban van null
js-ben is van null csak kiegészül még undefined állapottal

var
 '-> TDZ -> Temporal Dead Zone
 '-> hoisting -> https://www.w3schools.com/js/js_hoisting.asp

****három alap változó van: number, string, boolean. nincs double vagy float**

****strict mód használata**
strict mód az ECMAScript 5-ös verzióban jött be
    - strict módban például nem használhatunk nem deklarált változókat.

ECMAScript -> ez egy standard | szabványoknak a leírása
    '-> különböző elvárások hogy egy programozási nyelv milyen feltételeknek feleljen meg

összehasonlításnál:
=== -> típus + érték összehasonlítás
== -> csak érték összehasonlítás

nem egyenlő: !==

a c#-hoz hasonló foreach itt a forof, de a foreach is működik

```
objektum létrehozása:
let obj = {
    name: 'lali',
    age: 23,
    salary: 2323
}
console.log(obj);

forin -> objektumnak a tulajdonságait kiírja
ez olyan mint c#-ban a reflexió


metódus osztályhoz van kötve, a függvény pedig nem
function alma() {
    console.log("alma");
}
ha osztályba írnánk akkor a function szót ki kell hagyni!


Callback függvények -> A callback egy függvény, amelyet argumentumként adnak át egy másik függvénynek.









#6. TANANYAG JavaScript 2025.03.27

ami jó a js-ben: nagyon gyors benne a prototipizálás

/** tab -> ez a jsdoc comment, érdemes használni pl függévények előtt

/**
 * Két szám összeadása...
 *
 * @param {number|string} param1 - Első param
 * @param {number} param2 - Második param
 * @returns {number}
 */
function add(param1, param2){
    return param1 + param2
}

## Mai téma: DOM + események

`DOM`
   '-> Document Object Model
   ez egy köztes interfész
   azt teszit lehetővé hogy a js kódból a html-t módosítjuk
    Js kód és a html között van
DOM -> **WEB API-k része! NEM része a JS nyelvnek


*rákerestünk: WEB apis : https://developer.mozilla.org/en-US/docs/Web/API

API -> Application Programming Interface
Api működését tudni kell meg hogy mire jó mit csinál


*rákerestünk: windows os api list :
https://learn.microsoft.com/en-us/windows/win32/apiindex/windows-api-list

a javascript az ecmascript alapján jött létre


***Page load timeline
https://docs.newrelic.com/images/browser_diagram_page-load-timeline.webp
A network-öt nem tudjuk változtatni, mert nem tőlünk függ, de a web applictaion
```

time-ot tudjuk, attól függően hogy mennyire jól írtuk meg pl a backend-et

ezután jön a DOM processing majd utána a Page rendering
a DOM egy fa adatstruktúra. o.b betöltésekor DOM fa jön létre
a böngésző küld egy jelet a DOM-nak a DOM továbbítja a JS-nek, a JS mahinál a DOM-mal ami alapján a bögésző betölti az oldalt

React -> virtual dom-ot használni
Andular is használ egy ilyet -> change defection
más a nevük de a lényegük ugyanz

**Console-ban:

document.getElementByID
        .getElementsByName
    ha mindent jól csinálunk ID-ból csak egy van

document.getElementsByTagName("h1")

***újabb megoldás:

document.querySelector

classnál: document.querySelector(".important")

id-nál: document.querySelector("#heading")

#BOM -> https://www.w3schools.com/js/js_window.asp
Browser
Object
Model
https://files.codingninjas.in/article_images/javascript-bom-0-1637751814.webp

* timeout:
```
console.log("start");
setTimeout(() => {
    console.log("hello");
}, 5000);
```

ez ugyanaz mint ez:
```
console.log("start");
window.setTimeout(log, 5000)
function log() {console.log("hello");}
```

ez a BOM-hoz szorosan kapcsolódik

CLICK ESEMÉNYEK

fogaskerék -> group similar... kipipálása

```
    <button onclick="log()">Hello</button>
    <p onclick="log()">Button</p>
```

nem csak gombra lehet rátenni az onlcick-et

oldal forrás -> 'nyers' HTML ami a szervertől jön

inspect element -> DOM

#ZH kérdés lehet:
**eltérhet e az oldal forrása html mint amiben az inspect element-ben lévő kódtól

data[] ---> render() ---> DOM <---> HTML
[forrás] ------------------------->

```
classList.add("cssOsztaly")
classList.remove("cssOsztaly")
classList.toggle("cssOsztaly")



***2 fontos fogalom az események kapcsán (ezek nem js specifikusak)
1) buborékolás - event bubbling -> rákattintasz egy elemre akkor a szülő elemeire is rákattintasz
2) delagálás - event delegation -> ez az előzőnek a fordítottja

https://dmitripavlutin.com/static/9a8fc772a94452ca819295094c99b1a9/3e7da/javascript-event-propagation-5.png

gituhb-on fel van töltve ehhez példa: érdemes átnézni
https://github.com/siposm/bprof-frontend-weekly/blob/master/js-02/event-bubbling.html
https://github.com/siposm/bprof-frontend-weekly/blob/master/js-02/event-delegation.html


#fetch hívás
szerverről ha le akarunk tölteni adatot, akkor azt ezzel fogjuk tudni megcsinálni

FRONTEND-> |internet| ->       [API]->BACKEND-> DataBase
FRONTEND    <- render()<-[JSON] <-  |internet| <-       [API]<-BACKEND<- DataBase

hívás lehetőség:
1)  fetch
2)  XMLHTTPRequest (XHR)

`Axios` valójában a háttérben egy fetch hívás, de wrapperekkel elfedte a fetch hívást

Fetch API a WEB APIK egy eleme
https://github.com/siposm/bprof-frontend-weekly/blob/master/js-02/fetch.html
https://api.siposm.hu/


házi: múlt féléves backend api meghívása és feldolgozása
ha nem működik: CORS probléma lehet az ok, ekkor keressünk rá hogy `ASP CORS enable all`

fetchről tudni érdemes:

a JS single threaded : egy szálas
egyszerre egy dologra tud fúkuszálni
mikor jönnek a külnböző utasítások akkor azok ezen az egy szálon futnak végig:
|--[]--[]--[]------->t

emelett a JS aszinkron formábal működik
                '-> sokan azt hiszik hogy ez azt jelenti hogy "párhuzamos"
                akkor beszélünk valamiről hogy párhuzamos ha nem csak 1 szálunk van hanem több

                sokan azt hiszik hogy az aszinkronitás miatt a js is párhuzamos. De ez nem így van.


a JS:
- single threaded
- asynchronous
- non-blocking event loop

párhuzamosság: 1 vagy több dedikált szálon/magon futnak feladatok párhuzamosan
aszinkronitás: feladatok egymástól nem függve tudnak futni

tehát ha elindul a program akkor van mondjuk pár consolelog utána meghív egy aszinktron föggvényt ami egy aszinkron hívás

                                /|
                               / |
                       -------/  | - a visszatérést következő órán vesszük
                       ^         |
```

```
                        |          ˇ - ezek itt `callback`-ek
|---[cl1]------[function]----[cl2]---->


            |----u1.5----------
            |                 |
|---[u1]---[]--[2]--[]--[u3]---[1.5]----[2.5]---------->
                     |               |
                     |               |
                     ------u2.5-------
```

a fetch api promise-okkal dolgozik, amiben annyi extra van hogy annak 3 állapota van:
- pending, azon belül:
    - fulfilled
    - rejected

sima callback lényege hogy átadj a függvénynek egy függvényt amit benne meghívsz
promise "then chain"

callback
    |
    ˇ
promise https://github.com/siposm/bprof-frontend-weekly/blob/master/js-02/promise.html
    |
    ˇ
async-await

azért promise a neve, mert nem tudom hogy mi lesz ott de valami lesz, és ha kész akkor beteljesült és akkor tartotta az igértet

#7. TANANYAG JavaScript fetch, async await 2025.04.03

fetch("url") : Promise...

a promiseok aszinkron módon működnek

Promise 2+1 állapot
    '-> Pending
            '-> resolve
            '-> reject

https://iqratechnology.com/wp-content/uploads/2024/01/JS-Promise.png

Az órán vett kódok:

async await hogyan kapcsolódik a promise-hoz?

első fájl: promise-async-await.html
https://github.com/siposm/bprof-frontend-weekly/blob/master/js-03/promise-async-await.html

az await-tel a then ágakat tudod megspórolni.
asyn await a promisok köré egy szintatikai "wrapper"

második fájl: xss.html
https://github.com/siposm/bprof-frontend-weekly/blob/master/js-03/xss.html

az input mezőre mindig úgy nézünk hogy potenciális támadói felület.

Mindig kell szűrni az adatokat, mind frontenden mint backend-en, mert ennek hiányát a támadók kihasználhatják.

pl innerHTML segítségével beszúrhatnak a kódba olyan dolgokat amik oldal megnyitásakor lefutnak. És átirányíthatnak a saját backendjükre.

mi van a js hátterében? JS engine

```
JS Engine
    '-> Call stack | verem, LIFO elven működik
            '-> execution context -> itt tárolódnak el a primitív típusok
    '-> Heap
          '->   objects in memory
          '-> Garbage Collector kezeli azaz tisztítja

execution context ->végrehajtási kontextusok

primitive type -> boolean, number, string
```

** Néhány JS engine:*

- V8 -> google
- SpiderMonkey -> firefox
- JavaScriptCore -> Safari

JS engine az ami az ECMAScript alapján készül. Maga az engine C++-ban van írva

** execution context:*
https://media.licdn.com/dms/image/v2/D4D12AQH8UpsE1PYf5Q/article-cover_image-shrink_600_2000/article-cover_image-shrink_600_2000/0/1681056424281?e=2147483647&v=beta&t=kmsIAJeFBK-YphUljFP5GibFw7p35peVDAW5Faxvvho

ez egy jobb példa: https://simonzhlx.github.io/images/execution_stack.png

Mindezek köré jön a `JS runtime`

ez magába foglalja a JS engine. az engine a szíve, de ez nem elég

```
JS runtime
    '-> Web api -> böngésző biztosítja
            '-> DOM
            '-> Fetch
            '-> Timer
    '-> Callback Queue / Task queue | FIFO
```

Másik ismert runtime például a Node.js, ez egy runtime környezet. Másik környezet értelemszerűen a böngésző
Ehhez hasonló:
c# maga a nyelv amihez kell egy .Net runtime környezet

** Callback Queue <-event loop-> call stack*

```
    '-> FIFO
```

Amikor a fetch meghívódik az bekerül a callback queue-be akkor közben folyton nézi az event loop hogy mikor tudná berakni a call stack-be
fetch-nek a hívása bekerül a callback queue-ba ami ezurán a call stackbe

Callback queue-ban nincs végrehajtás, az az enginen belül van.

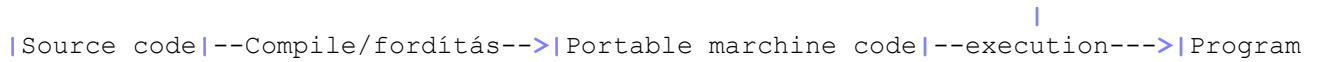
https://www.lydiahallie.com/blog/event-loop

érdekesség: https://www.jsv9000.app/

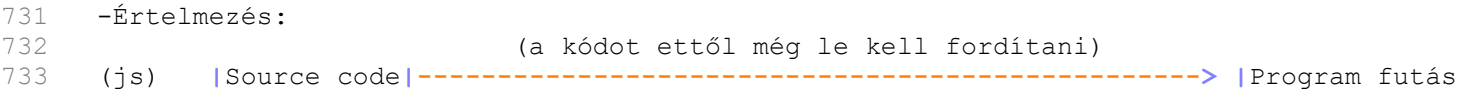
JS melyik két nagy nyelvi család egyikébe tartozik? interpretált vagy compiled?

interpreted -> értelmezett - interpretált

-Fordítás: itt a végrehajtás bármikor akár évekkel később is megtörénhet



-Értelmezés:



sorról sorra végrehajtás

- J.I.T.:
just-in-time compilation

a végrehajtás egyből megtörétnik
|
|source code|--compile fordítás--->|NOT portable machine code|---execution----> | Program futás|

JIT:
végrehajtás egyből
|src code| -> |parsing| -AST-> |compilation| -machine code/byte code-> |execution| (call stacken történik)
|-<--------<---|optimalization|-----<-|

modern js-re már nem teljesen igaz hogy interpretált hanem már inkább a JIT valósul meg

** AST ->abstract syntax tree (Absztrakt szintaxtis fa)*
https://astexplorer.net/

Scopes
- Global scope
- block scope
- function scope

a `var` változóval az a gond hogy az globalis, nem kezeli a blockszintűséget

a weboldal egy statikus valami ahol adatot közölsz.
webalkalmazásnál nem csak közlök hanem adatot fogadok is pl bejelentkezés
régen nem voltak webalkalmazások.

graceful degradation ~> választékos lebutítás => **top-down módszer** föntről lefele megy építkezés szempotjából
    - "minden maxos", erre lövök először. Max-os legúljabb rendszerre fogok koncentrálni, hogy azon fusson
    - viszont -> altenatívát biztosítok -> lebutított verzióban elérhető
              -> funkcionalitás nem sérül

---

progressive development ~> fokozatos fejlesztés => **buttom-up módszer** lentről fölfele megy építkezés szempotjából
    - minimális stabil alapot csinálok, ami minden környezetben használható
    - ezután -> fokozatosan építem/javítom

Imperatív és dekleratív UI kezelés : imperative-declerative.html
https://github.com/siposm/bprof-frontend-weekly/blob/master/js-03/imperative-declarative.html
Imperatív
    '-> hogyan csinálom meg

dekleratív
    '-> mit szeretnék látni
    keretrendszerek használják

dekleratív sajátossága
    adatforrás
        [] ---------------> UI leképezés

            adatfolyam iránya

CORS errors

ez valójában nem hiba hanem biztonsági mechanizmus, ami a szerverekhez kapcsolódik

Ez arra szolgál, hogy kifejezetten engedélyezzen egyes több eredetű kérelmet, míg másokat elutasítson

lokális fejlesztésnél is belefudhatunk a cors error-ba. Ennek utána kell nézni hogyan kell a backend-ben beállítani hogy ez működjön.

js modules fájl-t megnézni

https://github.com/siposm/bprof-frontend-weekly/tree/master/js-03/js-modules

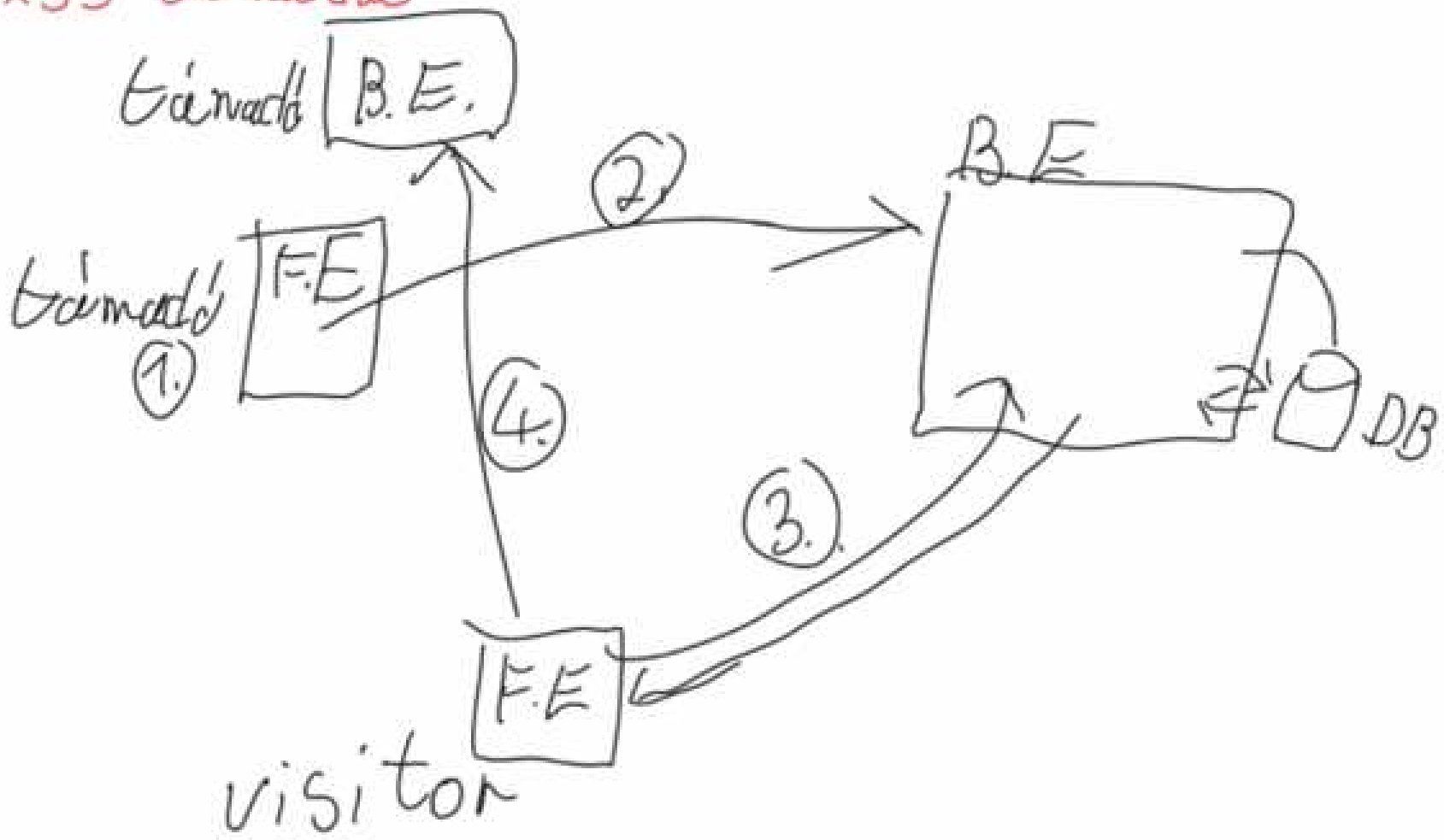
```
html -ben:
<script type="module" src="logic.js"></script>
                '-> azért kell module-nak nevezni, mert importálunk benne

functions.js -ben:
export { add, div}
export class Calculator {
    ...
}

logic.js -ben:
import {add, Calculator, div} { from "./functions.js"}
```

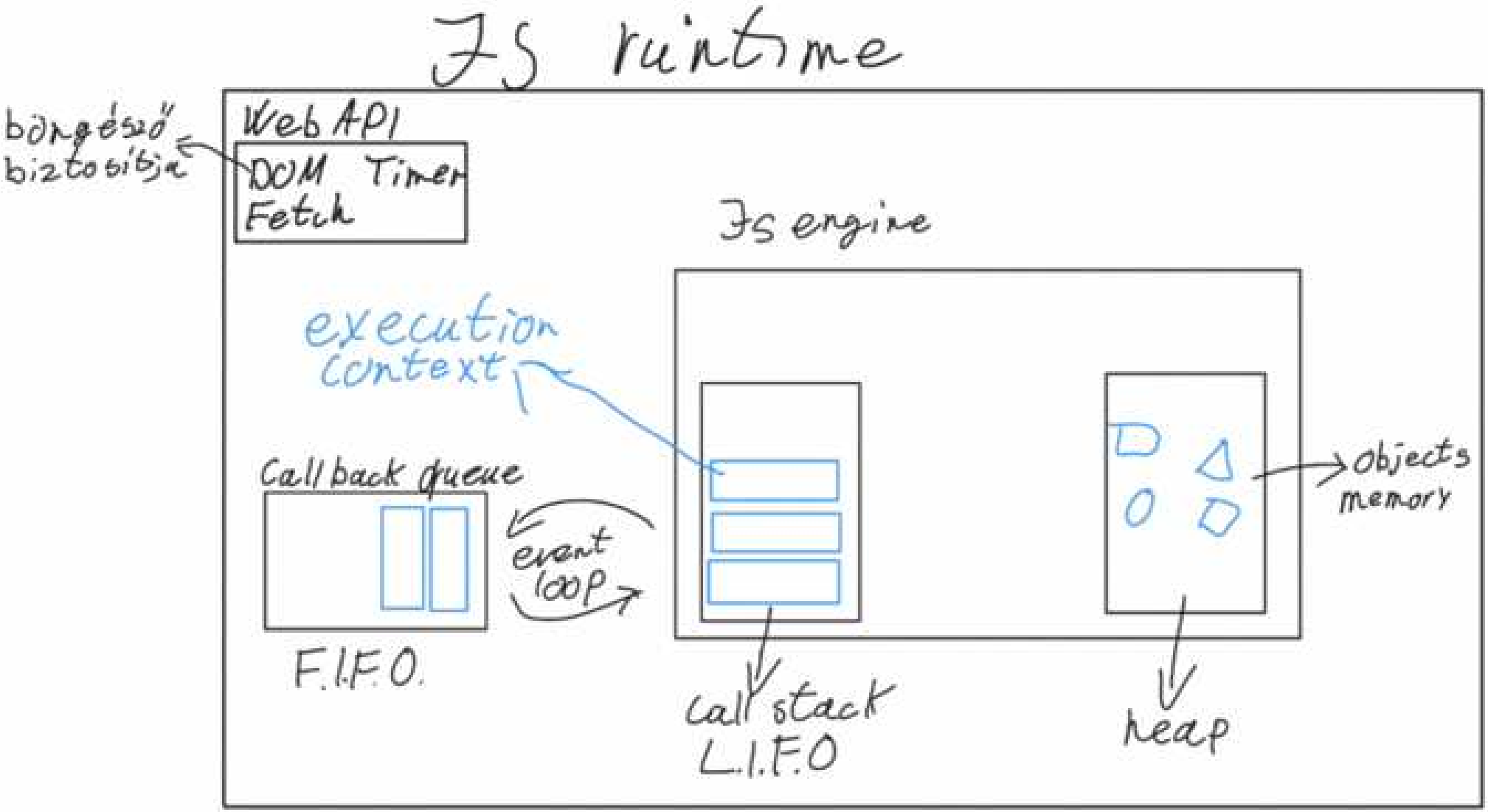
developer-crud

https://github.com/siposm/bprof-frontend-weekly/tree/master/js-03/developer-crud

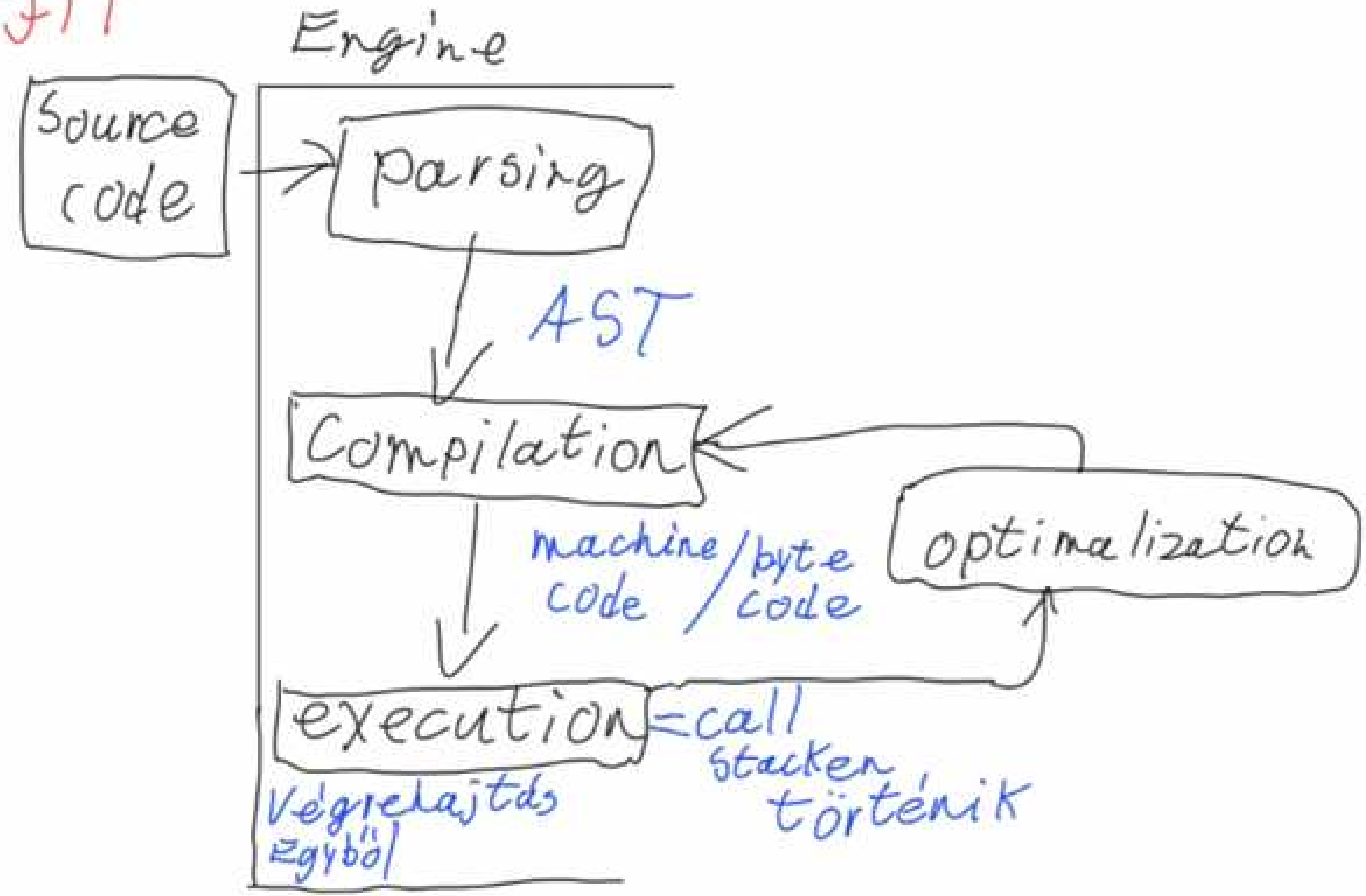
`ez jól jöhet a féléves elkészítéséhez`

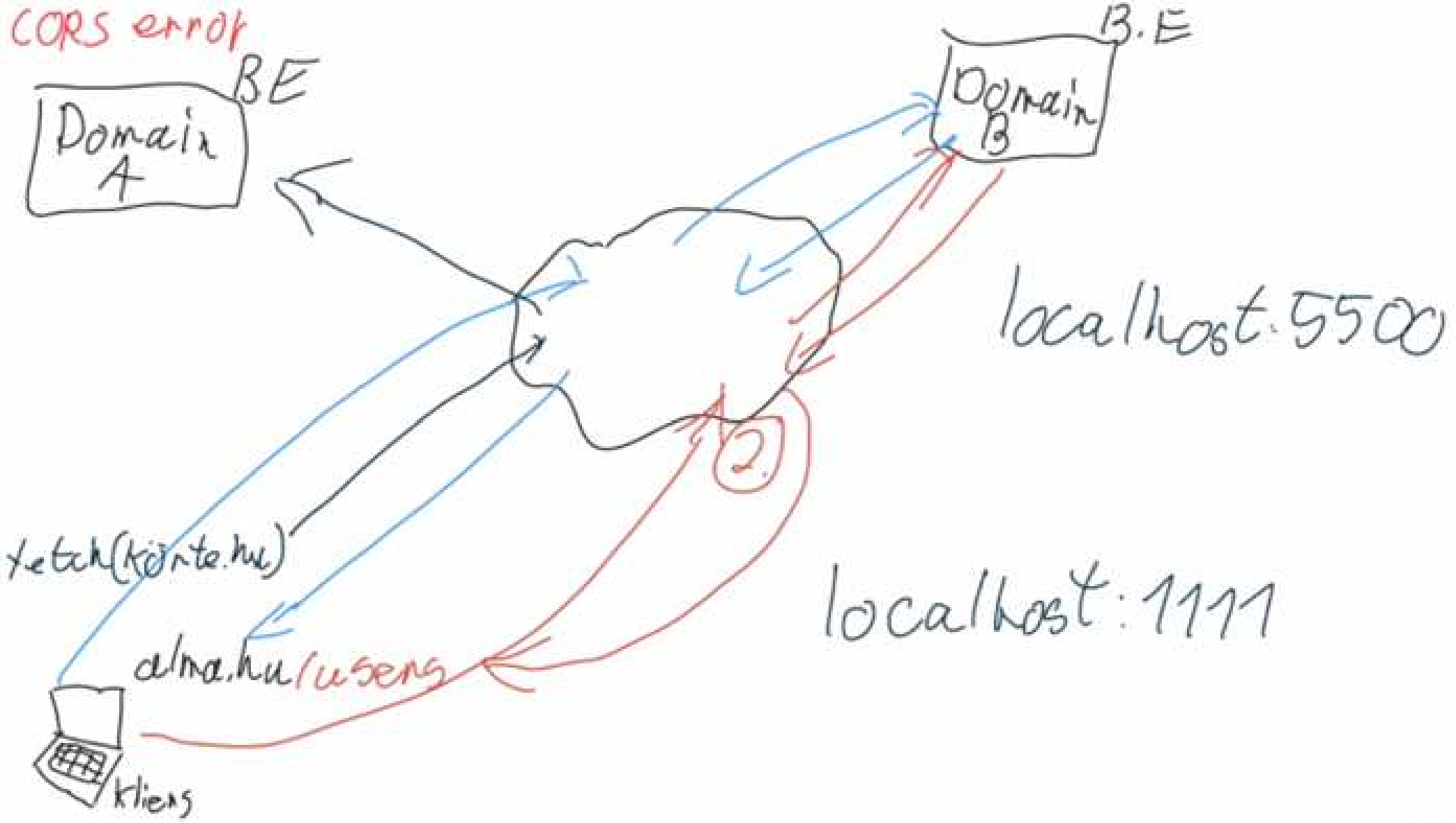
XSS támadás




JS runtime
böngésző biztosítja
Web API
DOM Timer
Fetch
JS engine
execution context
Call back queue
event loop
F.I.F.O.
call stack
L.I.F.O
heap
objects memory

JIT


Engine
Source code
parsing
AST
Compilation
optimalization
machine code / byte code
execution = call stacken történik
Végrehajtás egyből


CORS error
BE
Domain A
B.E
Domain B
localhost:5500
fetch(kornte.hu)
alma.hu/users
localhost:1111
klienS