WEB-programozás I.
Előadás beadandó feladat

Pál Bence
GHXXQA

Mérnökinformatikus Bsc

2025

Tartalomjegyzék

Bevezetés

Ebben a dokumentációban szeretném bemutatni a Webprogramozás 1 tantárgy előadásához készített beadandó projektemet. A feladat célja az volt, hogy egy önállóan elkészített weboldalt hozzak létre, kizárólag kliensoldali technológiák (HTML, CSS, JavaScript) felhasználásával, amely legalább öt különböző oldalt tartalmaz, és többféle interaktív elemet is használ.

A projekt során többféle webes megoldást is kipróbáltam: készítettem egy reakcióidő mérő alkalmazást Reactben, megvalósítottam egy kő-papír-olló játékot objektumorientált JavaScript segítségével, valamint grafikont is megjelenítettem Chart.js segítségével. Ezen kívül kapcsolatba léptem egy API-val AJAX-on keresztül, illeA projekt során többféle webes megoldást is kipróbáltam: készítettem egy reakcióidő mérő alkalmazást Reactben, megvalósítottam egy kő-papír-olló játékot objektumorientált JavaScript segítségével, valamint grafikont is megjelenítettem Chart.js segítségével. Ezen kívül kapcsolatba léptem egy API-val AJAX-on keresztül, illetve kipróbáltam a Server-Sent Events (SSE) technológiát is, amivel valós idejű adatokat tudtam megjeleníteni.tve kipróbáltam a Server-Sent Events (SSE) technológiát is, amivel valós idejű adatokat tudtam megjeleníteni.

A dokumentáció célja, hogy részletesen bemutassam az alkalmazás működését, a megvalósított feladatpontokat, valamint azt, hogyan és hol valósítottam meg ezeket a weboldalon belül. Mindezt képernyőképekkel is illusztrálom.

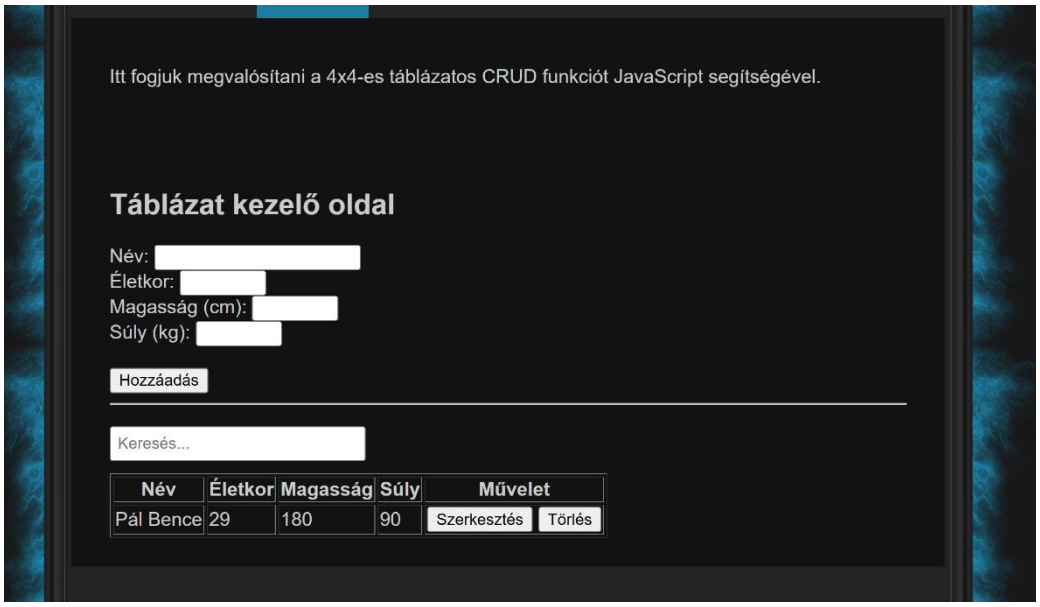
1. A kezdőlap, megjelenés/design és az első (Táblázat) funkció

A feladatot a WEB-programozás I. laboron tanultak szerint kezdtem el megvalósítani. A weboldal alapját az index.html állomány adja, amelyből a többi aloldal is kiindul. A projekt során arra törekedtem, hogy minden oldal egységes kinézetű legyen, és ugyanazt a szerkezetet használja: szerepel benne egy fejléc (header), vízszintes menü (nav), oldalsáv (aside), a fő tartalom (div#content) és egy lábléc (footer).

A menüsáv vízszintesen helyezkedik el az oldal tetején, és minden menüpont az adott aloldalra vezet. Az aktuális oldal menüpontját külön is kiemeltem, az active osztály segítségével, így mindig egyértelmű, hol jár a látogató. A menü minden oldalon azonos szerkezetű, így könnyen használható.

A fejléc tartalmaz egy h1 elemet, amely az egész oldal közös címe: **"WEB-programozás I. – Előadás beadandó feladat"**

A dizájn kialakításánál sötét háttérrel és világos szöveggel dolgoztam. A háttér és a fejléc külön háttérképet kapott, hogy esztétikusabb legyen. A tartalmi részek átlátszó fekete háttéren jelennek meg, ezzel is növelve a kontrasztot. A betűk színe világosszürke, a linkek és a kiemelések pedig ciánkék árnyalatot kaptak. A betűtípus egyszerű és jól olvasható, Arial.



1.1. Táblázat – CRUD műveletek, keresés és rendezés

Ezen az oldalon egy dinamikusan kezelhető, 4 oszlopos táblázatot hoztam létre, amelyben a felhasználó hozzáadhat, szerkeszthet, törölhet adatokat, illetve kereshet és rendezhet is azok között. A cél az volt, hogy egy teljes CRUD (Create, Read, Update, Delete) megoldást készítsek tisztán JavaScript használatával, háttérszerver nélkül.

A felület az alábbi mezőket tartalmazza:

- **Név**
- **Életkor**
- **Magasság (cm)**
- **Súly (kg)**

A mezők kitöltése után a *Hozzáadás* gombra kattintva az adatok megjelennek a táblázatban egy új sorban.

1.2. Hozzáadás és szerkesztés – submit eseménykezelő

A fő eseményfigyelő a következőképp indul:

```
form.addEventListener("submit", function (e) {
    e.preventDefault(); // ne töltse újra az oldalt
    ...
});
```

Ez gondoskodik róla, hogy az űrlap beküldése ne frissítse újra az oldalt, és a JavaScript kezelje a tartalom hozzáadását. A mezők értékeit JavaScripttel olvasom ki, és vagy új sort hozok létre, vagy módosítok egy meglévőt:

```
if (szerkesztettSor) {
    // meglévő sor frissítése
} else {
    // új sor hozzáadása
}
```

1.2.1. Szerkesztés és törlés – dinamikusan hozzárendelt eseménykezelők

Minden újonnan létrehozott sorhoz két gomb tartozik:

```
<td>
  <button class="editBtn">Szerkesztés</button>
  <button class="deleteBtn">Törlés</button>
</td>
```

A Szerkesztés gomb előtölti az űrlapot az adott sor adataival, míg a Törlés gomb egyszerűen eltávolítja a sort a DOM-ból:

```
row.querySelector(".deleteBtn").addEventListener("click", function ()
{
    row.remove();
});
```

1.2.2. Keresés – valós idejű szűrés JavaScript-tel

A táblázat felett található keresőmező lehetővé teszi az azonnali szűrést:Az egyenes cső veszteségtényezője a következő összefüggéssel számítható:

```
document.getElementById("searchInput").addEventListener("input",
function () {
    const keresett = this.value.toLowerCase();
    ...
});
```

A row.style.display értékét módosítom attól függően, hogy a sor szövege tartalmazza-e a keresett kifejezést.

1.2.3. Rendezés – oszlopokra kattintva

A rendezés minden oszlopra működik, szám és szöveg típus szerint is:

```
header.addEventListener("click", () => {
    const oszlopIndex = parseInt(header.dataset.index);
    ...
});
```

A rendezesAllapot objektumban tárolom, hogy egy oszlop épp növekvő vagy csökkenő sorrendben van-e. A sorok újrarendezés után újra hozzáadásra kerülnek a táblázathoz:

```
tableBody.innerHTML = "";
sorok.forEach(sor => tableBody.appendChild(sor));
```

Felhasználói élmény

Az oldal felépítése, dizájnja egyezik a többi aloldallal. A tartalom a div#content szakaszba került, a menüpont pedig active osztályt kapott, hogy látszódjon, épp melyik oldalon járunk. Az adatok rögzítése és kezelése teljes mértékben a kliensoldalon történik, nincs szükség háttérszerverre vagy adatbázisra.

2. HTML5 API oldalam – Bevezetés

Ezen az oldalon különböző HTML5 API-k működését mutatom be önálló példák segítségével. A látogató gombok segítségével válthat az egyes technológiák között, ezek külön-külön szekciókban jelennek meg. A célom az volt, hogy minden példa egyszerű, de jól érthető módon demonstrálja a kiválasztott API működését – akár tanulási célra is használható formában.

Az alábbi technológiákat mutatom be:

- Web Storage (LocalStorage)
- Geolocation
- Canvas
- Drag & Drop
- SVG
- Web Worker
- Server-Sent Events

A kezelőfelület egységesen illeszkedik az oldal többi részéhez: sötét alapszín, világos szöveg, az aktív menüpont kiemelve, és minden szekció reszponzívan viselkedik.

2.1. Web Storage (LocalStorage)

Az első bemutatott HTML5 technológia a **Web Storage API**, azon belül is a **LocalStorage**. Ennek segítségével adatokat lehet elmenteni a böngészőben anélkül, hogy azt újratöltéskor elveszítenénk.

Az oldalon a felhasználó beírhat egy tetszőleges szöveget, majd a „Mentés” gombbal elmentheti azt a böngésző tárolójába. A mentett érték az oldal újratöltése után is megmarad.

Fontosabb kódrészletek:

```
// Ha van elmentett adat, megjelenítjük
const stored = localStorage.getItem("mentettSzoveg");
if (stored) {
    savedText.textContent = "Elmentve: " + stored;
}
```

Ez a rész gondoskodik arról, hogy ha az oldal betöltésekor van korábban elmentett adat, akkor azt rögtön megjelenítse a felhasználónak.

```
// Mentés gomb eseménykezelő
saveBtn.addEventListener("click", () => {
    const value = input.value;
    localStorage.setItem("mentettSzoveg", value);
    savedText.textContent = "Elmentve: " + value;
});
```

Ez az eseményfigyelő felel azért, hogy a „Mentés” gomb megnyomásakor a szöveg ténylegesen bekerüljön a localStorage-ba. Ezután az érték ki is íródik a felhasználónak.

Ez egy **egyszerű, de hasznos példa** arra, hogyan lehet kliensoldalon perzisztens (tartós) adatot tárolni szerver használata nélkül. A localStorage nagy előnye, hogy gyors, és nem igényel háttér-infrastruktúrát – ideális tanuláshoz, vagy kisebb alkalmazásokhoz.

2.2. AGeolocation API

A második HTML5 technológia, amit bemutattam, a **Geolocation API**, amely lehetővé teszi a felhasználó földrajzi pozíciójának meghatározását. Ez egy böngészőbe épített szolgáltatás, amely megfelelő engedély esetén visszaadja a szélességi és hosszúsági koordinátákat.

Ez a példa egy gombra kattintva próbálja lekérni a helyzetet, és kiírja a koordinátákat. Fontos megjegyezni, hogy a **Geolocation API csak HTTPS kapcsolat alatt működik**, különben a böngésző megtagadja a hozzáférést a helyadatokhoz.

Fontosabb kódrészletek:

```
if (navigator.geolocation) {
    geoResult.textContent = "Helyzet lekérése folyamatban...";
```

Ez a rész ellenőrzi, hogy a böngésző támogatja-e a navigator.geolocation API-t. Ha igen, megkezdjük a pozíció lekérését.

```
navigator.geolocation.getCurrentPosition(
    position => {
        const lat = position.coords.latitude.toFixed(5);
        const lon = position.coords.longitude.toFixed(5);
        geoResult.textContent = `Szélesség: ${lat}, Hosszúság: ${lon}`;
    },
```

Sikeres pozíciólekérés esetén a getCurrentPosition metódus callback függvényében megkapjuk a position objektumot, amiből kiolvassuk a koordinátákat. Az értékeket toFixed(5)-tel kerekítettem, hogy áttekinthetőbbek legyenek.

```
error => {
    let message = "Ismeretlen hiba történt.";
    switch (error.code) {
        case error.PERMISSION_DENIED:
            message = "A felhasználó megtagadta a helymeghatározást.";
            break;
        ...
    }
    geoResult.textContent = "Nem sikerült lekérni a helyzetet: " +
message;
}
```

Ha hiba történik, itt kezeljük az összes lehetséges esetet: pl. a felhasználó megtagadta az engedélyt, vagy technikai okból nem elérhető a pozíció. A felhasználó egyértelmű visszajelzést kap a hiba típusáról.

Ez a megoldás jól demonstrálja, hogyan lehet **felhasználói interakcióval, valós időben lekérni érzékeny adatot**, és azt felhasználóbarát módon megjeleníteni. A hibakezelés külön figyelmet kapott, hogy minden esetben értelmes visszajelzés történjen.

2.3. Canvas API

A harmadik példában a **Canvas API**-t mutatom be, amely lehetőséget ad arra, hogy JavaScript segítségével grafikát rajzoljunk egy <canvas> elemre. Ez egy nagyon hasznos technológia például játékok, vizualizációk vagy interaktív animációk készítéséhez – minden közvetlenül a böngészőben történik.

Az oldalon található gomb (Rajzolj valamit) megnyomására egy **téglalap**, egy **kör**, valamint egy **szövegfelirat** jelenik meg a vásznon. A rajzolás minden kattintásnál újraindul, így a korábbi tartalom mindig törlődik.

Fontosabb kódrészletek:

```
const ctx = canvas.getContext("2d");
```

A getContext("2d") hívással megszerezzük a vászon (canvas) rajzolási kontextusát. Ez szükséges ahhoz, hogy később alakzatokat és szöveget jelenítsünk meg a felületen.

```
ctx.clearRect(0, 0, canvas.width, canvas.height);
```

Mielőtt új dolgokat rajzolnánk, előbb töröljük a vászon előző tartalmát. Ez biztosítja, hogy minden gombnyomás friss képet eredményezzen.

```
ctx.fillStyle = "skyblue";
ctx.fillRect(10, 10, 100, 60);
```

Ez a rész rajzol egy világoskék téglalapot a bal felső sarok közelébe. A fillStyle szín beállítása után a fillRect() metódus rajzolja ki az alakzatot.

```
ctx.beginPath();
ctx.arc(200, 60, 30, 0, 2 * Math.PI);
ctx.fillStyle = "orange";
ctx.fill();
```

Itt egy narancssárga kört rajzolunk a vászon közepére. Az arc() metódus egy kört vagy körívet rajzol, amit a fill() metódussal töltünk ki.

```
ctx.font = "16px Arial";
ctx.fillStyle = "black";
ctx.fillText("Canvas példa", 100, 140);
```

Végül egy szöveget írunk ki a vászonra. A betűméret és stílus az ctx.font beállításával történik, míg a fillText() helyezi el a szöveget.

Ez a példa jól szemlélteti, hogyan lehet grafikát rajzolni kód segítségével, és hogyan lehet különböző elemeket (alakzat, szöveg) kombinálni egy interaktív felületen. A Canvas API ugyan egyszerű példával indult, de komolyabb projektekhez is jól skálázható.

2.4. Drag & Drop

A következő HTML5 API példa a **Drag & Drop**, amely lehetővé teszi a felhasználók számára, hogy egy kijelölt elemet egérrel megragadjanak, áthúzzanak egy célterületre, és ott elhelyezzék. Az oldalon egy piros négyzet látható, amit a felhasználó áthúzhat egy szürke szaggatott határvonalú dobozba.

Amint az elem a célterületre kerül, **dinamikusan újrapozícionálódik** az egér helyzetéhez igazítva.

Fontosabb kódrészletek:

```
dragMe.addEventListener("dragstart", (e) => {
    e.dataTransfer.setData("text/plain", "draggedBox");
});
```

A `dragstart` esemény során megadjuk az adatot (`draggedBox`), amelyet a célterület felismer a „ledobás” során. Ez segít azonosítani, hogy valóban a húzható elemről van-e szó.

A célterület (dropZone) alapból nem fogadja el a húzást, ezért e.preventDefault() segítségével engedélyezzük azt. Emellett vizuális visszajelzést is adunk – a háttér világoskékre vált, hogy jelezze: ide lehet ejteni az elemet.

A drop esemény során kiszámoljuk az egér aktuális helyzetét a célterülethez képest, majd **relatív pozícionálással** oda helyezzük a piros négyzetet. Az appendChild() metódussal fizikailag is áthelyezzük az elemet a célterület DOM-jába.

```
dropZone.addEventListener("drop", (e) => {
    e.preventDefault();

    const x = e.clientX - zoneRect.left;
    const y = e.clientY - zoneRect.top;

    dragMe.style.position = "absolute";
    dragMe.style.left = x - dragMe.offsetWidth / 2 + "px";
    dragMe.style.top = y - dragMe.offsetHeight / 2 + "px";
});
```

Ez a példa jól mutatja, hogy a Drag & Drop API nemcsak "átmozgatásra", hanem **interaktív pozícionálásra** is alkalmas, és az események kombinálásával nagyon dinamikus, látványos élményt nyújt a felhasználóknak.

2.5. SVG – Interaktív vektorgrafika

Az oldalon található példa az **SVG (Scalable Vector Graphics)** technológiát használja, amely vektoros ábrák megjelenítésére alkalmas HTML-ben. A beépített SVG elem alapból tartalmaz egy téglalapot és egy kört, de a felhasználó további köröket is elhelyezhet benne az egér segítségével.

Ez a példa azt szemlélteti, hogy az SVG nemcsak statikus képek megjelenítésére alkalmas, hanem dinamikusan bővíthető és módosítható is JavaScript segítségével.

```
const svg = document.getElementById("svgArea");

svg.addEventListener("click", (e) => {
    const rect = svg.getBoundingClientRect();
    const x = e.clientX - rect.left;
    const y = e.clientY - rect.top;
```

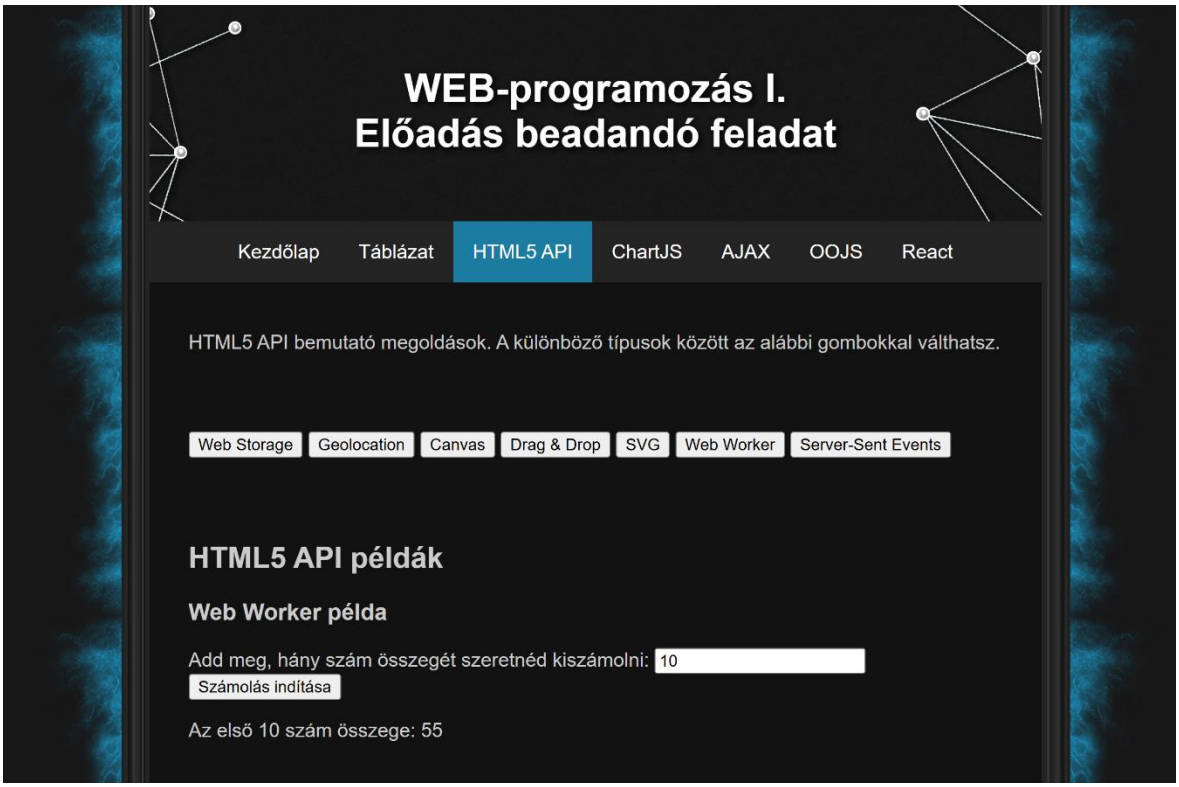
A felhasználó kattint egy pontban az SVG területen, és az e.clientX, e.clientY értékeket az SVG saját koordinátarendszerére konvertáljuk.

```
const newCircle =
document.createElementNS("http://www.w3.org/2000/svg", "circle");
newCircle.setAttribute("cx", x);
newCircle.setAttribute("cy", y);
newCircle.setAttribute("r", 20);
newCircle.setAttribute("fill", "purple");
```

A createElementNS() segítségével új SVG elemet hozunk létre, jelen esetben egy **20 pixeles lila kör** formájában. Ezután az appendChild() metódussal hozzáadjuk az SVG DOM-hoz, így az megjelenik a felhasználó által kattintott helyen.

Ez a példa egyszerű, de jól bemutatja, hogy az SVG grafika **programozottan is bővíthető**, így például diagramok, szerkesztők, térképek vagy játékok részeként is használható.

2.6. Web Worker – Háttérben végzett számítás



A HTML5 Web Worker API lehetővé teszi, hogy számításokat külön szálon futtassunk, így az **UI (felhasználói felület)** nem akad meg hosszabb műveletek közben sem. Ezzel megelőzhetjük, hogy az oldal "lefagyjon" egy nagyobb számítás végzésekor.

Ebben a példában a felhasználó megadhat egy számot, és a rendszer kiszámolja az 1-től n-ig terjedő számok összegét a háttérben, egy **külön JavaScript fájlban futó worker segítségével**.

Fontosabb kódrészletek:

workerapi.js (főszál – a felhasználói oldalon)

```
const worker = new Worker("js/html5/worker.js");
```

Ezzel a sorral hozunk létre egy új Web Worker példányt, amely a worker.js fájlt futtatja külön szálon.

```
worker.addEventListener("message", (event) => {
    workerResult.textContent = `Az első ${numberInput.value} szám
összege: ${event.data}`;
});
```

A gomb megnyomásakor az n értékét elküldjük a workernek. Ez egy aszinkron folyamat, nem blokkolja a fő szálat.

```
startBtn.addEventListener("click", () => {
    const n = parseInt(numberInput.value);
    worker.postMessage(n);
});
```

Amint a worker befejezte a számítást, visszaküldi az eredményt a postMessage() segítségével. A főszál ezt egy eseményen keresztül fogadja és megjeleníti a felhasználónak.

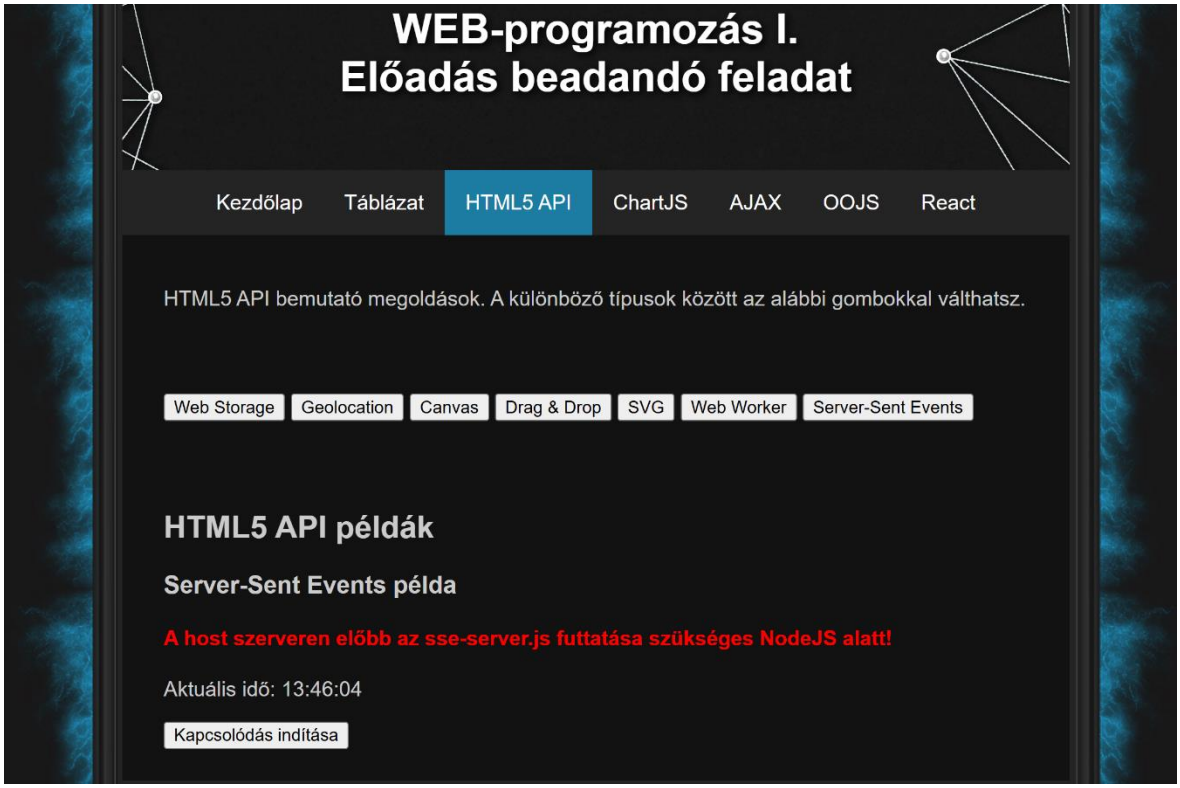
worker.js (a külön szálon futó fájl):

```
onmessage = function (event) {
    const n = event.data;
    let sum = 0;

    for (let i = 1; i <= n; i++) {
        sum += i;
    }

    postMessage(sum);
};
```

A worker.js nem fér hozzá a DOM-hoz, de fogad üzenetet (onmessage) és visszaküldi az eredményt (postMessage). Ez a példa egy egyszerű, lineáris összegzést végez el.

Ez a megoldás szemléletes példája annak, hogyan lehet **párhuzamosított számítást** végezni kliensoldalon, miközben a fő felület interaktív marad. Nagyobb adatfeldolgozásnál vagy játékfejlesztésnél ez különösen hasznos lehet.

2.7. Servers-Sent Events (SSE)



A HTML5 API példák közül a legösszetettebb a **Server-Sent Events**, amely lehetővé teszi, hogy egy szerver folyamatosan adatokat küldjön a böngésző felé egy tartós kapcsolat segítségével. Ezt gyakran használják valós idejű alkalmazásokban (pl. chat, időjárás, élő adatok).

Az én példámban a szerver másodpercenként elküldi az aktuális időt, amit a kliensoldalon az oldal ki is ír. A kapcsolat folyamatos, nem szükséges frissíteni az oldalt.

Tesztelés és indítás:

A működéshez szükség van egy Node.js alapú kis szerverre is, amelyet egy külön fájl (sse-server.js) tartalmaz. A szerver indítása parancssorban történik:

```
node sse-server.js
```

Ezután a böngésző a http://localhost:3000/events URL-en keresztül kapcsolódik a szerverhez. A működéshez **lokális környezet és Node.js telepítés szükséges**. Ha a szerver nem fut, a kliensoldalon hibaüzenet jelenik meg.

Fontosabb kódrészletek (kliensoldal [sse.js]):

```
const source = new EventSource("http://localhost:3000/events");

source.onmessage = function (event) {
    sseData.textContent = "Aktuális idő: " + event.data;
};
```

Az EventSource objektum tartós kapcsolatot létesít a megadott URL-lel. Az onmessage esemény minden érkező üzenetkor fut, és megjeleníti az adatot a felhasználónak.

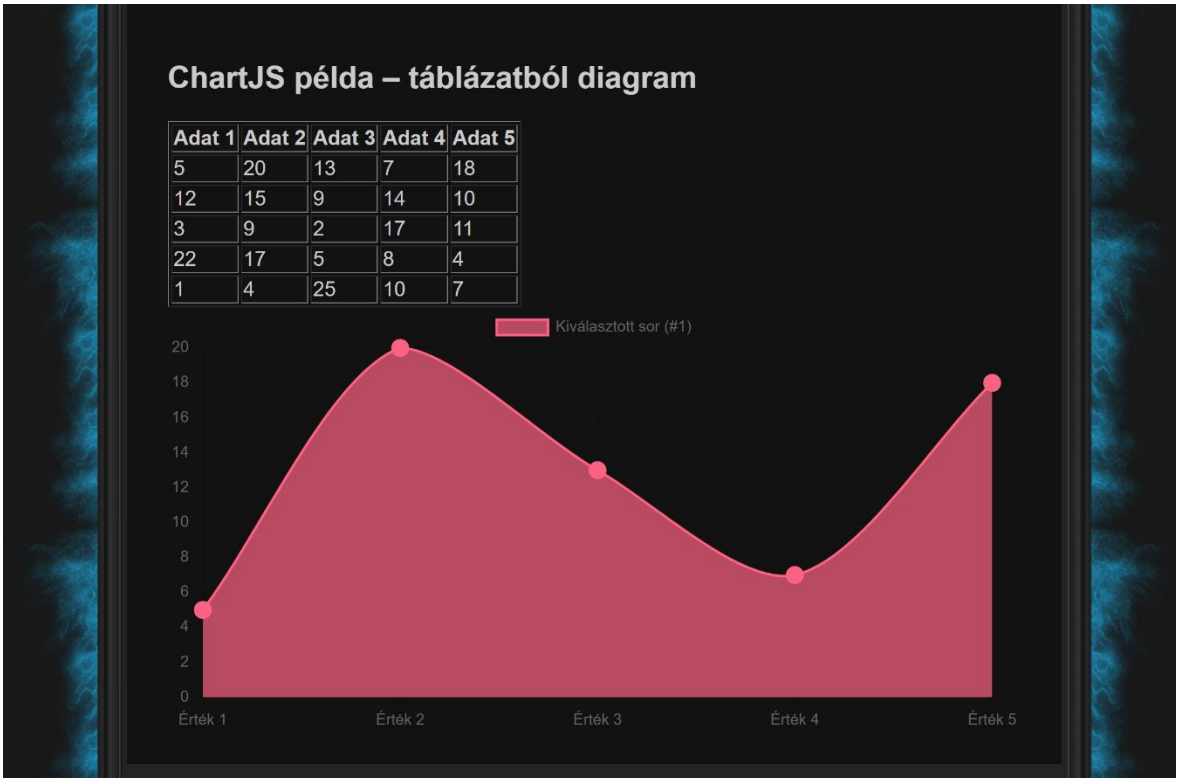
Fontosabb kódrészletek (szerveroldal [sse-server.js]):

```
if (req.url === "/events") {
    res.writeHead(200, {
        "Content-Type": "text/event-stream",
        ...
    });

    setInterval(() => {
        const now = new Date().toLocaleTimeString();
        res.write(`data: ${now}\n\n`);
    }, 1000);
}
```

A szerver másodpercenként küld egy sort az aktuális idővel. A Content-Type: text/event-stream fejléc biztosítja, hogy a böngésző SSE kapcsolatként kezelje a válaszfolyamot.

Ez a példa remek demonstrációja annak, hogyan működik a **valós idejű kommunikáció a kliens és szerver között**, és mennyire egyszerűen beállítható egy alapszintű SSE kapcsolat saját környezetben.

3. ChartJS



Ebben a példában a ChartJS könyvtár segítségével hoztam létre egy interaktív **vonaldiagramot**. A cél az volt, hogy a felhasználó egy táblázat sorára kattintva az abban szereplő értékeket grafikusan is megtekinthesse. A megoldás dinamikusan frissül, és minden kattintásra új színnel jelenik meg a kiválasztott sor.

A ChartJS egy népszerű, könnyen használható JavaScript könyvtár, amely ideális adatok gyors és látványos megjelenítésére.

Fontosabb kódrészletek:

```
let chart = new Chart(ctx, {
    type: "line",
    data: {
        labels: ["Érték 1", "Érték 2", "Érték 3", "Érték 4", "Érték 5"],
        datasets: [{
            label: "Kiválasztott sor",
            data: [],
            ...
        }]
    },
    ...
});
```

Itt hozom létre az alapértelmezett vonaldiagramot. A `datasets[0].data` értéke egyelőre üres – ez majd kattintásra töltődik fel adatokkal.

```
rows.forEach((row, index) => {
    row.addEventListener("click", () => {
        const values = Array.from(row.cells).map(cell =>
parseFloat(cell.textContent));
        ...
        chart.update();
    });
});
```

Minden sorra eseményfigyelőt helyezek, amely a sor száma alapján:

- kiolvassa a sor értékeit,
- beállítja őket a grafikonra,
- új színt rendel hozzá a `colors` tömbből,
- és frissíti a grafikont.

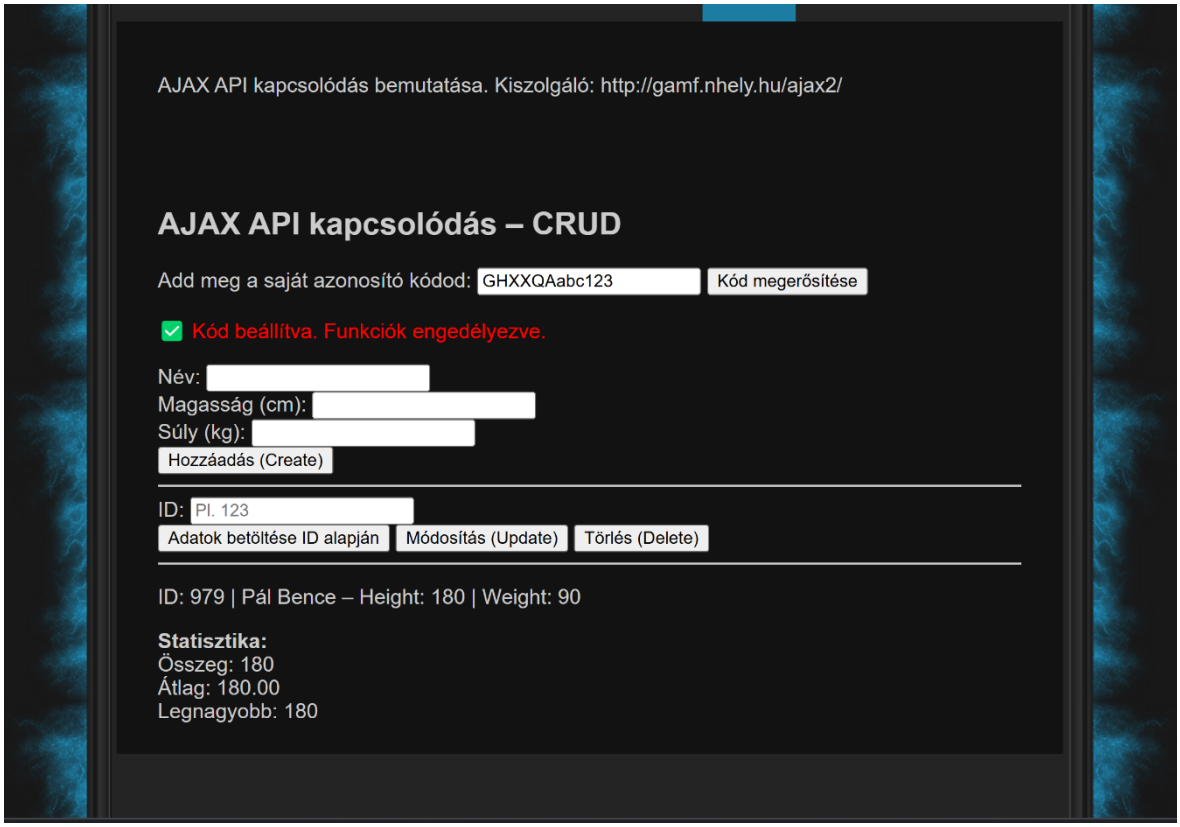
Ez a rész valósítja meg az **interaktív élményt**: nincs újratöltés, minden azonnal megjelenik.

```
chart.data.datasets[0].borderColor = borderColors[index %
borderColors.length];
chart.data.datasets[0].backgroundColor = colors[index %
colors.length];
```

A különböző sorokhoz eltérő színeket társítok, így könnyen megkülönböztethetőek, ha több sort nézünk egymás után.

Ez a feladat remekül demonstrálja, hogy egy egyszerű HTML táblázat **vizuálisan is feldolgozható** egy erőteljes, de könnyen használható JavaScript könyvtár segítségével. A vizualizáció nemcsak látványosabb, hanem segíthet az adatok jobb megértésében is.

4. AJAX



Az AJAX feladat célja az volt, hogy kapcsolódjunk egy **külső API-hoz** (http://gamf.nhely.hu/ajax2/) és azon keresztül végezzünk adatkezelést (CRUD műveleteket). Fontos követelmény volt, hogy minden felhasználó csak a saját adatait kezelhesse – ehhez saját, **Neptun+egyéni azonosító** megadását kértem az oldal betöltése után.

A funkciók a következők:

- **Create** – új rekord létrehozása (név, magasság, súly)
- **Read** – adatok lekérése és statisztika megjelenítése
- **Update** – meglévő rekord módosítása ID alapján
- **Delete** – rekord törlése ID alapján

Kódalapú azonosítás – biztonságosabb megoldás:
A felhasználónak először meg kell adnia egy azonosítót (pl. GHXXQAabc123). Ezután engedélyezem az összes többi műveletet. A code változó globális szinten tárolódik:

```
let code = ""; // Globális változóként mentjük el
...
code = codeInput.value.trim();
readData(); // automatikus betöltés a saját adatokkal
```

Create – új adat létrehozása:

A form beküldésére egy submit eseménykezelő figyel:

```
const body =
`op=create&name=${name}&height=${height}&weight=${weight}&
code=${code}`;

fetch("http://gamf.nhely.hu/ajax2/", {
   method: "POST",
   headers: { "Content-Type": "application/x-www-form-urlencoded"
},
   body
})
```

A felhasználó által megadott adatokat POST kérésben küldöm el a szervernek, majd az űrlapot ürítem, és újra lekérem az adatokat.

Read – adatok lekérése és statisztika:

A readData() függvény meghívásakor nemcsak a rekordokat listázom ki, hanem **magasság statisztikát is számítok**:

```
let heightSum = 0;
let heightMax = -Infinity;
...
const avg = heightSum / data.list.length;
```

Az eredmény HTML-ben jelenik meg a #stats div-ben.

Update – meglévő adat módosítása:

A módosítás két lépésből áll:

1. getDataBtn: Az ID alapján betölti az adatokat a mezőkbe
2. updateBtn: Elküldi az új értékeket ugyanazzal az ID-vel

```
const body =
`op=update&id=${id}&name=${name}&height=${height}&weight=$
{weight}&code=${code}`;
```

Delete – törlés azonosító alapján:

A deleteBtn eseménykezelője elküldi a megfelelő POST kérést:

```
const body = `op=delete&id=${id}&code=${code}`;
```

Sikeres törlés után újra betöltöm a frissített adatlistát.

Ez a megoldás teljes körű adatkezelést valósít meg egy valós API-n keresztül, kliensoldali validációval és biztonságos kódkezeléssel. A fetch() metódus POST kérésekkel működik, és a application/x-www-form-urlencoded formátumot használja, ami egyszerű és kompatibilis.

5. OOJS – Kő-Papír-Olló játék objektumorientáltan

Az objektumorientált JavaScript feladatban egy klasszikus **kő-papír-olló** játékot valósítottam meg, három különálló osztály segítségével. A célom az volt, hogy a játékmenetet átlátható, újrafelhasználható osztályokba szervezzem – hasonlóan a Java vagy C# nyelvben megszokott OOP mintákhoz.

A játék működése egyszerű: a felhasználó választ egy opciót, a gép véletlenszerűen választ egy másikat, majd kiértékeljük az eredményt és megjelenítjük azt.

Felépítés – osztályok

```
class Game {
    constructor() {
        this.choices = ["ko", "papir", "ollo"];
    }

    getResult(playerChoice, aiChoice) {
        ...
    }
}
```

A Game osztály tartalmazza az alaplogikát: az összes választható opciót, valamint a getResult() metódust, amely kiértékeli a játék eredményét a szabályok szerint.

```
class Player extends Game {
    constructor(name) {
        super();
        this.name = name;
    }
}
```

A Player osztály a Game osztályból származik, és egy name tulajdonsággal rendelkezik. Így többféle játékost is tudnánk kezelni – itt most csak a felhasználót.

```
class AIPlayer extends Game {
    constructor() {
        super();
    }

    randomChoice() {
        ...
    }
}
```

Az AIPlayer osztály szintén a Game osztályból származik, és rendelkezik egy randomChoice() metódussal, amely véletlenszerűen választ egy elemet a lehetőségek közül.

DOM kezelés és játék indítása

A gombok a HTML oldalban data-choice attribútummal rendelkeznek. Ezekre eseményfigyelőt teszek:

```
document.querySelectorAll("button[data-choice]").forEach(btn => {
    btn.addEventListener("click", () => {
        ...
    });
});
```

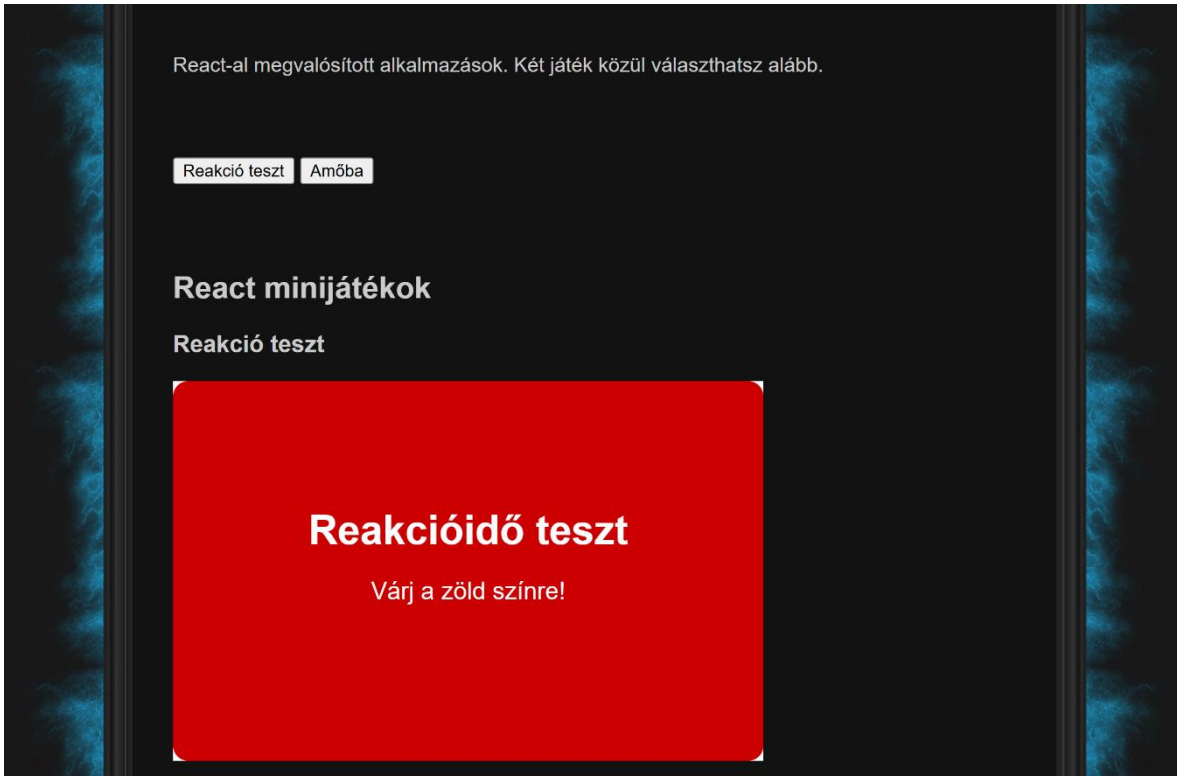
A kattintás hatására:

1. A játékos választása playerChoice
2. A gép válasza aiChoice
3. Az eredmény a getResult() metódussal kerül kiértékelésre
4. A resultDiv HTML elemben jelenik meg az eredmény

```
resultDiv.innerHTML = `
    <strong>Te:</strong> ${playerChoice.toUpperCase()}<br>
    <strong>Gép:</strong> ${aiChoice.toUpperCase()}<br>
    <strong>Eredmény:</strong> ${outcome}
`;
```

Ez a megoldás jól mutatja be az **öröklődést**, az **objektum példányosítást**, valamint **a funkciók szétválasztását** külön osztályokba. A játék könnyen bővíthető (pl. pontszám, új játéktípusok), és tanulási céllal ideális OOJS bevezető.

6. React Reakció teszt



A felhasználó egy kezdőképernyőn kattint egy „Start” gombra. Ezután a képernyő piros színűvé válik, és egy véletlenszerű (2–5 másodperces) késleltetés után **zöldre vált**. A felhasználónak ekkor minél gyorsabban kell kattintania, hogy jó reakcióidőt érjen el.

- **Ha túl korán kattint**, hibaüzenet jelenik meg („Túl korán kattintottál!”).
- **Ha jó időben kattint**, az aktuális reakcióidő milliszekundumban kerül kiírásra.
- A felhasználó új próbát indíthat újra a „Start újra” gombra kattintva.

Főbb React komponensek és logika:

A játékot egyetlen komponens valósítja meg (ReactionTester), amely állapotokat használ a React useState és useEffect hookjai segítségével.

Példák a belső működésből:

```
const [status, setStatus] = useState("idle"); // idle | waiting | ready |
tooSoon
const [startTime, setStartTime] = useState(null);
const [reactionTime, setReactionTime] = useState(null);
```

1. A status változó határozza meg az aktuális állapotot.
2. A startTime rögzíti, mikor vált zöldre a kijelző.
3. A reactionTime pedig kiszámolja, hány ms telt el a kattintásig.

Késleltetés kezelése:

A zöldre váltás ideje egy `setTimeout` segítségével valósul meg, amit a `useEffect()` hook vezérel:

```
useEffect(() => {
  if (status === "waiting") {
    const timeout = setTimeout(() => {
      setStartTime(Date.now());
      setStatus("ready");
    }, Math.random() * 3000 + 2000);
    return () => clearTimeout(timeout);
  }
}, [status]);
```

Ez biztosítja, hogy minden próbálkozásnál eltérő várakozási idő legyen, így a felhasználó nem tud előre számolni rá.

Felhasználói interakció kezelése:

A teljes képernyő egyetlen div-ként működik, és annak onClick eseménye figyeli, hogy mikor kattintott a felhasználó:

```
const handleClick = () => {
    if (status === "ready") {
        const endTime = Date.now();
        setReactionTime(endTime - startTime);
        setStatus("finished");
    } else if (status === "waiting") {
        setStatus("tooSoon");
    } else if (status === "idle" || status === "tooSoon" || status ===
"finished") {
        setReactionTime(null);
        setStatus("waiting");
    }
};
```

Vizualizáció és visszajelzés:

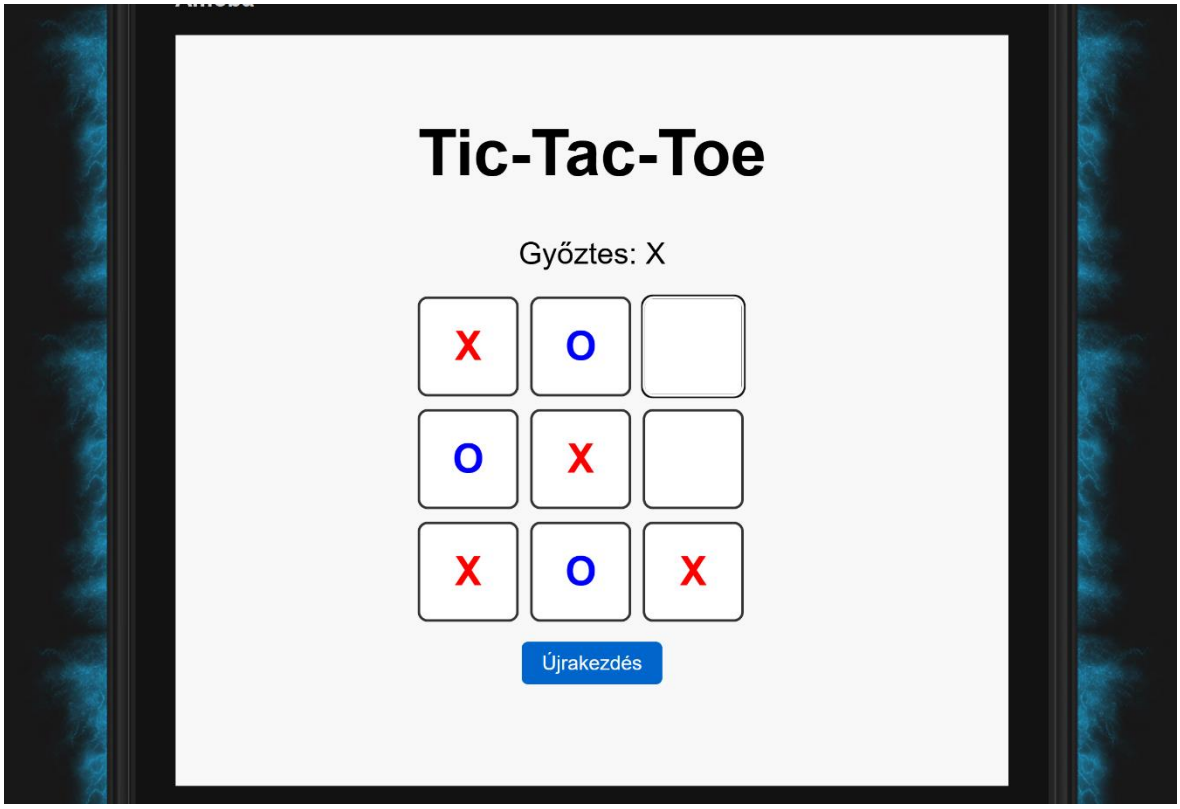
A háttérszín minden állapothoz más:

- red – várakozás
- green – készen áll a kattintásra
- gray – alaphelyzet
- orange – túl korai kattintás

A reakcióidő **nagy, jól olvasható szövegként jelenik meg**, így akár oktatási célra is használható a játék.

Ez a feladat nemcsak React szintaxisgyakorlás volt, hanem a **felhasználói élmény**, a **késleltetések kezelése**, valamint az **interakciók pontos követése** is fontos szerepet kapott. A logika teljes mértékben komponens-alapú, és könnyen továbbfejleszthető például statisztikával vagy pontozással.

7. React – Amőba játék (Tic-Tac-Toe)



A React rész kiegészítéseként egy klasszikus **Amőba játékot (Tic-Tac-Toe)** is készítettem. A cél az volt, hogy két játékos (X és O) felváltva léphessen a táblára, és az alkalmazás automatikusan felismerje a győztest vagy a döntetlent. A játék teljesen kliensoldali, újratölthető, és nem használ külső csomagokat vagy háttérszervert.

Játékmenet és működés:

- A játékmező egy 3×3-as rács, amelyre a felhasználók felváltva kattinthatnak.
- Az első játékos az **X**, a második az **O**.
- A React komponens állapota alapján követi, ki a soros, és mikor van vége a játéknak.
- Ha valaki nyer, vagy a mező megtelik (döntetlen), a rendszer megállítja a játékot és kiírja az eredményt.
- A „Új játék” gombbal újrakezdhető a játék.

Főbb logikai részek:

A játék egyetlen fő komponensből áll, amely a mező állapotát egy tömbben (squares) tárolja:

```
const [squares, setSquares] = useState(Array(9).fill(null));
const [xIsNext, setXIsNext] = useState(true);
```

Mezők kezelése:

A játékmezők kattintásra frissülnek:

```
const handleClick = (index) => {
  if (winner || squares[index]) return;

  const newSquares = [...squares];
  newSquares[index] = xIsNext ? "X" : "O";

  setSquares(newSquares);
  setXIsNext(!xIsNext);
};
```

Itt ellenőrzöm, hogy van-e már nyertes, vagy hogy az adott mező foglalt-e, majd frissítem az állapotot.

Győztes ellenőrzése:

Egy segédfüggvény (calculateWinner) végigellenőrzi az összes nyerési lehetőséget:

```
function calculateWinner(squares) {
    const lines = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8], // sorok
        [0, 3, 6], [1, 4, 7], [2, 5, 8], // oszlopok
        [0, 4, 8], [2, 4, 6]             // átlók
    ];

    for (const [a, b, c] of lines) {
        if (squares[a] && squares[a] === squares[b] && squares[a] ===
squares[c]) {
            return squares[a];
        }
    }
    return null;
}
```

A winner változó alapján a felhasználónak megfelelő visszajelzést adok: ki nyert vagy döntetlen történt.

Újraindítás:

A játék újraindítása egy gomb segítségével történik, ami visszaállítja a squares és xIsNext állapotokat az alapértelmezettre:

```
<BUTTON CLASSNAME="RESET" ONCLICK={RESETGAME}>ÚJRAKEZDÉS</BUTTON>
```

Ez a React projekt kiválóan bemutatja a **komponens-alapú felépítést**, a **state kezelés** alapjait, valamint az **interaktív logika** és feltételes renderelés kombinációját. A játék tovább bővíthető (pl. pontszámok, animációk, több mező), így jó alap egy összetettebb React projekthez.

Melléklet

GitHub Projekt: https://github.com/ImBenny95/gamf-web1-2025-lev3-ea-beadando/

Demo oldal: http://ghxxqa.szakdoga.net/web-prog-i-ea-beadando/

Ha szükséges a Demo oldal FTP elérését elküldöm a Teams felületén, privát üzenetben, ehhez az elérhetőségem:

ghxxqa@hallgato.nje.hu