**WEB-programozás I.**

**Bevezetés**

A projektet pár hiányában két GitHub fiókkal dolgoztam ki. A fő fiók (SDani0210) hozta létre a repót, a második fiókot (Durrot) kollaborátorként hozzáadtam. A commitokat mindkét fiókkal végzem.

A GitHub repó elérhetősége: <https://github.com/SDani0210/webprog-hazi_NN8T44>

Az oldal elérési linkje: http://cosmicalpractice.nhely.hu

1. **Feladat**

A honlap kiinduló állománya az **index.html**, amelyet egységes szerkezet alapján építettem fel. Az oldal minden változatán megtalálható a fejléc (header), a vízszintes navigációs menüsor (nav), az oldalsáv (aside), a tartalmi rész (main – div-content) és a lábléc (footer). A menüsorban az aktuálisan kiválasztott menüpont külön stílussal van kiemelve. A láblécben a saját nevemet és Neptun kódomat tüntettem fel.

A lábléc elhelyezését úgy oldottam meg, hogy mindig az oldal alján maradjon, még akkor is, ha kevés tartalom jelenik meg. A stílusokat a **style.css** fájl tárolja, így az oldal minden része egységes megjelenést kapott.

**Táblázat menü (képek kellenek)**

A „Táblázat” menüpontban egy űrlap segítségével hoztam létre egy dinamikus CRUD-kezelő táblázatot JavaScript segítségével. Az adatok név, életkor, város és telefonszám szerint kerülnek rögzítésre és jelennek meg a táblázatban.

* A felületen minden mezőre építettem validációs szabályokat:
* A mezők nem lehetnek üresek
* A név és város mezők maximális hossza 30 karakter
* A telefonszám maximum 15 karakter
* Az életkor csak 0–120 közötti szám lehet

A táblázatban szereplő sorokra szerkesztési (✏️) és törlési (🗑️) lehetőség is van. Ha a felhasználó egy sort szerkeszt, a hozzáadás gomb ikonja pipára vált, és a korábban megadott adatokat frissíti, nem új sort hoz létre.

A felhasználók listája valós időben szűrhető bármely mező alapján (kivéve a gombok oszlopát), és az oszlopfejlécek kattinthatók, így a táblázat bármely oszlop szerint rendezhető növekvő vagy csökkenő sorrendbe.

Az alkalmazás kinézetét Bootstrap segítségével készítettem el, és színekkel valamint hover-hatással is jeleztem a rendezhető oszlopokat.

**HTML5 API**

Ebben a feladatrészben egyetlen HTML oldalon belül valósítottam meg a következő HTML5 API technológiákat.

* Web Storage: A localStorage használatával figyelem, hányszor töltöttem be az oldalt. A szám növekszik, és a látogató visszajelzést kap.
* Web Worker: Egy stopperóra valósult meg, amely tized- és ezredmásodperces pontossággal működik. A start gomb futtatja a háttérszálat, a stop leállítja, majd a start gomb újraindítja (nem nullázza).
* Geolocation API: A böngésző engedélykérés után kiírja a felhasználó szélességi és hosszúsági pozícióját. Hiba esetén egyértelmű visszajelzés történik.
* Drag and Drop API: Egy papír galacsin képet lehet behúzni egy szemetes kép ikonjába. A sikeres húzás után egy szöveges üzenet jelenik meg. A képek a projekt img/ mappájából származnak.
* Canvas API: Az egér mozgatásával rajzolni lehet a vásznon. A clear gombbal a felület törölhető.
* SVG: Egy radioaktivitás veszélyt jelző piktogram készült el három külön forgatott, íves, kifelé szélesedő fekete lapáttal, középen fekete körrel és sárga háttérrel.
* Server-Sent Events (SSE): PHP szerver telepítése után és szerver oldali futtatás során SSE keretein belül megcsináltam egy szimpla órát, ami óra:perc:másodperc értékben mutattja az aktuális időt.

A példák mindegyike interaktív, stílusos, és Bootstrap-pel egységes dizájnba illeszkedik. A teljes rész az html5.html fájlban valósult meg.

**ChartJS menü**

Ebben a részfeladatban egy interaktív diagram került megvalósításra a Chart.js könyvtár segítségével. A megoldás egy 5x10-es véletlenszerű számokat (0–100 között) tartalmazó táblázatot jelenít meg, ahol a felhasználó több sort is kijelölhet checkboxok segítségével. A kijelölt sorok adatai eltérő (random) színű vonalakkal jelennek meg a táblázat alatt található vonaldiagramon.

**Fő funkciók:**

* Táblázatot javascript kód generálja.
* Checkboxok minden sor elején, hogy ki tudjuk választani melyik sor értékeit szeretnénk látni.
* Egyidejüleg több sort is ki lehet jelölni.
* Az adatok valós idejű frissítése a diagramon a kijelölések alapján.
* Eltérő szín a különböző sorokhoz tartozó vonalakhoz.

**AJAX menü**

Ebben a feladatrészben egy teljes CRUD műveleteket támogató AJAX-alapú webalkalmazást készítettem, amely a megadott http://gamf.nhely.hu/ajax2/ API-val kommunikál. Az alkalmazás képes új rekordokat felvenni, meglévőket módosítani, törölni, valamint listázni az összes adatot vagy csak ID alapján listázni őket. Az oldalon aktív visszajelzések vannak minden művelet állapotáról.

**Megvalósított funkciók:**

* **Hozzáadás** (create): űrlapon megadott név, magasság és súly alapján új rekordot küld az API-nak. Sikeres létrehozás után a táblázat automatikusan frissül.
* **Lekérdezés – teljes lista** (read): külön gomb szolgál arra, hogy minden rekordot lekérjek a saját code-om alapján.
* **Lekérdezés – ID alapján**: egy másik mezőbe beírt ID alapján csak az adott rekordot jeleníti meg a táblázatban.
* **Módosítás** (update): a megadott ID-jű rekordot a beírt új értékekkel frissíti.
* **Törlés** (delete): egy adott ID-jű rekord végleges eltávolítása.

**Validáció:**

* Minden mező kötelező, és legfeljebb 30 karakter hosszú lehet. Hibás vagy hiányzó adat esetén az űrlap visszajelzést ad, és a művelet nem fut le.

**Visszajelzés:**

* A sikeres és sikertelen műveletekről színes visszajelzést jelenít meg (zöld: sikeres, piros: hiba).
* A szerver visszajelzésének (res == 1) ellenőrzésével garantált a helyes működés.

**Statisztika:**

* A listázott rekordokból számoltatok magasságösszeget, átlagot és legnagyobb értéket is, amely a táblázat alatt jelenik meg.

**Járműkártyák**

Ebben a részben egy objektumorientált JavaScript alkalmazást készítettem, amely különféle járműveket jelenít meg kártyás formában. A feladat célja az objektumorientált programozás alapfogalmainak – class, constructor, metódusok, öröklődés (extends, super) – gyakorlati alkalmazása volt.

**Használt technológiák**

* HTML, CSS (saját és Bootstrap)
* JavaScript osztályok (class alapú OOP)
* DOM-manipuláció: document.createElement(), appendChild()

**A megvalósítás leírása**

Létrehoztam egy Vehicle alaposztályt, amely tartalmazza a járművek közös tulajdonságait: márka, típus, lóerő, hajtáslánc, váltó típusa, származási ország és a jármű képe. Ez az osztály tartalmaz egy createCard() nevű metódust is, amely HTML-elemeket generálva létrehozza a jármű vizuális kártyáját. A kártyák kétoldalasak: az elején a jármű képe, a hátulján pedig a részletes adatai láthatók. Kattintásra a kártya megfordul.

Az extends kulcsszó segítségével három gyermedosztályt hoztam létre:

* Car – személyautók
* Truck – teherautók
* Motorcycle – motorkerékpárok

A járműkártyák egy JavaScript tömbben lettek felsorolva, példányosítva a megfelelő osztályból, majd a createCard() metódussal generálva kerültek ki a DOM-ba a vehicleContainer nevű divbe.

**Design**

A kártyák vizuális megjelenését CSS-sel valósítottam meg. A képek object-fit: cover beállítással csak részlegesen látszódnak a kártyán belül, nekem jobban tetszett így mintha kitöltötte volna az egész kártyát. A kártyák animált forgatással váltanak az elő- és hátoldal között.

1. **React alkalmazások**

***Számpárosító játék***

**Az alkalmazás célja:**

Ez az egyoldalas React-alkalmazás egy memória alapú számpárosító játékot valósít meg. A felhasználó célja, hogy megtalálja az összes szám párját a lehető legrövidebb idő alatt és a legkevesebb kattintással.

Az alkalmazás tulajdonságai:

* 20 kártya (10 számpár) 4×5 elrendezésben.
* Kattintásszámláló: minden kártyára történő kattintás növeli az értéket.
* Időmérő: a játék kezdetétől számítja az eltelt időt, és a játék befejeztével megjeleníti azt.
* Reset gomb: Újrakezdi a játékot.
* Egyszerű, reszponzív felület Bootstrap osztályokkal

**Technológiai háttér**

A játék React segítségével készült, a következő kulcselemekkel:

* React komponens (SzamparApp) – funkcionális komponens, ami egyéni állapotokat (useState) és életciklus-kezelést (useEffect) használ.
* useState – a kártyák állapotának, a felnyitott kártyák indexeinek, a kattintások és az időmérés kezelésére.
* useEffect – a játék logikáját vezérli, például a párok összehasonlítását vagy az időmérés leállítását.

**Főbb állapotváltozók:**

const [cards, setCards] = useState(generateCards); // kártyák tömbje

const [flipped, setFlipped] = useState([]); // aktuálisan felfordított 2 kártya indexei

const [disabled, setDisabled] = useState(false); // kattintás tiltása animáció idejére

const [clicks, setClicks] = useState(0); // kattintásszámláló

const [startTime, setStartTime] = useState(null); // játékidő indítása

const [elapsedTime, setElapsedTime] = useState(null); // játékidő leállítása

**Komponens struktúrája**

SzamparApp

├── <ResetButton /> // újrakezdéshez

├── <Stats /> // kattintás- és időkiírás

└── <CardGrid />

 └── <Card /> × 20

**Megjelenés**

* + A kártyák egy rácsos elrendezésben jelennek meg. A kiválasztott mező háttérszínt vált, és felfedi a számot. Ha a két felfordított kártya megegyezik, úgy maradnak, ha nem, egy rövid idő után visszafordulnak.
  + A játék végén a jobb oldalon egy halványszürke dobozban megjelennek a statisztikák:
* az eltelt idő (másodpercben),
* a kattintások száma.
  + A Reset gomb újrakezdi a játékot, visszaállít minden állapotot.

***Kockaszín***

**Az alkalmazás célja**

Ez az egyoldalas React-alkalmazás lehetővé teszi, hogy a felhasználó dobjon 1 vagy 2 kockával, és a dobott értékek összege alapján egy színt kapjon visszajelzésként. A színt a program dinamikusan hozzárendeli, és megjeleníti egy stilizált SVG dobókockán keresztül.

**Főbb jellemzők**

* **1 vagy 2 dobókockával való dobás** – véletlenszerű értékek generálása 1 és 6 között.
* **Összegzett érték megjelenítése** – a kockák összegének kiszámítása.
* **Szín hozzárendelés** – az összeg alapján egy szín jelenik meg.
* **SVG dobókockák megjelenítése** – a dobások számától függően 1 vagy 2 kocka színezve jelenik meg.
* **Egyszerű, látványos vizualizáció** – React komponensként renderelt SVG grafika.

**Technológiai háttér**

Az alkalmazás teljes mértékben React-tel készült, a következő elemekkel:

* useState – a dobások, összeg és szín állapotainak tárolására.
* createElement – JSX helyett közvetlen DOM leírás (mivel a feladat nem enged JSX-et).
* **Globális regisztráció (window.KockaszinApp)** – hogy a komponens dinamikusan váltható legyen az App szintjén.
* Bootstrap és inline stílusok a gombokhoz és az elrendezéshez.

**Főbb állapotváltozók:**

js

MásolásSzerkesztés

const [dobasok, setDobasok] = React.useState([]);

const [ossz, setOssz] = React.useState(null);

const [szin, setSzin] = React.useState(null);

**Komponens felépítése**

plaintext

MásolásSzerkesztés

KockaszinApp

├── <button> 1 kocka

├── <button> 2 kocka

├── <p> Dobott értékek és összeg

├── <p> Színnév kiírás

└── <svg> Kockák színezve

**Színek**

A 2-től 12-ig terjedő értékekhez hozzárendelt színek (pl. 2 = piros, 6 = lila, 12 = fekete) egy szín-összeg táblából kerülnek kiválasztásra, majd alkalmazásra az SVG-n.

**Használati példa**

* A felhasználó megnyomja a „2 kocka” gombot.
* A program dob: pl. 4 + 5 = 9
* A 9-es összeghez társított szín: türkiz
* A felületen megjelenik két türkiz színű SVG dobókocka, középen az összeggel, és kiírásra kerül: *„A dobás színe: türkiz”*