Bevezetés

A Záródolgozat Témája, milyen funkciók érhetőek el az oldalon?

Olyan témát szerettünk volna a záródolgozatunkban létrehozni ami valóban olyan problémára adhat megoldást ami könnyebbé teheti az emberek mindennapjait, ezért készítettük el a Szavazz.hu-t. Ez az oldal az online szavazás hiányát pótolhatná ami még nincs Magyarországon.

Hogy mit értünk ezalatt?

A felhasználó regisztráció után és a bejelentkezés folyamat végrehajtását követően tud szavazni a szimpatizáns pártra, a szavazás lejárata után pedig meg tudja tekinteni az eredményeket. Az oldalon lehetősége van a felhasználónak az aktuális híreket olvasni.

Miért pont politikai választás?

Az elmúlt egy évben miután nagykorúak lettünk és miután élhettünk a szavazat jogával kicsit jobban belemerültünk a politika világába. Egyszer csak azon gondolkodtunk, hogy Magyarországon miért nincs online választás, holott külföldön (pl.: Amerika, Egyesült Királyság, Franciaország, Hollandia, stb.) számos országban van online választás, valahol csak online tudnak az emberek szavazni. Ezért készítettük el az oldalunkat hogy ez Magyarországon is megváltozzon, modernizálódojon.

Kiknek szánjuk ezt a programot?

Minden 18. életévét betöltő Magyar állampolgárnak ajánljuk ezt az oldalt. Ez az oldal megkönnyíti a választásokat, mivel nem kell az embereknek elmenniük a választó körzetekbe, egyszerüen otthonról meg lehet oldani az. Az oldalunk a szavazatok gyarapodásában is segíthet, mivel Magyarország lakosságának több mint az 1/3-ad része nem szokott elmenni a választásokra különböző okok miatt és ebben is segíthet az oldalunk a kényelmes, egyszerű szavazat leadásával.

Fejlesztői Dokumentáció

Ezeket a szoftvereket használtuk a projekt elkészítéséhez:

Visual Studio Code:

A Visual Studio Code szerintünk rendkívül népszerű és felhasználóbarát kódszerkesztő, amelyet sok fejlesztő előnyben részesít rugalmassága és egyszerű kezelhetősége miatt. A projekt során végig ezt használtam, mivel rengeteg hasznos bővítménnyel és automatikus kódkiegészítéssel támogatja a munkát, különösen JavaScript nyelven. A jól testreszabható felület és a gyorsbillentyűk jelentősen meggyorsították a fejlesztési folyamatot.

XAMPP / MySQL:

Az XAMPP egy könnyen telepíthető és használható fejlesztői környezet, amely ideális választás helyi webszerver futtatásához. A projekt során ezt használtam, mert gyorsan beállítható és minden szükséges komponenst (Apache, MySQL, PHP) egy csomagban tartalmaz. Segítségével egyszerűen tudtam tesztelni és futtatni a webes alkalmazásomat anélkül, hogy külön-külön kellene telepítenem a szerverelemeket. Az iskolában ezt a szoftvert

GitHub / GitHub Desktop:

A GitHub és a GitHub Desktop nagyban megkönnyítették a verziókezelést a projekt során. A GitHub lehetővé tette, hogy biztonságosan tároljuk a kódunkat felhőben, valamint nyomon kövessem a változtatásokat. A GitHub Desktop pedig felhasználóbarát felületével gyors és egyszerű műveletvégzést biztosított anélkül, hogy parancssorral kellett volna dolgoznom. Így hatékonyabban tudtunk együtt dolgozni és visszakeresni a korábbi módosításokat.

Ezeket a programozási nyelveket használtuk:

Node.js:

A backend fejlesztéshez a Node.js-t választottuk, mert gyors, megbízható és tökéletesen illeszkedik a JavaScript alapú stack-hez. Lehetővé teszi, hogy ugyanazzal a nyelvvel dolgozzunk a frontend és a backend oldalon is, ami leegyszerűsítette a fejlesztési folyamatot. Emellett rengeteg elérhető csomag és modul áll rendelkezésre hozzá, amelyekkel könnyen bővíthető és testreszabható az alkalmazás működése.

React.js:

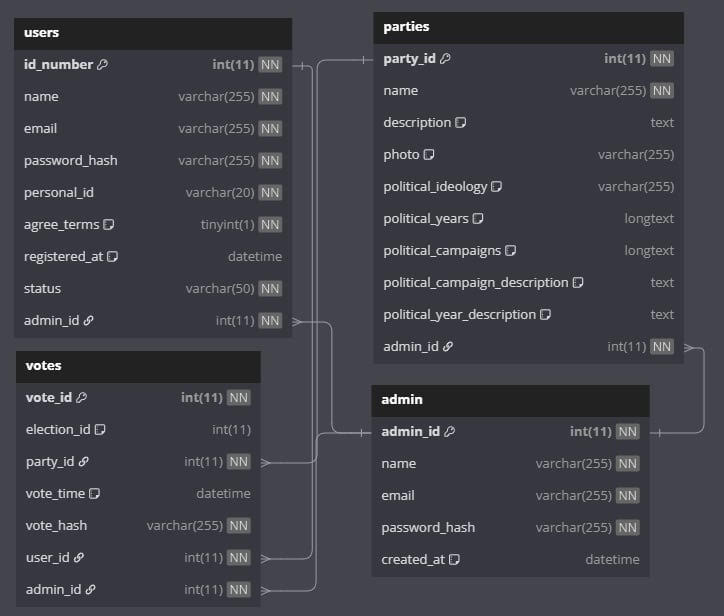
A frontend fejlesztéshez a React-et választottuk, mivel modern, komponens-alapú felépítése lehetővé teszi az átlátható és újrahasznosítható kód írását. Különösen előnyös volt, hogy kiválóan együttműködik a Node.js-sel, így a frontend és a backend között gördülékeny adatkapcsolat valósulhatott meg. Emellett a React gyors frissítési mechanizmusa és a széles körű közösségi támogatás is hozzájárult ahhoz, hogy ez legyen a legjobb választás a projekt számára.

CSS:

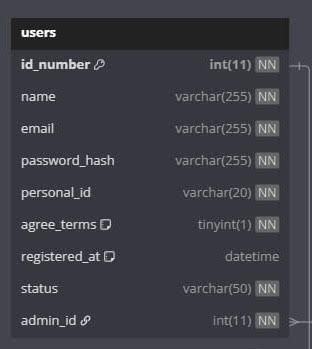
A projekt során a CSS-t használtuk az oldal megjelenésének kialakítására, mivel egyszerűen alkalmazható és rugalmas megoldást nyújt a dizájn testreszabására. Segítségével könnyedén formázhattuk az elemeket, beállíthattuk az elrendezést, színeket, sőt, animációkat is hozzáadhattunk. A reszponzivitás megvalósítása is egyszerű volt vele, így az oldal minden eszközön – legyen az mobil, tablet vagy asztali gép – megfelelően jelenik meg. A CSS lehetővé tette, hogy az oldal kinézete letisztult és felhasználóbarát legyen, anélkül, hogy bonyolult eszközöket kellett volna használnunk.

MySQL:

Az adatok tárolásához a MySQL adatbázis-kezelőt választottuk, mivel megbízható, könnyen kezelhető és jól integrálható a Node.js alapú backenddel. Lehetővé tette a strukturált adatkezelést, a felhasználók, szavazatok és egyéb fontos információk biztonságos tárolását. A lekérdezések gyorsan és hatékonyan futottak, emellett az adatbázis szerkezetét is könnyen tudtuk módosítani a projekt igényei szerint. Egyszerűsége és stabilitása miatt ideális választás volt a projekt számára.

Az Adatbázis:  


A “users” tábla:



A „users” tábla az oldal felhasználóinak adatait tárolja, ideértve a regisztrációhoz és bejelentkezéshez szükséges információkat, valamint a felhasználói jogosultságokat is.

**id\_number**: A felhasználó azonosítója, elsődleges kulcs (PRIMARY KEY), automatikusan növekvő egész szám (INT), amely egyedi módon azonosítja a felhasználót az adatbázisban.

**name**: A felhasználó teljes neve, VARCHAR(255) típusú mező, kötelezően kitöltendő (NOT NULL).

**email**: A felhasználó e-mail címe, amely egyben a bejelentkezéshez is szükséges. VARCHAR(255) típusú, szintén nem lehet üres. Egyediség biztosítása érdekében gyakran UNIQUE kulcsként is meg van jelölve.

**password\_hash**: A felhasználó jelszavának titkosított változata. A jelszavakat nem nyílt szövegként, hanem bcrypt algoritmussal titkosítva tároljuk, ezzel biztosítva a maximális adatvédelmet.

**personal\_id**: A felhasználó adószáma, amely egy VARCHAR(20) típusú mező. Segítségével egyedi módon azonosítható a felhasználó.

**agree\_terms**: Egy TINYINT(1) típusú mező, amely jelzi, hogy a felhasználó elfogadta-e az adatkezelési szabályzatot vagy a felhasználási feltételeket. Az érték 1, ha elfogadta, és 0, ha nem.

**registered\_at**: A regisztráció dátuma és időpontja DATETIME formátumban. Ez alapján nyomon követhető, mikor történt a regisztráció, akár év-hónap-nap óra:perc:másodperc pontossággal.

**status**: A felhasználó státusza, például lehet „aktív”, „felfüggesztett” vagy „törölt”. VARCHAR(50) típusú mezőként szerepel az adatbázisban.

**admin\_id**: Egy idegen kulcs (FOREIGN KEY) jellegű mező INT(11) típussal, amely arra szolgálhat, hogy kapcsolatot teremtsen egy másik adminisztratív felhasználót tartalmazó táblával. Segítségével naplózható, ha például egy admin hozott létre egy felhasználót.

A “parties” tábla:



A „parties” tábla a választáson részt vevő politikai pártok adatait tartalmazza. Ez a tábla szolgál alapul a pártok listázásához, megjelenítéséhez és adminisztratív kezeléséhez az oldalon.

**party\_id**: A párt egyedi azonosítója, elsődleges kulcs (PRIMARY KEY), automatikusan növekvő egész szám (INT). Segítségével különböztetjük meg az egyes pártokat az adatbázisban.

**name**: A párt neve, amely VARCHAR(255) típusú mezőként van eltárolva. Ez a megnevezés jelenik meg a felhasználók számára is, így kötelezően kitöltendő adat.

**description**: A párt rövid leírása, TEXT típusú mező, amely bemutatja a párt általános célkitűzéseit, jellemzőit vagy küldetését. Ez jelenhet meg például a szavazóoldalon tájékoztatásként.

**photo**: A párthoz tartozó kép vagy logó fájlneve/útvonala, amely VARCHAR(255) típusban van tárolva. Ez biztosítja a pártok vizuális megjelenítését a felületen.

**political\_ideology**: A párt politikai irányultsága (pl. konzervatív, liberális, zöld, stb.), VARCHAR(255) típusú mezőként. Segít a választóknak azonosítani a párt ideológiai irányvonalát.

**political\_years**: LONGTEXT típusú mező, amely az adott párt működésének történetét tartalmazza évek szerint – akár több évnyi információ is elfér benne, például fontos események, választási eredmények.

**political\_campaigns**: Szintén LONGTEXT típusú mező, amely részletesen tartalmazza a párt által valaha indított kampányokat, programokat, kezdeményezéseket.

**political\_campaign\_description**: A kampányok összefoglaló, tömör leírása TEXT mezőként. Ez lehetővé teszi, hogy a kampányok rövid ismertetése külön is megjelenjen például előnézetként.

**political\_year\_description**: Az évek szerinti tevékenységek rövidebb, összefoglaló leírása. Ez a mező TEXT típusú, és a részletes évleírás (political\_years) mellett kiegészítő információkat tartalmazhat.

**admin\_id**: Egy INT(11) típusú mező, amely azt tárolja, hogy melyik adminisztrátor hozta létre vagy módosította a párt adatait. Általában idegen kulcsként kapcsolódik az „admins” vagy „users” táblához, és lehetőséget ad az adminisztratív változások naplózására.

A “votes” tábla:



A „votes” tábla a leadott szavazatok adatait tárolja. Ez a tábla biztosítja a szavazatok rögzítését, nyilvántartását és az eredmények későbbi feldolgozását. A felépítése lehetővé teszi a választások, pártok és felhasználók összekapcsolását, miközben a titkosítás révén biztosítja a szavazók anonimitását.

**vote\_id**: A szavazat egyedi azonosítója, INT(11) típusú mezőként van eltárolva. Elsődleges kulcs (PRIMARY KEY), amely minden egyes szavazatot külön nyilvántart. A rendszer biztonsága érdekében ez az érték titkosítva van, így nem visszakövethető, hogy melyik felhasználó adta le a szavazatot.

**election\_id:** Az adott választás azonosítója, INT(11) típusú mező. Lehetővé teszi, hogy egy adatbázis több választást is kezeljen egyszerre. Külső kulcsként kapcsolódhat egy „elections” nevű táblához, ha az elérhető.

**party\_id:** A kiválasztott politikai párt azonosítója, INT(11) típusú mező. Ez a mező kapcsolódik a „parties” táblához, és jelzi, hogy a szavazat melyik pártra érkezett.

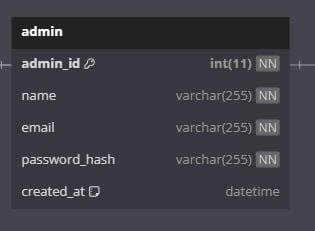
**vote\_time:** A szavazat leadásának pontos időpontja, DATETIME típusú mezőként eltárolva. Ezzel nyomon követhető, mikor történt a szavazás, például a választási időszakok érvényességének ellenőrzése során.

**vote\_hash:** A szavazat titkosított lenyomata, amely VARCHAR(255) típusú mező. A hash biztosítja, hogy a szavazat egyedi és sérthetetlen legyen, anélkül, hogy visszafejthető lenne a választás eredeti tartalma.

**user\_id:** A szavazó felhasználó azonosítója, INT(11) típusú mező. Ez az érték szintén titkosítva van eltárolva, hogy a rendszer garantálni tudja a választók anonimitását, és ne lehessen visszakeresni, hogy ki melyik pártra szavazott.

**admin\_id:** Egy INT(11) típusú mező, amely azt rögzíti, hogy melyik adminisztrátor hozta létre vagy érvényesítette a szavazatot. Idegen kulcsként kapcsolódhat az „admins” vagy „users” táblához. Segítségével naplózható, ha manuális bejegyzés vagy ellenőrzés történt adminisztrátori oldalról.

Az “admin” tábla:



Az „admin” tábla az oldal adminisztrátori felhasználóinak adatait tárolja. Ezek az adminok rendelkeznek a jogosultsággal arra, hogy kezeljék a szavazásokat, módosítsák a pártokat, szabályozzák az oldal működését és felügyeljék a rendszer működését. A tábla biztonságos kialakítása lehetővé teszi a hitelesítést és a tevékenységek naplózását.

**admin\_id:** Az adminisztrátor egyedi azonosítója, INT(11) típusú mező, amely elsődleges kulcsként (PRIMARY KEY) szolgál. Segítségével különíthetők el az adminisztrátorok, és naplózható, hogy mely admin végzett el adott módosítást a rendszerben.

**name:** Az adminisztrátor neve, VARCHAR(255) típusú mezőként van eltárolva. Ez a név jelenhet meg például az adminfelületen vagy naplózási eseményekben.

**email:** Az adminisztrátor e-mail címe, VARCHAR(255) típusú mező. Ez az adat az azonosításhoz és belépéshez szükséges, valamint elérhetőségi célt is szolgál.

**password\_hash:** A jelszó titkosított verziója, VARCHAR(255) típusban eltárolva. A jelszavak biztonságos kezelését a rendszer bcrypt titkosítással végzi, így azok visszafejthetetlenek és teljes védelmet nyújtanak az admin fiókok számára.

**created\_at**: Az admin fiók létrehozásának időpontja, DATETIME típusú mezőként tárolva. Segítségével követhető, hogy mikor lett az admin regisztrálva a rendszerben, és támogatja a jogosultságok naprakész kezelését.

Algoritmusok és API végpontok

A **/register** útvonalú végpont esetében azt szerettem volna elérni, hogy egy teljes értékű felhasználói regisztráció történjen meg a szerveren, ahol a felhasználó által megadott adatok megfelelően ellenőrizve, titkosítva, illetve biztonságosan letárolva kerülnek be az adatbázisba.

// Regisztrációs endpoint

app.post('/register', (req, res) => {

  const { name, email, pass, personal\_id, agreeTerm } = req.body;

  if (!name || !email || !pass || !personal\_id || agreeTerm === undefined) {

    return res.status(400).json({ message: 'Minden mezőt ki kell tölteni!' });

  }

  const checkQuery = 'SELECT \* FROM users WHERE email = ?';

  db.query(checkQuery, [email], (err, result) => {

    if (err) {

      console.error('Hiba a felhasználó ellenőrzésekor:', err);

      return res.status(500).json({ message: 'Belső hiba történt' });

    }

    if (result.length > 0) {

      return res.status(400).json({ message: 'Ez az email már regisztrálva van' });

    }

    const encryptedEmail = encryptData(email);

    const encryptedName = encryptData(name);

    const encryptedPersonalId = encryptData(personal\_id);

    const insertQuery = `INSERT INTO users (name, email, password\_hash, personal\_id, agree\_terms, status)

                         VALUES (?, ?, ?, ?, ?, 'active')`;

    bcrypt.hash(pass, 10, (err, hashedPassword) => {

      if (err) {

        console.error('Hiba a jelszó hash-elésekor:', err);

        return res.status(500).json({ message: 'Belső hiba történt' });

      }

db.query(insertQuery, [encryptedName, encryptedEmail, hashedPassword, encryptedPersonalId, agreeTerm ? 1 : 0], (err, result) => {

        if (err) {

          console.error('Hiba a felhasználó hozzáadása közben:', err);

          return res.status(500).json({ message: 'Belső hiba történt' });

        }

        res.status(200).json({ message: 'Sikeres regisztráció!' });

      });

    });

  });

});

Ez a végpont akkor aktiválódik, amikor a felhasználó a regisztrációs űrlapot beküldi. A req.body segítségével kiszedi az űrlapmezők értékeit (név, email, jelszó, személyi azonosító és a feltételek elfogadása). Először megnézi, hogy minden mező ki van-e töltve. Ha valamelyik hiányzik, azonnal visszajelzést ad a kliensnek egy hibaüzenettel, és leáll a feldolgozás.

Ezután egy SQL lekérdezés segítségével megnézi, hogy a megadott email cím szerepel-e már az adatbázisban. Ha igen, újabb hibaüzenetet küld, hogy az email cím már regisztrálva van. Ha nem szerepel, akkor titkosítja a felhasználó érzékeny adatait, tehát az emailt, a nevet és a személyi azonosítót az encryptData() nevű függvénnyel – ez biztonságtechnikai okokból fontos.

A jelszó soha nem kerül nyersen az adatbázisba, hanem a bcrypt.hash() segítségével előbb egy egyirányú, biztonságos hashelési folyamaton megy keresztül. A hash-elt jelszó így már visszafejthetetlen, még a rendszeradminisztrátor számára is. Ha ez sikeres, akkor egy SQL INSERT lekérdezéssel elmenti a felhasználó adatait az adatbázisba, ahol a státusza automatikusan 'active' lesz. A agreeTerm mezőt 1-es vagy 0-ás értékkel menti attól függően, hogy a felhasználó elfogadta-e a feltételeket.

Végül, ha minden jól ment, a szerver egy "Sikeres regisztráció!" üzenetet küld vissza a kliensnek, ami azt jelzi, hogy a folyamat sikeresen lezárult.

A **/login** útvonalú végpont esetében azt szerettem volna elérni, hogy a rendszer képes legyen biztonságosan és pontosan kezelni a felhasználók (és adminok) bejelentkezését, az adatok titkosításának figyelembevételével.

// Bejelentkezés végpont

app.post('/login', (req, res) => {

  const { email, password } = req.body;

  // Először a felhasználókat ellenőrizzük

  db.query('SELECT \* FROM users WHERE email = ?', [encryptData(email)], (err, results) => {

    if (err) {

      console.error('Hiba történt a users lekérdezésekor:', err);

      return res.status(500).json({ message: 'Belső hiba történt!' });

    }

    if (results.length > 0) {

      const user = results[0];

      const decryptedEmail = decryptData(user.email);

      if (decryptedEmail !== email) {

        return res.status(400).json({ message: 'Helytelen email vagy jelszó' });

      }

      bcrypt.compare(password, user.password\_hash, (err, isMatch) => {

        if (err) {

          console.error('Hiba a jelszó ellenőrzésekor:', err);

          return res.status(500).json({ message: 'Belső hiba történt' });

        }

        if (!isMatch) {

          return res.status(400).json({ message: 'Helytelen email vagy jelszó' });

        }

        return res.status(200).json({

          message: 'Sikeres bejelentkezés',

          user: {

            id: user.id\_number,

            name: decryptData(user.name),

            email: decryptedEmail,

            isAdmin: false,

          },

        });

      });

    } else {

      db.query('SELECT \* FROM admin WHERE email = ?', [email], (err, results) => {

        if (err) {

          console.error('Hiba történt az admin lekérdezésekor:', err);

          return res.status(500).json({ message: 'Belső hiba történt!' });

        }

        if (results.length === 0) {

          return res.status(400).json({ message: 'Helytelen email vagy jelszó' });

        }

        const admin = results[0];

        if (admin.email !== email) {

          return res.status(400).json({ message: 'Helytelen email vagy jelszó' });

        }

        bcrypt.compare(password, admin.password\_hash, (err, isMatch) => {

          if (err) {

            console.error('Hiba a jelszó ellenőrzésekor:', err);

            return res.status(500).json({ message: 'Belső hiba történt' });

          }

          if (!isMatch) {

            return res.status(400).json({ message: 'Helytelen email vagy jelszó' });

          }

          return res.status(200).json({

            message: 'Sikeres bejelentkezés',

            user: {

              id: admin.admin\_id,

              name: admin.name,

              email: admin.email,

              isAdmin: true,

            },

          });

        });

      });

    }

  });

});

Ez a végpont akkor aktiválódik, amikor a felhasználó a bejelentkezési űrlapot beküldi. A req.body segítségével kiolvassa az email-címet és a jelszót. Először az adatbázis users táblájában keres egy olyan sort, ahol az email-cím azonos az elküldött (és a szerveroldalon újra titkosított) email-címmel. Az összehasonlítást a encryptData() függvény végzi el.

Ha talál ilyet, az adatokat lekéri, és az email-címet visszafejti a decryptData() függvénnyel, majd újra összehasonlítja az eredeti, beküldött emaillel. Ez egy extra biztonsági ellenőrzés, hogy a titkosítás és dekódolás során semmilyen torzulás nem történt. Ha az email nem egyezik, vagy nincs egyezés a hash-elt jelszóval (amit a bcrypt.compare() végez), a válasz az, hogy a bejelentkezés sikertelen.

Ha viszont a users táblában nincs egyező email, akkor a rendszer megpróbálja az admin táblában keresni a bejelentkező felhasználót. Itt nincs titkosított email, ezért egyszerű SQL lekérdezéssel és összehasonlítással történik az admin azonosítása. Az admin jelszó ellenőrzése szintén bcrypt.compare() függvénnyel történik.

Sikeres bejelentkezés esetén a rendszer egy válaszobjektumban visszaküldi a felhasználó vagy admin nevét, email-címét, azonosítóját, illetve egy isAdmin logikai értéket, amely megkülönbözteti az adminokat a rendes felhasználóktól.

A **/voting** útvonalú végpont esetében azt szerettem volna elérni, hogy a felhasználó egy választás során csak egyszer adhassa le a szavazatát, valamint a rendszer ezt biztonságosan, megfelelő ellenőrzésekkel és visszajelzéssel kezelje.

// Szavazat leadása

app.post('/voting', (req, res) => {

  const { election\_id, party\_id, vote\_hash, user\_id } = req.body;

  if (!election\_id || !party\_id || !vote\_hash || !user\_id) {

    return res.status(400).json({ message: 'Minden mezőt ki kell tölteni!' });

  }

  const checkVoteQuery = 'SELECT \* FROM votes WHERE election\_id = ? AND user\_id = ?';

  db.query(checkVoteQuery, [election\_id, user\_id], (err, result) => {

    if (err) {

      console.error('Hiba történt a szavazat ellenőrzésekor:', err);

      return res.status(500).json({ message: 'Hiba történt a szavazat ellenőrzésekor!' });

    }

    if (result.length > 0) {

      return res.status(400).json({ message: 'Már leadtad a szavazatot!' });

    }

    const query = 'INSERT INTO votes (election\_id, party\_id, vote\_hash, user\_id) VALUES (?, ?, ?, ?)';

    db.query(query, [election\_id, party\_id, vote\_hash, user\_id], (err, result) => {

      if (err) {

        console.error('Hiba történt a szavazat mentésekor:', err);

        return res.status(500).json({ message: 'Hiba történt a szavazat leadásakor!' });

      }

      res.status(200).json({ message: 'Sikeresen leadta a szavazatot!' });

    });

  });

});

Ez a végpont akkor aktiválódik, amikor a felhasználó a szavazási űrlapot beküldi. A szerver a req.body segítségével beolvassa a election\_id, party\_id, vote\_hash és user\_id értékeket. Először megvizsgálja, hogy ezek közül bármelyik hiányzik-e. Ha igen, azonnal visszatér egy 400-as hibakóddal, és jelzi, hogy minden mezőt ki kell tölteni.

Ezután az adatbázisban lefuttat egy SQL lekérdezést, amely azt vizsgálja, hogy az adott felhasználó (user\_id) az adott választásban (election\_id) már szerepel-e a votes táblában. Ez megakadályozza, hogy valaki többször is leadja a szavazatát ugyanabban a választásban. Ha talál ilyen rekordot, akkor 400-as hibakóddal válaszol, hogy a szavazat már le van adva.

Amennyiben még nem történt szavazás, a végpont lefuttat egy új INSERT SQL lekérdezést, amellyel rögzíti a szavazatot az adatbázis votes táblájába. A vote\_hash mező opcionálisan a szavazat titkosított változatát is tartalmazhatja, így biztosítva a szavazat védelmét vagy ellenőrizhetőségét.

Sikeres mentés után a szerver 200-as státusszal és pozitív visszajelzéssel válaszol, miszerint a szavazat leadása sikeres volt.

A **/election-results** végpont esetében azt szerettem volna elérni, hogy egy olyan lekérdezést hajtson végre az adatbázisból, amely összesíti az egyes pártokra leadott szavazatok számát, és ezt csökkenő sorrendben, azaz a legtöbb szavazatot kapott párttól a legkevesebbig jeleníti meg.

app.get('/election-results', (req, res) => {

  const  query = `

    SELECT p.party\_id, p.name AS party\_name, COUNT(v.vote\_id) AS totalVotes

    FROM parties p

    LEFT JOIN votes v ON p.party\_id = v.party\_id

    GROUP BY p.party\_id, p.name

    ORDER BY totalVotes DESC;

`;

  db.query(query, (err, results) => {

    if (err) {

      console.error(err);

      return res.status(500).json({ error: 'Hiba történt a pártokra leadott szavazatok lekérésekor.' });

    }

    return res.json(results.map((row) => ({

      party\_id: row.party\_id,

      party: row.party\_name,

      votes: row.totalVotes

    })));

  });

});

Ez az API-végpont egy GET típusú HTTP-kérést vár, és célja a szavazás jelenlegi eredményeinek lekérdezése. A lekérdezés az adatbázis parties és votes tábláit kapcsolja össze egy LEFT JOIN segítségével, ami azt jelenti, hogy azok a pártok is meg fognak jelenni az eredménylistában, amelyekre még egyetlen szavazat sem érkezett. Ez különösen fontos, hogy a szavazáson részt vevő összes párt reprezentálva legyen az eredménylistában, akkor is, ha nulla szavazatot kaptak.

A COUNT(v.vote\_id) segítségével a rendszer megszámolja, hogy hány szavazat kapcsolódik az adott párthoz. A GROUP BY kulcsszóval csoportosítja a rekordokat pártonként, az ORDER BY totalVotes DESC utasítás pedig biztosítja, hogy az eredmények a legtöbb szavazatot kapott párttal kezdődjenek.

A db.query() metódus futtatja le az SQL-lekérdezést, és ha nem történik hiba, akkor a visszakapott eredményeket JSON-formátumban küldi vissza a kliensnek. Minden rekord tartalmazza a párt azonosítóját (party\_id), a párt nevét (party), és az adott pártra leadott szavazatok számát (votes).

Ez a végpont tehát egy olyan, valós időben frissülő eredménylistát biztosít, amely a szavazás állását mutatja, és felhasználható például a felhasználói felületen grafikonként vagy listaként való megjelenítéshez.

A **/api/users/:id** útvonalon elérhető PUT típusú API-végpontot arra hoztam létre, hogy lehetőséget biztosítson a felhasználók adatainak szerveroldali frissítésére, egyedi azonosító (id\_number) alapján.

// Felhasználói adatok frissítése

app.put('/api/users/:id', (req, res) => {

  const { name, email, personal\_id, status } = req.body;

  const userId = req.params.id;

  const query = 'UPDATE users SET name = ?, email = ?, personal\_id = ?, status = ? WHERE id\_number = ?';

  db.query(query, [name, email, personal\_id, status, userId], (err, result) => {

    if (err) {

      res.status(500).send('Error updating user');

      return;

    }

    res.send('User updated');

  });

});

Ez a végpont egy PUT metódust használ, ami REST szabvány szerint arra való, hogy egy adott erőforrást — jelen esetben egy konkrét felhasználó adatait — teljes egészében frissítsen. Az útvonal :id paramétere az adatbázisban szereplő felhasználó egyedi azonosítója (id\_number), amelyet az URL-ből nyer ki a rendszer a req.params.id segítségével.

A req.body objektumból kinyert változók (name, email, personal\_id, status) tartalmazzák a kliens által beküldött frissített adatokat. Ezek az értékek bekerülnek az SQL UPDATE utasításba, amely a users táblában megkeresi azt a sort, ahol az id\_number megegyezik a megadott userId-val, majd frissíti a többi mezőt a beküldött értékekkel.

A db.query() metódus futtatja az SQL-parancsot. Ha a lekérdezés végrehajtása során hiba történik (pl. az adatbázis elérhetetlenné válik, vagy érvénytelen mezőket adtunk meg), akkor a szerver 500 Internal Server Error státuszkódot küld vissza. Ha a frissítés sikeres, egy egyszerű válasz ('User updated') érkezik vissza a klienshez.

Ez a végpont tehát elsősorban adminisztratív célokat szolgálhat, például egy admin felület segítségével a felhasználók adatainak módosítására vagy státuszának (active, inactive stb.) kezelésére.

Ebben a részben egy olyan szerveroldali megoldást hoztam létre, amely lehetővé teszi, hogy az adminisztrátor manuálisan beállíthassa a visszaszámlálás dátumát egy POST típusú végponton keresztül, illetve egy időzített rendszerellenőrzés automatikusan visszaállítsa azt nullára, ha a dátum lejárt.

app.post('/api/date-plus', (req, res) => {

 const { countdownDate } = req.body;

  if (!countdownDate) {

    return res.status(400).json({ error: 'A név és a leírás kötelező mezők!' });

  }

  const date = new Date(countdownDate);

  if (isNaN(date.getTime())) {

    return res.status(400).json({ error: 'Invalid date format' });

  }

  db.query('UPDATE settings SET countdown\_date = ? WHERE id = 1', [countdownDate], (err, results) => {

    if (err) {

      console.error('Database error:', err);

      return res.status(500).json({ error: 'Database error' });

    }

    res.json({ message: 'Countdown date updated successfully' });

  });

});

Ez a /api/date-plus végpont tehát arra szolgál, hogy a settings nevű adatbázistábla countdown\_date mezőjét frissítsük. Az adminisztrátor egy dátumot küld a kliensoldalról (req.body.countdownDate), amelyet a szerver előbb validál:

Ellenőrzi, hogy egyáltalán meg lett-e adva a dátum.

Megpróbálja a countdownDate értéket Date objektummá konvertálni, és ha az érvénytelen, hibát dob.

Sikeres ellenőrzés után egy SQL UPDATE lekérdezéssel frissíti a countdown\_date értékét, és visszaküld egy megerősítő üzenetet.

const checkAndResetCountdown = () => {

  const now = new Date();

  db.query('SELECT countdown\_date FROM settings WHERE id = 1', (err, results) => {

    if (err) {

      console.error('Database error:', err);

      return;

    }

    if (results.length > 0) {

      const countdownDate = new Date(results[0].countdown\_date);

      if (countdownDate <= now) {

        db.query('UPDATE settings SET countdown\_date = NULL WHERE id = 1', (updateErr) => {

          if (updateErr) {

            console.error('Database update error:', updateErr);

          } else {

            console.log('Countdown date reset to NULL.');

          }

        });

      }

    }

  });

};

setInterval(checkAndResetCountdown, 1 \* 1000);

Ez a checkAndResetCountdown nevű függvény egy setInterval segítségével 1 másodpercenként lefut, és lekérdezi a jelenleg beállított countdown\_date értékét. Ha az aktuális dátum meghaladja ezt az értéket, akkor automatikusan NULL-ra állítja a mezőt. Ez megakadályozza, hogy a lejárt visszaszámlálási idő továbbra is aktívként szerepeljen az adatbázisban.

Ez a funkcionalitás biztosítja, hogy a választási időzítő ne maradjon érvényben a határidő lejárta után, és ne befolyásolja a további adminisztratív vagy felhasználói műveleteket.

A **/api/reset-countdown** útvonalon elérhető POST típusú API-végpontot arra hoztam létre, hogy lehetőséget biztosítson a választási határidő (countdown\_date) szerveroldali nullázására az id = 1 azonosítójú bejegyzés esetén.

app.post('/api/reset-countdown', (req, res) => {

  db.query('UPDATE settings SET countdown\_date = "0000-00-00 00:00:00" WHERE id = 1', (err, results) => {

    if (err) {

      console.error('Database error:', err);

      return res.status(500).json({ error: 'Database error' });

    }

    res.json({ message: 'Countdown date reset successfully' });

  });

});

Ez a végpont egy POST metódust használ, amely a REST szabvány szerint olyan műveletekre alkalmas, ahol módosítjuk a szerver állapotát — jelen esetben az adatbázisban tárolt countdown\_date értékét. Az útvonalon nem kell adatot beküldeni a kérés törzsében (body), mivel a frissítendő érték rögzített: "0000-00-00 00:00:00".

A db.query() metódus segítségével fut le az SQL UPDATE utasítás, amely a settings táblában a countdown\_date mezőt nullázza, de csak annál a rekordnál, amelynek id értéke 1. Ha a lekérdezés végrehajtása során hiba történik (pl. az adatbázis elérhetetlenné válik), akkor a szerver 500 Internal Server Error státuszkódot küld vissza. Ha a frissítés sikeres, akkor egy egyszerű válasz ({ message: 'Countdown date reset successfully' }) érkezik vissza a klienshez.

A **/api/reset-all** útvonalon elérhető POST típusú API-végpontot arra hoztam létre, hogy lehetőséget biztosítson a rendszer teljes alaphelyzetbe állítására: a szavazás határidejének nullázására, a szavazatok törlésére, valamint az AUTO\_INCREMENT visszaállítására a votes táblában.

app.post('/api/reset-all', (req, res) => {

  db.beginTransaction((err) => {

    if (err) {

      console.error('Transaction error:', err);

      return res.status(500).json({ error: 'Transaction error' });

    }

    // 1. Az időzítő lenullázása

    db.query('UPDATE settings SET countdown\_date = "0000-00-00 00:00:00" WHERE id = 1', (updateErr) => {

      if (updateErr) {

        return db.rollback(() => {

          console.error('Database update error:', updateErr);

          res.status(500).json({ error: 'Failed to reset countdown date' });

        });

      }

      // 2. Szavazatok törlése

      db.query('DELETE FROM votes', (deleteErr) => {

        if (deleteErr) {

          return db.rollback(() => {

            console.error('Vote deletion error:', deleteErr);

            res.status(500).json({ error: 'Failed to delete votes' });

          });

        }

        // 3. Auto-increment visszaállítása

        db.query('ALTER TABLE votes AUTO\_INCREMENT = 1;', (alterErr) => {

          if (alterErr) {

            return db.rollback(() => {

              console.error('Auto-increment reset error:', alterErr);

              res.status(500).json({ error: 'Failed to reset auto-increment' });

            });

          }

          // Ha minden sikerült, akkor commit

          db.commit((commitErr) => {

            if (commitErr) {

              return db.rollback(() => {

                console.error('Commit error:', commitErr);

                res.status(500).json({ error: 'Transaction commit failed' });

              });

            }

            res.json({ message: 'Reset successful: countdown reset, votes deleted, and auto-increment reset' });

          });

        });

      });

    });

  });

});

Ez a végpont egy POST metódust használ, mivel komplex adatbázisműveletet hajt végre, ami módosítja a szerver állapotát. A tranzakció (db.beginTransaction) biztosítja, hogy a három lépés (időzítő nullázása, szavazatok törlése, auto-increment visszaállítása) vagy mind együtt történjen meg, vagy semmi ne változzon, ha bármelyik lépés hibát okoz.

Az első lekérdezés az UPDATE settings SQL-paranccsal a countdown\_date mezőt állítja 0000-00-00 00:00:00 értékre, ami jelzi, hogy nincs aktív szavazási időszak.

Ezután a DELETE FROM votes parancs törli az összes eddig leadott szavazatot a votes táblából.

Végül az ALTER TABLE votes AUTO\_INCREMENT = 1 utasítás visszaállítja az AUTO\_INCREMENT értéket, így a következő szavazat azonosítója ismét 1 lesz.

Ha bármelyik lépés során hiba lép fel, akkor a db.rollback() segítségével minden változtatás visszavonásra kerül. Ha minden sikeres, a db.commit() lezárja a tranzakciót, és a kliens egy sikeres választ kap:

{ message: 'Reset successful: countdown reset, votes deleted, and auto-increment reset' }

A **/api/users/:id\_number/change-password** útvonalon elérhető PUT típusú API-végpontot arra hoztam létre, hogy lehetőséget biztosítson a felhasználóknak saját jelszavuk biztonságos megváltoztatására, a meglévő jelszó ismeretében.

// Jelszó változtatás endpoint

app.put('/api/users/:id\_number/change-password', (req, res) => {

  const { id\_number } = req.params;

  const { currentPassword, newPassword } = req.body;

  // Ellenőrizzük, hogy mindkét jelszó meg van adva

  if (!currentPassword || !newPassword) {

    return res.status(400).json({ message: 'A jelenlegi és az új jelszó megadása kötelező.' });

  }

  // Először lekérjük a felhasználó jelenlegi jelszavát az adatbázisból

  const query = 'SELECT password\_hash FROM users WHERE id\_number = ?';

  db.query(query, [id\_number], (err, result) => {

    if (err) {

      return res.status(500).json({ message: 'Hiba történt a felhasználó keresésekor.' });

    }

    if (result.length === 0) {

      return res.status(404).json({ message: 'Felhasználó nem található.' });

    }

    // A jelenlegi jelszó összehasonlítása

    const storedPasswordHash = result[0].password\_hash;

    bcrypt.compare(currentPassword, storedPasswordHash, (err, isMatch) => {

      if (err) {

        return res.status(500).json({ message: 'Hiba történt a jelszó ellenőrzésekor.' });

      }

      if (!isMatch) {

        return res.status(400).json({ message: 'A jelenlegi jelszó nem helyes.' });

      }

      // Ha a jelenlegi jelszó helyes, titkosítjuk az új jelszót

      bcrypt.hash(newPassword, 10, (err, hashedPassword) => {

        if (err) {

          return res.status(500).json({ message: 'Hiba történt az új jelszó hashelésekor.' });

        }

        // Az új jelszó frissítése az adatbázisban

        const updateQuery = 'UPDATE users SET password\_hash = ? WHERE id\_number = ?';

        db.query(updateQuery, [hashedPassword, id\_number], (err, result) => {

          if (err) {

            return res.status(500).json({ message: 'Hiba történt a jelszó módosítása során.' });

          }

          if (result.affectedRows > 0) {

            res.status(200).json({ message: 'A jelszó sikeresen megváltozott.' });

          } else {

            res.status(404).json({ message: 'Felhasználó nem található.' });

          }

        });

      });

    });

  });

});

Ez a végpont egy PUT metódust használ, mivel REST elvek szerint ez az erőforrások (jelen esetben a jelszó) frissítésére szolgál. Az útvonal paraméterként (:id\_number) várja a felhasználó egyedi azonosítóját, amely alapján kikeresi az adatbázisból a jelenlegi jelszót.

A kliens által küldött currentPassword és newPassword mezők ellenőrzés után a következő lépések történnek:

A rendszer lekéri az adott felhasználó jelenlegi jelszó-hashét az adatbázisból.

A bcrypt.compare() függvény összehasonlítja a megadott jelenlegi jelszót a tárolt hash-elt verzióval.

Ha egyeznek, akkor a bcrypt.hash() segítségével újra titkosítjuk az új jelszót.

Végül egy UPDATE utasítással frissítjük az users táblában a password\_hash mezőt az új hash-sel.

Ha bármelyik lépés során hiba történik, a rendszer megfelelő hibaüzenetet küld vissza (400, 404, vagy 500 státuszkóddal). Sikeres jelszómódosítás esetén a válasz:

{ message: 'A jelszó sikeresen megváltozott.' }.

Ez a funkció fontos része egy biztonságos felhasználói rendszernek, mivel lehetőséget biztosít arra, hogy a felhasználó saját maga módosítsa jelszavát a hitelesítés után.

A **/api/votes** útvonalon elérhető GET típusú API-végpontot arra hoztam létre, hogy lehetőséget biztosítson egy adott felhasználó korábban leadott szavazatainak lekérdezésére, a felhasználó azonosítója alapján.

// API végpont a szavazatok lekérésére

app.get('/api/votes', (req, res) => {

  const userId = req.query.user\_id; // A felhasználó ID-ját a query paraméterből kapjuk

  if (!userId) {

    return res.status(400).json({ message: 'User ID is required' });

  }

  const query = 'SELECT \* FROM votes WHERE user\_id = ?';

  db.query(query, [userId], (err, results) => {

    if (err) {

      console.error('Hiba a szavazatok lekérésekor:', err);

      return res.status(500).json({ message: 'Hiba történt' });

    }

    // Visszaadjuk a szavazatokat

    res.json(results);

  });

});

Ez a végpont lekérdezi az adatbázisból az adott felhasználóhoz tartozó összes szavazatot, ha az user\_id paraméter szerepel az URL-ben query formában, például így:

/api/votes?user\_id=42.

Működés lépései:

A req.query.user\_id segítségével lekéri a felhasználó azonosítóját a lekérdezésből.

Ha nincs megadva user\_id, akkor 400 Bad Request válasz érkezik.

A rendszer SQL-lel lekérdezi a votes táblát, ahol user\_id megegyezik a megadott értékkel.

Hibakezelés történik: ha a lekérdezés során hiba történik, akkor 500 Internal Server Error kerül visszaküldésre.

Siker esetén visszaadja az eredményt JSON formátumban – az adott felhasználó összes szavazatát.

Regisztráció

Regisztráció nélkül nem tudnánk különbséget tenni a látogatók és a felhasználók között. Ahhoz, hogy a látogatóból funkciókat élvező felhasználó lehessen, ki kell töltenie 4 darab szövegdobozt (<input type="text">), amelyek a teljes név, email cím, adószám, illetve a jelszó megerősítése mezőket tartalmazzák. Emellett szerepel 2 darab jelszómező (<input type="password">), melyek közül az egyik egy JavaScript függvénnyel átváltható normál szöveges mezővé, így a felhasználó láthatja az általa beírt jelszót.

Jelen oldalon a felhasználónak a regisztráció érvényesítéséhez el kell fogadnia az Általános Szerződési Feltételeket egy checkbox kipipálásával. Nem látható ezen az oldalon nemválasztó rádiógomb vagy születési dátum kiválasztására szolgáló select mező, de a jövőbeli bővítés során ezek is beépíthetők lehetnek. A regisztráció végén a felhasználó a „Regisztráció” feliratú gombra kattintva véglegesítheti az adatokat, amelyek ezt követően bekerülnek az adatbázisba – ezt egy felugró ablak fogja visszaigazolni.

Bejelentkezés

A bejelentkezésre azért van szükség, hogy a látogató korlátolt lehetőségeit feloldja, és élhessen a felhasználói kiváltságokkal (például szavazás, profilkezelés, admin funkciók elérése stb.). A felhasználó regisztráció után automatikusan át lesz irányítva erre a bejelentkezési űrlapra (<form>), ahol az első, felső szövegdobozban megadhatja az email címét ($\_POST['Felhasznalo\_Email']), az alatta lévő jelszómezőben pedig a jelszavát ($\_POST['Felhasznalo\_Jelszo']).

A megadott adatokat a rendszer az adatbázisban tárolt értékekkel hasonlítja össze, vagyis az email-cím (Felhasznalo\_Email) alapján keres, majd a megadott jelszót (Felhasznalo\_Jelszo) ellenőrzi. Az ellenőrzés a felhasznalok tábla megfelelő rekordjai között történik.

Amennyiben a $\_POST értékek megegyeznek a Felhasznalo\_Email és Felhasznalo\_Jelszo mezők adatbázisban tárolt értékeivel, és a felhasználó rákattintott a „Bejelentkezés” gombra a form alján, a rendszer sikeresen belépteti a felhasználót. Ezt egy felugró ablak jelzi majd, ahol keresztnevén köszönti és üdvözli őt.

Szavazat leadása

Hogyan lehet szavazni?

A szavazásban való részvételhez több lépésen keresztül vezet az út, amely biztosítja a rendszer átláthatóságát, hitelességét és a felhasználók egyediségét.

1. Regisztráció:

Ahhoz, hogy szavazni tudjon, először regisztrálnia kell az oldalon. A regisztráció során meg kell adnia a teljes nevét, email címét, jelszavát, valamint egy érvényes adószámot. A jelszót kétszer kell megadni a pontosság ellenőrzése érdekében. Továbbá el kell fogadnia az Általános Szerződési Feltételeket is. Ezek után a „Regisztráció” gombra kattintva a rendszer létrehozza a felhasználói fiókot.

2. Bejelentkezés:

A regisztrációt követően a felhasználónak be kell jelentkeznie az email címe és a jelszava megadásával. A rendszer összeveti a megadott adatokat az adatbázisban szereplő adatokkal, és ha minden helyes, sikeres bejelentkezés történik.

3. Szavazás elérhetősége:

A szavazás csak akkor lehetséges, ha az éppen aktív, vagyis az admin által meghatározott időintervallumon belül zajlik. Amennyiben a szavazás még nem kezdődött el, vagy már lezárult, a szavazási felület nem lesz elérhető.

4. Szavazás menete:

Ha a szavazás aktív, a bejelentkezett felhasználó megtekintheti a listázott pártokat, és kiválaszthatja azt, amelyikre szavazni szeretne. A szavazatot egyetlen kattintással lehet leadni az adott párt mellett található „Szavazok” gomb megnyomásával. A szavazat leadása végleges, és egy felhasználó csak egy alkalommal szavazhat a választási időszak alatt.

Teszt dokumentáció

A weboldalunk fejlesztése során kiemelten fontosnak tartottuk a stabilitást és megbízhatóságot, ezért a funkciók ellenőrzését teszteléssel is biztosítjuk. A tesztelési folyamatokat két fő részre osztottuk: backend és frontend tesztekre, így külön kezeljük a szerveroldali és a felhasználói felület működésének ellenőrzését.

**Backend tesztek – Postman**

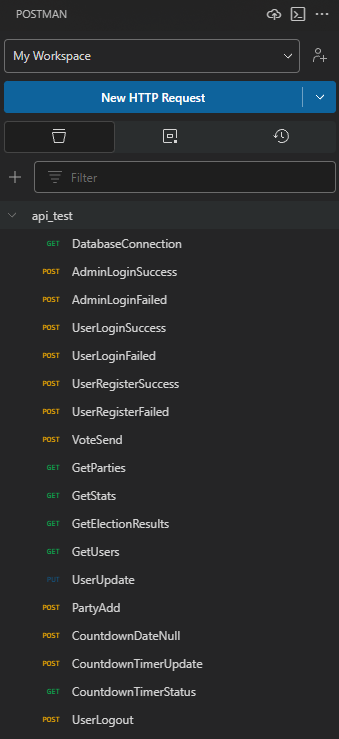
A backend teszteket a **Postman** nevű eszköz segítségével készítettük el, amely kiválóan alkalmas REST API-k tesztelésére. Ezek a tesztek lefedik az összes fontos szerveroldali műveletet, például:

* Regisztráció és bejelentkezés
* Szavazási lehetőség ellenőrzése
* Felhasználói jogosultságok kezelése
* Adatbázis-műveletek (pl. új felhasználó létrehozása, törlés, frissítés)

**Miért jó a Postman tesztelés?**

* Lehetővé teszi a **gyors és ismételhető tesztelést** különféle bemeneti adatokkal.
* Segítségével **automatizált tesztcsomagokat** is létrehozhatunk, amelyek futtatása fejlesztés után gyors hibakeresést tesz lehetővé.
* Könnyedén **vizsgálhatjuk a válaszokat, hibakódokat**, és validálhatjuk az API működését.

**Miért választottuk?**  
A Postman egy könnyen használható, de mégis erőteljes eszköz, amely kifejezetten alkalmas volt a Node.js alapú backendünk REST API végpontjainak tesztelésére. Fejlesztés közben gyors visszajelzést adott a működésről, és segített elkerülni a hibás logikák élesítését.



**Frontend tesztek – Playwright + JavaScript**

A frontend (React) alkalmazás teszteléséhez a **Playwright** könyvtárat használtuk JavaScript nyelven. A Playwright egy modern, automatizált UI-teszt eszköz, amely támogatja a legnépszerűbb böngészőket, és képes **felhasználói interakciók szimulálására**.

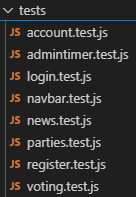
A tesztjeink az alábbiakat fedik le:

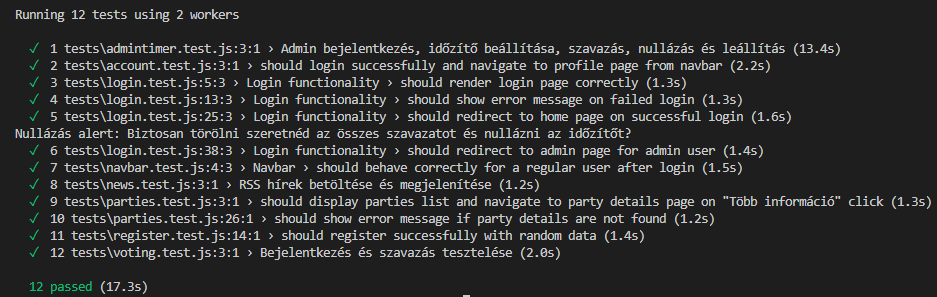
* Felhasználói regisztrációs és bejelentkezési folyamat
* Szavazás gomb megjelenése, elérhetősége
* Admin felület hozzáférésének ellenőrzése
* Szavazás gomb inaktivitása, ha a szavazás nem aktív

**Miért jó a Playwright?**

* Lehetővé teszi a **valós böngészőben való tesztelést**, ezzel közel valós felhasználói élményt szimulál.
* **Gyorsan fut**, stabil, és könnyen integrálható CI/CD környezetekbe.
* Képes képernyőképeket, videókat készíteni, valamint **hiba esetén pontos hibát és helyet jelez**.

**Miért választottuk?**  
A Playwright kiváló támogatást nyújt a React alapú felületünk interaktív elemeinek automatizált teszteléséhez. Azért döntöttünk mellette, mert megbízhatóbb és modernebb alternatívát kínál például a régebbi Selenium alapú megoldásoknál, és könnyen kezelhető JavaScript környezetben.





Összegzés:

A tesztelés így átfogó: a Postman biztosítja az API-k és szerveroldali logika helyes működését, míg a Playwright a felhasználói felület tesztelését végzi. Ezzel a megközelítéssel nemcsak gyors hibafelismerést tudunk elérni, hanem a felhasználói élményt és az oldal funkcionalitását is hosszú távon megbízhatóvá tesszük.

Felhasználói Dokumentáció

A weboldalam célja, és amit Te profitálhatsz belőle:

Kedves Felhasználó! Egy olyan modern, működő platformon jársz, amely lehetőséget biztosít arra, hogy aktívan részt vegyél a politikai közéletben, informálódj, és véleményt nyilváníts a választások során. Regisztráció és bejelentkezés után az alábbi funkciókat érheted el:

– Aktuális politikai hírek olvasása: Első kézből értesülhetsz a legfrissebb politikai eseményekről, törvényjavaslatokról és pártállásokról.

– Pártok részletes megtekintése: Megismerheted a szavazásban részt vevő pártokat, programjaikat és célkitűzéseiket, így tudatos döntést hozhatsz.

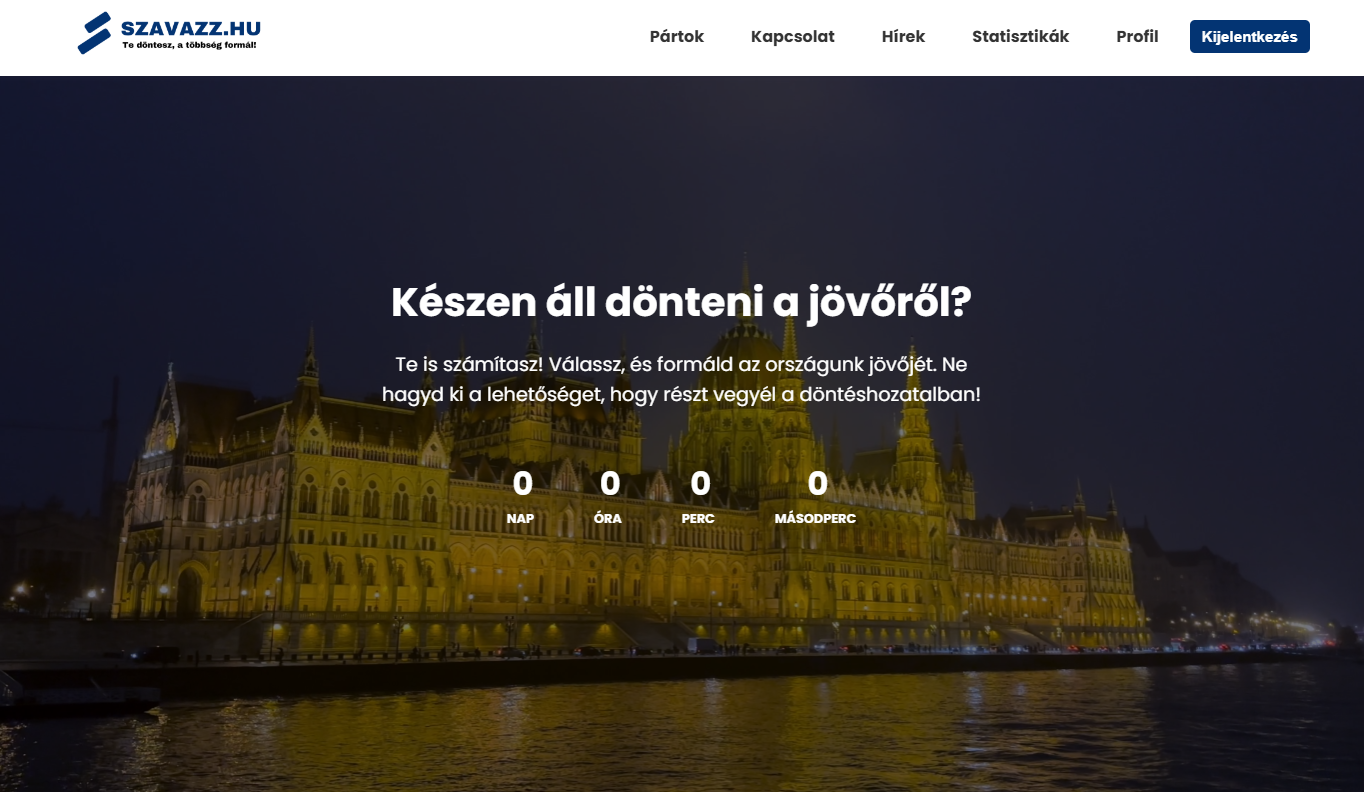
– Szavazás leadása: Amennyiben a szavazás aktív, lehetőséged van arra, hogy egyetlen kattintással lead a voksodat a kiválasztott pártra – mindezt biztonságos, felhasználóbarát felületen keresztül.

– Eredmények megtekintése: A szavazás lezárulta után nyilvános formában elérhetővé válik az eredmény, így mindenki számára átlátható lesz a voksolás végeredménye.

– Jelszó módosítása: Amennyiben szeretnéd frissíteni a jelszavadat, ezt egyszerűen, néhány kattintással megteheted a profilodon belül.

– Kapcsolatfelvétel az üzemeltetőkkel: Ha kérdésed merül fel, hibát tapasztalsz az oldalon, vagy bármilyen visszajelzést szeretnél küldeni, az erre kialakított kapcsolatfelvételi űrlapon keresztül gyorsan és egyszerűen megteheted.

A kezdőlap és a menü fejléc



Amint a szavazz.hu címre átkerültél, meg fogod látni fent ezt, fejlécet, a menüt.

Itt választhatsz az alábbiak közül:

-Pártok: Itt lehet megtekinteni azokat a pártokat amelyekre majd később le lehet adani a szavazatokat

- Kapcsolat: Erre rálépve itt lehet velünk felvenni a kapcsolatot ha bármilyen hibát érzékelsz vagy ha valamiben segítségünket kérnéd.