Webtechnológia és webalkalmazás-fejlesztés - Frontend (React)

Frontend fejlesztés React könyvtárral

Magda Donát

Széchenyi István Egyetem, Győr

https://git.mdnd-it.cc/Donat/GKNB$_M$STM071

2026. március 17.

Mi a React?

Rövid definíció

A React egy nyílt forráskódú JavaScript **könyvtár**, amelyet a Meta (Facebook) fejlesztett ki. Fő célja: webalkalmazások felhasználói felületének (UI) hatékony és karbantartható megírása.

- Nem egy teljes keretrendszer — csak a megjelenítési réteggel foglalkozik.
- Kiegészítő könyvtárakkal (routing, adatkezelés) teljes alkalmazás is építhető.
- 2013 óta aktívan fejlesztik; ma az egyik legelterjedtebb frontend technológia.

Deklaratív vs. imperatív szemlélet

Régi, imperatív szemlélet

„Keresd meg a gombot, módosítsd a szövegét, frissítsd a listát. . . "

⇒ Megadjuk, *hogyan* csinálja a böngésző lépésről lépésre.

React: deklaratív szemlélet

„Ha a kosár üres, ezt jelenítsd meg; ha van termék, amazt."

⇒ Megadjuk, *mit* akarunk látni — a React intézi a DOM-frissítést.

Mi az a virtuális DOM?

DOM = Document Object Model

A böngésző az oldalt egy faszerkezetben (DOM) tárolja. Sok elem esetén a közvetlen DOM-módosítás lassú lehet.

- A React egy **virtuális DOM**-ot tart fenn a memóriában — ez egy gyors, könnyű másolata a valódi DOM-nak.
- Állapotváltozáskor összehasonlítja az új és a régi virtuális DOM-ot (*diffing*).
- Csak a ténylegesen megváltozott részeket frissíti a böngészőben — ezért gyors és hatékony.

Miért érdemes React-et tanulni?

- **Újrafelhasználhatóság**: egyszer megírt komponens bárhová beilleszthető.
- **Nagy közösség**: rengeteg oktatóanyag, könyvtár és álláshirdetés.
- **Skálázhatóság**: kis projekttől nagy vállalati alkalmazásig alkalmazható.
- **Átváltható tudás**: React Native-szal mobilalkalmazás is írható ugyanezzel a szemlélettel.

Projektindítás Vite-tal

Miért Vite?

A Vite egy modern build eszköz: gyorsan indítja el a fejlesztői szervert, és azonnal tükrözi a módosításokat a böngészőben. Ezt hívják HMR-nek (Hot Module Replacement — „élő újratöltés").

```
npm create vite@latest sajat-app -- --template react
cd sajat-app
npm install
npm run dev
```

- Az `npm run dev` elindítja a fejlesztői szervert (alapból `localhost:5173`).
- `npm run build` paranccsal production-kész, tömörített kódot kapunk.

JSX — HTML és JavaScript egyben

Mi az a JSX?

A JSX (JavaScript XML) egy szintaktikai kiterjesztés: HTML-szerű kódot írhatunk JavaScript-ben. A böngésző közvetlenül nem érti — a build eszköz fordítja sima JavaScript-té.

- JavaScript kifejezéseket `{}` zárójelbe írjuk: változó, függvényhívás, feltétel.
- Minden JSX elemnek pontosan egy gyökéreleme lehet (vagy `<>...</>` fragment).
- A `class` HTML attribútum neve JSX-ben `className` — a `class` JS-ben foglalt szó.

Mi az a komponens?

Analógia: LEGO kockák

Képzeljük el, hogy a weboldalunk LEGO-ból épül. Minden egyes „kocka" egy önálló, újrafelhasználható egység — ezt nevezzük **komponensnek**.

- Egy komponens egy JavaScript **függvény**, amely JSX-et ad vissza — leírja, mit jelenítsen meg.
- A teljes oldal komponensek fájából áll: van egy gyökér (`App`), abból nőnek ki az alkomponensek.
- Például: `<Header />`, `<ProductList />`, `<Footer />` — mindegyik egy-egy felelősségi kör.

Funkcionális és osztály alapú komponensek

- **Funkcionális komponens (modern)**: egyszerű JavaScript függvény, hookok segítségével kezeli az állapotot. Ez az ajánlott forma.
- **Osztály alapú komponens (régebbi)**: ES6 osztályból örököl, életciklus-metódusokat használ. Ma már nem szükséges új kódban alkalmazni.
- A React csapat 2019 óta a funkcionális megközelítést javasolja — rövidebb, könnyebben tesztelhető.

Mi az a prop?

Analógia: paraméterek

Ahogy egy függvénynek paramétereket adhatunk, a komponenseknek is adhatunk adatokat kívülről. Ezeket **props**-nak (properties — tulajdonságok) hívjuk.

- A props **csak olvasható**: a komponens nem módosíthatja a kapott értéket.
- Props segítségével ugyanaz a komponens különböző tartalommal jeleníthető meg — ez teszi újrafelhasználhatóvá.
- Szinte bármilyen érték átadható: szöveg, szám, tömb, függvény, sőt más komponens is.

A `children` prop és a kompozíció

- A `children` egy speciális prop: a komponens nyitó- és zárócímkéje *közé* írt tartalmat jelenti.
- Például egy `<Modal>` komponens bármilyen tartalmat fogadhat belülről — nem kell előre tudni, mi lesz benne.
- Ez a **kompozíció** alapelve: összetett UI-t kisebb, általános darabokból rakunk össze öröklés helyett.

Kulcsok (key) listáknál

Miért kell a key?

Ha React egy listát renderel, azonosítania kell, melyik elemet kell frissíteni, törölni vagy hozzáadni. A key prop ebben segít — egyedi azonosítóként szolgál.

- A key értéke legyen **stabil és egyedi** — tipikusan az adatbázisból érkező ID.
- A tömb indexét csak akkor használjuk, ha a lista soha nem változik sorrendben.
- Hiányzó vagy nem egyedi key esetén React figyelmeztetést dob, és hibás frissítések léphetnek fel.

Mi az a Hook?

Az alapgondolat

A React 16.8 előtt az állapotkezelés és életciklus-logika csak osztály alapú komponensekben volt lehetséges. A hookok ezt hozták el **funkcionális komponensekbe** is — így ma már egyszerűbb szintaxissal írhatunk komplex viselkedést.

- A hook neve mindig `use`-szal kezdődik (pl. `useState`, `useEffect`).
- Beépített hookok lefedik a leggyakoribb igényeket: állapot, mellékhatás, kontextus, optimalizálás.
- **Custom hook**: saját logikát csomagolhatunk újrafelhasználható hookba — pl. `useWindowSize`, `useAuth`.

useState — az alapvető állapot hook

Mit csinál?

Az `useState` lehetővé teszi, hogy egy komponens **emlékezzen** valamire a renderelések között — pl. egy számláló értékére, egy modal nyitott/zárt állapotára.

- Visszaad egy értéket és egy **szetter függvényt**: `const [ertek, setErtek] = useState(kezdoErtek)`.
- Ha a szetter meghívódik, a React **újrarendereli** a komponenst az új értékkel.
- A kezdeti érték csak az első renderelésnél számít — utána React maga kezeli.

useReducer — összetettebb állapothoz

- Ha az állapot több összefüggő értékből áll, vagy a frissítési logika bonyolult, `useReducer` jobb választás lehet.
- Működése hasonló a Redux elvéhez: van egy **akció** (mi történt) és egy **reducer** függvény (hogyan változik az állapot).
- A frissítési logika egy helyen, a reducerben él — olvashatóbb, tesztelhető.

useRef — referencia DOM elemre vagy értékre

- Az useRef olyan értéket tárol, amely **nem vált ki újrarenderelést** megváltozásakor.
- Leggyakoribb használat: közvetlen DOM-elem elérése — pl. input fókuszálása, video lejátszás indítása.
- Hasznos timer ID-k és előző értékek tárolására is: bármire, amit renderelések között „el akarunk menteni" anélkül, hogy az oldal frissüljön.

Hook szabályok — miért fontosak?

Alapszabályok

1. Hookot csak **komponens vagy custom hook legfelső szintjén** hívjunk.
2. Ne ciklusban, feltételben vagy beágyazott függvényen belül hívjuk.

- **Miért?** A React a hookok *hívási sorrendje* alapján párosítja az állapotokat a komponenshez. Ha a sorrend változik (pl. feltétel miatt kimarad egy hook), React összetéveszti, melyik állapot melyikhez tartozik.
- Az ESLint `eslint-plugin-react-hooks` csomag automatikusan ellenőrzi ezeket.

Mi az állapot (state)?

Analógia: a komponens "emlékezete"

Az állapot az, amit a komponens **megjegyez** renderelések között. Gondoljunk rá úgy, mint egy cetlire, amire a komponens feljegyzi az aktuális értékeit — hány termék van a kosárban, be van-e jelentkezve a felhasználó, milyen szöveg van az input mezőben.

- Az állapot megváltozásakor a React **automatikusan újrarendereli** az érintett komponenst.
- Az állapot a komponensen *belül* él — más komponens nem látja közvetlenül.

Állapot típusok a gyakorlatban

- **Lokális UI állapot**: csak egy komponenst érint — pl. le van-e hajtva egy dropdown, aktív-e egy fül.
- **Megosztott állapot**: több komponens is olvas vagy ír belőle — pl. a bejelentkezett felhasználó neve.
- **Szerverállapot**: API-ból érkező, kiszolgálón élő adat, amelyet le kell kérni és szinkronban kell tartani.
- **URL állapot**: az aktuális útvonal, query paraméterek — természetesen megosztható és könyvjelzőzhető.

Egyirányú adatfolyam

Mi ez pontosan?

Reactben az adatok mindig **szülőtől gyermek felé** áramlanak (props-on keresztül). A gyermek *nem módosíthatja* a szülő állapotát közvetlenül — csak egy callback függvényen (eseménykezelőn) keresztül jelezhet vissza.

- Ezért kiszámítható: mindig tudjuk, honnan ered egy érték.
- Hibakeresés könnyebb — az adatfolyamot felfelé haladva követhetjük.

State emelés (Lifting State Up)

- Ha két testvérkomponensnek ugyanazt az adatot kell látnia, az állapotot a legközelebbi közös ősbe **emeljük fel**.
- Az ős kezeli az állapotot, és props-on keresztül osztja szét a gyermekek között.
- **Példa**: egy szűrő mező és egy lista — mindkettőnek ugyanaz a szűrési feltétel kell, ezért azt a szülőben tároljuk.

Mikor kell külső állapotkezelő?

- Ha az állapot emelése sok szinten keresztül megy, és a köztes komponensek csak "átpasszolják" — ezt hívjuk **prop drilling**-nek.
- Ha az állapot nagyon sok helyen kell, és bonyolult szinkronizálás szükséges.
- Külső megoldások: **Redux Toolkit** (nagy, komplex alkalmazásokhoz), **Zustand** (könnyű, egyszerű API), **Jotai** (atomi, finomhangolt).
- Legtöbb esetben a beépített eszközök (useState + Context) elegendők — ne bonyolítsuk túl.

Kontrollált és nem kontrollált input

Kontrollált input

Az input mező értékét a React állapot tárolja és vezérli. A `value` prop és az `onChange` esemény mindig szinkronban tartja a React állapotot és a mezőt — bármikor pontosan tudjuk, mi van a mezőben.

Nem kontrollált input

A DOM maga kezeli az értéket — React csak beküldéskor kéri le (`useRef`-fel). Egyszerűbb felépítés, de valós idejű validáció nehezebb.

Miért érdemes kontrollált inputot használni?

- Mindig pontos képünk van az aktuális beviteli értékről — nem kell a DOM-ból "kiolvasni".
- **Valós idejű validáció:** hibát jelzünk gépelés közben, nem csak beküldéskor.
- Feltételes engedélyezés/tiltás, formázás, karakterlimit könnyen kezelhető.
- Hátránya: sok mezőnél sok state szükséges — erre nyújtanak megoldást a könyvtárak.

Validáció — mit kell ellenőrizni?

- **Kötelező mezők**: ne lehessen üres beküldés.
- **Formátum**: email, telefonszám, irányítószám megfelelő-e?
- **Üzleti szabályok**: min. 8 karakteres jelszó, egyező jelszavak, stb.
- A hibákat erős vizuális visszajelzéssel kell jelezni — a felhasználó ne találgasson, mi a baj.

Miért használjunk könyvtárat?

- Kézi megvalósításnál sok ismétl'd' kód keletkezik — minden mez'höz saját state, validáció, hibaüzenet.
- **React Hook Form**: minimális újrarenderelés, nagyon gyors, könnyen tanulható API — a legtöbb esetben ez az ajánlott.
- **Formik**: régebben nagyon népszerű, sok csapat már ismeri; kicsit több boilerplate.
- **Zod / Yup**: séma alapú validáló könyvtárak — a validációs logikát típusbiztos sémában írjuk le, React Hook Form-mal jól kombinálható.

Mi az a mellékhatás (side effect)?

Definíció

Mellékhatásnak nevezzük mindazt, ami a komponens **renderelésén kívül** történik: hálózati kérés, DOM-módosítás, timer beállítása, eseménykezelő regisztrálása.

- A renderelési fázisban ilyen műveletek nem végzhetők — a React elvárja, hogy a renderelés **tiszta** (mellékhatásmentes) legyen.
- A `useEffect` hook nyújtja a lehetőséget arra, hogy e műveleteket *renderelés után* hajtsuk végreh.

`useEffect` — mikor és hányszor fut le?

- **Függőségi történk**: minden egyes renderelés után lefut — általában nem ezt akarjuk.
- **Üres történk (`[]`)**: csak az első megjelenítéskor fut le egyszer — pl. induló adatlekéréshez.
- **Értékekkel (`[id, szuro]`)**: amikor a felsorolt értékek bármelyike megváltozik, újra lefut.

A függőségi történk tehát azt szabályozza, **mikor** akarjuk az effektet ismét futtatni.

Cleanup — takarítás az effect után

Miért kell cleanup?

Ha egy komponens eltűnik az oldalról (pl. navigáció közben), az esetlegesen elindított timerek, eseményfigyelők tovább futnak, memóriaszivárgást okozva. A cleanup függvény ezt akadályozza meg.

- Az effect **visszaadhat egy cleanup függvényt**: ez fut le a komponens unmountjakor, illetve az effect következő lefutása előtt.
- Tipikus eset: `clearInterval`, `removeEventListener`, API kérés megszakítása (`AbortController`).

Leggyakoribb hibák `useEffect`-tel

Figyelem

- **Hiányzó dependency**: ha egy változót használunk de nem szerepel a tömbben, a hook régi értékkel dolgozhat — ez nehezen felderíthető hiba.
- **Végtelen ciklus**: ha az effect frissít egy állapotot, ami maga is dependency, kör forgás indul.
- **Túl sok logika egy effectben**: egy effect egy felelősséget kezeljen; ha kettőt kezelne, bontsuk szét.

Mit jelent a kliensoldali navigáció?

Hagyományos vs. SPA navigáció

Hagyományos weboldalon minden oldalváltáskor a böngésző **új HTTP kérést** küld, és teljesen újratölti az oldalt. SPA (Single Page Application) esetén az oldal egyszer töltődik be, a navigáció pedig JavaScript-tel történik, ezért gyorsabb és folyamatosabb élményt ad.

A React Router a legelterjedtebb könyvtár erre a feladatra React alkalmazásokban.

React Router — hogyan működik?

- A `BrowserRouter` figyeli a böngésző URL-jét.
- A `Routes` és `Route` elemek URL mintákhoz komponenseket rendelnek.
- A `Link` URL-t vált teljes oldalfrissítés nélkül.
- A React újrarendereli a megfelelő oldalkomponenst az aktuális útvonal alapján.

React Router példa kód 1/2

```
import { BrowserRouter, Routes, Route, Link } from "react-router-dom";

    function App() {
    return (
        <BrowserRouter>
        <nav>
            <Link to="/">Fooldal</Link>
            <Link to="/termekek">Termekek</Link>
        </nav>
```

React Router példa kód 2/2

```
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/termekek" element={<Products />} />
        <Route path="/termekek/:id" element={<ProductDetails />} />
      </Routes>
    </BrowserRouter>
  );
}
```

Dinamikus útvonalak (React Router)

- Paraméteres route példa: `/termekek/:id`
- Ha az URL `/termekek/42`, akkor a `useParams()` eredménye: `{ id: "42" }`
- Tipikusan ilyen paraméterrel kérünk le részletes adatot API-ból.

Mappa routing (file-system routing) — mi ez?

Alapelv

Mappaalapú routingnál az útvonalakat nem külön `Routes` konfigurációban írjuk, hanem a mappastruktúra **automatikusan** meghatározza őket.

- Ez főleg Next.js-ben elterjedt (App Router).
- Az `app/` mappában minden `page.tsx` egy útvonalat jelent.
- Dinamikus route mappanévben: `[id]`.

Mappa routing példa (Next.js App Router)

```
app/
  page.tsx                -> /
  kapcsolat/
    page.tsx              -> /kapcsolat
  termekek/
    page.tsx              -> /termekek
    [id]/
      page.tsx            -> /termekek/:id
```

Dinamikus mappa route komponens példa

```
// app/termekek/[id]/page.tsx
export default function ProductPage({ params }) {
  return <h2>Termek azonosito: {params.id}</h2>;
}
```

React Router vs. mappa routing

- **React Router**: explicit útvonal definíció a kódban (rugalmas, kontrollált).
- **Mappa routing**: mappaszerkezetből automatikus route (gyors indulás, kevesebb konfiguráció).
- Kezdőknek mindkettőt érdemes ismerni: React Router SPA-hoz, mappa routing Next.js projektekhez.

A prop drilling probléma

Mi az a prop drilling?

Ha egy adat a komponensfa tetején él, de csak mélyen lent lévő komponensnek kell, akkor az összes közbülső komponensen át kell „ütni" props-szal — még akkor is, ha a köztes komponensek maguk nem használják. Ez nehézkessé és törékennyé teszi a kódot.

A **Context API** ezt a problémát oldja meg: az adatot egy közös „kút"-ban tároljuk, amelyből bármely leszármazott komponens közvetlen props nélkül is ihat.

Mire jó a Context API?

- **Ritkán változó, globális adatok** átadásához ideális — pl. téma (dark/light), aktív nyelv, bejelentkezett felhasználó adatai.
- Két lépés: 1) **Provider** — közzéteszi az értéket a fában; 2) **useContext** — olvasásra foglalja le a komponensben.
- Nem teljes értékű állapotkezelő: sűrűn változó adatokhoz ne használjuk, mert minden fogyasztó komponens újrarenderelődik.

A Context és a Provider szerepe

Context

A `createContext` egy konténer, amely az adatot tárolja. Maga nem csinál semmit — csak definiálja, hogy melyik adat lesz megosztva.

Provider

A `Provider` egy komponens, amely körülvesz más komponenseket, és az értéket **elérhetővé teszi** számukra. Minden komponens, amely a Provider-ben van, felhasználhatja az adatot.

Context és Provider — alapvető kódpélda

```
import { createContext, useContext } from "react";

// 1. Context létrehozása
const ThemeContext = createContext("light");

// 2. Provider komponens a legfelső szinten
function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Header />
      <Content />
    </ThemeContext.Provider>
  );
}
```

Context és Provider — alapvető kódpélda folytatás

```
// 3. Leszármazott komponensben: useContext
function Header() {
  const theme = useContext(ThemeContext);
  return <h1>Tema: {theme}</h1>;
}
```

Állapottal rendelkező Context — teljesebb példa

```
// ThemeContext.js
const ThemeContext = createContext();

export function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

export function useTheme() {
  return useContext(ThemeContext);
}
```

Felhasználás az alkalmazásban

```
// App.js
import { ThemeProvider } from "./ThemeContext";

function App() {
  return (
    <ThemeProvider>
      <Header />
      <MainContent />
    </ThemeProvider>
  );
}
```

Felhasználás az alkalmazásban folytatás

```
// Header.js
function Header() {
  const { theme, setTheme } = useTheme();
  return (
    <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
      Aktualis tema: {theme}
    </button>
  );
}
```

Hogyan működik a gyakorlatban?

- `createContext()` a kontextus objektumot hozza létre.
- A `Provider` értéket ad át a `value` prop-on.
- Mélyen lent elhelyezkedő komponensek a `useContext()` hookkal használják az adatot.
- Custom hook (pl. `useTheme()`) felhasználóbarátabb API-t nyújt.

Jó gyakorlatok és gyakori hibák

- **Custom hook**: szokjunk meg a saját hook megírásának (pl. `useTheme`), ez takarítja a kódot és segíti az újrafelhasználást.
- **Túl nagy Context**: ha túl sok adat van egy Context-ben, minden frissítés újrarendereli az összes fogyasztót — inkább bontsuk fel több kisebb Context-re.
- **nem Context-ben való adat**: gyorsan változó, nagy mennyiségű adat (pl. valós idejű pozíciók) nem Context-ben való — inkább store (Redux, Zustand) vagy külön state.
- **Default érték**: adjunk adjunk értelmes `createContext()` paraméterét — dev módban segít descobrire a hiányzó Providert.

Context vs. más megoldások

- **Props**: egyszerű, néhány szintű adatátadásra. Érthető, explicit.
- **Context**: ritkán változó, globális adatok. Szétszórt komponensek között.
- **State Manager (Redux/Zustand)**: komplett üzleti logika, komplex frissítések, szinkronizálás.
- **Szerverállapot (TanStack Query)**: API-ból érkező adatok, cache, szinkronizálás.

A CSS megközelítések áttekintése

- **Globális CSS**: hagyományos, minden komponens látja — névütközések kockázatával.
- **CSS Modules**: minden komponensnek saját, automatikusan lokalizált CSS osztályai vannak — nincs névütközés.
- **CSS-in-JS** (pl. styled-components, Emotion): közvetlenül JavaScript-ben írunk stílusokat, dinamikus értékek könnyen használhatók.
- **Utility-first** (Tailwind CSS): előre definiált kis osztályokat kombinálunk — nincs saját CSS fájl.

Mi az a Tailwind CSS?

Az alapötlet

A Tailwind CSS nem komponenseket, hanem apró, egy-egy CSS tulajdonságot képviselő **utility osztályokat** ad. Ezeket közvetlenül a JSX-be írjuk — pl. `flex`, `pt-4`, `text-gray-700`.

- Előny: nem kell kitalálni osztályneveket, nincsenek névütközések, konzisztens design tokenek.
- Hátrány: a JSX-ben hosszú osztálylisták keletkezhetnek — komponens-szintű absztrakciókkal kezelhető.

Design rendszer gondolkodás

- Jól felépített alkalmazásban definiálunk **UI primitíveket**: `Button`, `Input`, `Card`, `Badge` stb.
- Ezek belül rejtenek el minden stílust — a felhasználójuk csak a variánst adja meg (pl. `variant="primary"`).
- **Egységes spacing, színek, tipográfia**: ezeket design tokenek (Tailwind config vagy CSS változók) rögzítik, nem ismétlünk hardcoded értékeket.

Hozzáférhetőség (accessibility)

- **Kontraszt**: a szöveg és háttér közötti kontraszt arány elégítse ki a WCAG irányelveket.
- **Fókuszállapot**: billentyűzettel is navigálható legyen az oldal — a fókuszjelzőt ne távolítsuk el.
- **Szemantikus HTML**: helyes heading hierarchia, gomb valóban `<button>`, ne `<div onClick>`.
- **ARIA attribútumok**: ahol a HTML szemantika nem elég (pl. egyedi komponensek).

Hogyan kap adatot egy React alkalmazás?

Az alaphelyzet

A React alkalmazás a böngészőben fut — az adatok (felhasználók, termékek, stb.) általában egy **háttérszerveren** élnek. Az adatcserét HTTP kérésekkel végezzük: a frontend elküldi a kérést, a szerver JSON formátumban válaszol.

- A böngésző beépített `fetch` API-ját vagy az **Axios** könyvtárat használhatjuk.
- Az adatlekérést általában egy `useEffect`-ben indítjuk el — amikor a komponens megjelenik.

Fetch vs. Axios

- **Fetch**: beépített, nem kell telepíteni. Hátránya: a hibás HTTP kódokat (pl. 404) *nem* dobja el automatikusan, JSON-t kézzel kell kezelni.
- **Axios**: telepíthető könyvtár. Automatikusan kezeli a JSON-t, a HTTP hibákra kivételt dob, és **interceptorokkal** (pl. token hozzáadása minden kéréshez) könnyű kiegészíteni.
- Nagyobb projekteknél az Axios általában kényelmesebb és biztonságosabb.

Állapotok az adatlekérés során

- **Betöltés (loading)**: amíg a kérés folyamatban van, mutassunk spinner-t vagy skeleton-t — ne maradjon üres az oldal.
- **Siker**: megérkeztek az adatok, megjelenítjük.
- **Hiba (error)**: a hálózat megszakadt, a szerver hibát adott — barátságos hibaüzenetet mutassunk, ne csak üres felületet.
- **Üres állapot (empty)**: a válasz sikeres, de nincs adat — pl. "Nincsenek még termékek".

Szerverállapot-kezelés: TanStack Query

- Manuálisan kezelni a loading/error/data állapotokat minden komponensben ismétlődő és hibalehetőséges.
- A **TanStack Query** (korábbi nevén React Query) ezt automatizálja: cache-eli az adatokat, háttérben frissíti, kezeli az újrapróbálkozást.
- **Optimista frissítés**: azonnal frissítjük a UI-t a szerver válasza előtt — ha a szerver hibát ad, visszaállítjuk. Így az alkalmazás gyorsabbnak tűnik.

Mi az az újrarenderelés?

Az alapjelenség

Amikor egy komponens állapota vagy props-a megváltozik, a React **újrarendereli** azt a komponenst — és alapértelmezetten az összes gyermekét is. Ez legtöbbször tökéletesen gyors. Nagy és mély fáknál azonban szükségtelen újrarenderelések lassíthatnak.

Fontos szabály: **mérjünk, aztán optimalizáljunk!** Ne alkalmazzunk előre memoizációt ott, ahol valójában nincs probléma.

React DevTools Profiler

- A React DevTools böngésző-bővítmény beépített **Profiler** lapján látjuk, hogy melyik komponens hányszor és mennyi ideig rendereledett.
- Felvesszük egy interakció közben a renderelési folyamatot, majd azonosítjuk a lassú vagy felesleges részeket.
- Csak azután nyúljunk optimalizáláshoz, ha a profiler valódi problémát mutat.

Memoizáció — mit és mikor?

- **React.memo**: ha egy komponens props-ai nem változnak, ne renderelje újra. Érdemes drága megjelenítésű, sokszor hívott komponenseken alkalmazni.
- **useMemo**: ha egy számítás nagyon drága (pl. óriási lista szűrése), cache-eljük az eredményt — csak akkor számítsuk újra, ha a bemenetek megváltoztak.
- **useCallback**: ha egy függvényt props-ként adunk át, és fontos, hogy a referenciája stabil maradjon (pl. React.memo-val védett gyermeknek).

Kódfelosztás és lusta betöltés

A probléma

Nagy alkalmazásoknál a teljes JavaScript csomag egyszerre töltődik be — ez lassítja az első megjelenést.

- **React.lazy + Suspense**: az egyes oldalakat/modulokat csak akkor töltjük be, amikor a felhasználó ténylegesen odanavigál.
- **Lista virtualizáció** (pl. `react-window`): ha több ezer listaelemet kell megjeleníteni, csak a látható elemeket rendereljük — a többi DOM-ban sem szerepel.

Miért teszteljünk?

Az alapmotiváció

A tesztek nem csak hibákat fednek fel — **dokumentálják a szándékolt viselkedést** és biztosítékot adnak arra, hogy jövőbeli módosítások nem törnek el meglévő funkciókat. Refaktorálás és csapatmunka esetén különösen értékes.

Tesztelési szintek

- **Unit teszt**: egyetlen függvény, hook vagy logikai egység viselkedését ellenőrzi, függetlenül a többitől. Gyors, izolált.
- **Integrációs teszt**: több komponens vagy réteg együttműködését vizsgálja — pl. egy form elküldésének teljes folyamata, beleértve az állapotkezelést.
- **E2E teszt** (End-to-End): egy valódi böngészőben szimulál felhasználói lépéseket — pl. Playwright-tal. Lassabb, de a legvalóságosabb visszajelzést adja.

Ajánlott eszközök

- **Vitest**: modern, Vite-alapú tesztfuttató — gyors, könnyen konfigurálható, ES module-barát.
- **Jest**: régebben az iparági standard, nagyon elterjedt, rengeteg dokumentáció.
- **React Testing Library**: komponenseket *felhasználói szemszögből* tesztel — nem implementációs részleteket, hanem azt, amit a felhasználó lát és tehet.
- **Mock Service Worker (MSW)**: API válaszokat szimulál hálózati szinten — nem kell valódi szerver a tesztekhez.

Mit érdemes tesztelni?

- **Kritikus folyamatok**: regisztráció, bejelentkezés, fizetés, jogosultság-ellenőrzés.
- **Hibaállapotok**: mi történik, ha a hálózat megszakad, a szerver hibát ad, az input érvénytelen?
- **Szélsőértékek (edge case)**: üres lista, nagyon hosszú szöveg, speciális karakterek.
- **Hozzáférhetőség**: a kritikus UI elemek megfelelő ARIA szerepkörrel és felirattal rendelkeznek-e?

Mi az a TypeScript?

JavaScript + típusok

A TypeScript a JavaScript egy **típusos kibővítése**: szintaktikailag ugyanolyan, de lehetővé teszi, hogy megmondjuk, egy változó szám-e, szöveg-e, vagy éppen egy konkrét adatstruktúra. A böngésző ezután sem érti — fordításkor sima JavaScript-té alakul.

Miért TypeScript Reacthez?

- **Korai hibajelzés**: a szerkesztő azonnal jelzi, ha rossz típusú props-ot adunk át egy komponensnek — nem kell a böngészőben futtatni.
- **Öndokumentáló kód**: a props típusai rögtön megmondják, mit vár a komponens — nincs szükség külön kommentre.
- **Karbantarthatóság**: refaktoráláskor a fordító jelzi, hol kell módosítani — nagyobb változtatások biztonságosabbak.
- Kis projekten is megéri a tanulási görbéje, mert a hosszabb távú haszon jelentős.

Mi az a Next.js?

React + szerver

A Next.js egy teljes React keretrendszer: beépített routing, szerveroldali renderelés (SSR), statikus oldalak generálása (SSG), képoptimalizálás, és más funkciók — mindez konfiguráció nélkül.

- **SSR (Server-Side Rendering)**: a szerver minden kérésnél rendereli az oldalt — friss adatok, jó SEO.
- **SSG (Static Site Generation)**: build-időben generál HTML-t — villámgyors, CDN-ről tálalható.

Mikor melyiket válasszuk?

- **Vite + React (SPA)**: admin felületek, dashboardok, belső eszközök — ahol a SEO nem kritikus, a bejelentkezés után töltődik be a tartalom.
- **Next.js**: publikus weboldalak, e-commerce, blog, dokumentáció — ahol a keres'optimalizálás és a gyors els' betöltés fontos.
- **Mindkét esetben ajánlott**: TypeScript, lint, automatikus tesztek, CI/CD csővezeték.

Mi történik a build folyamatban?

Build = el'készítés az éles kiadásra

Fejlesztés közben a kód sokszor nem optimális formában van — sok fájl, olvasható változónevek, fejlesztői segédletek. A build folyamat ezeket átalakítja böngészőbarát, gyors formára.

- **Minifikálás**: felesleges szóközök, kommentek eltávolítása, változónevek rövidítése — kisebb fájlméret.
- **Tree-shaking**: a nem használt kód kiszűrése a végleges csomagból.
- **Kódfelosztás**: az alkalmazás több kisebb fájlra bontva töltődik — csak ami kell, töltődik le.

Környezeti változók

- Az alkalmazásnak fejlesztési és éles környezetben más beállításokra van szüksége (API URL, kulcsok stb.).
- Ezeket **.env fájlokban** tároljuk — a `.env.local` soha nem kerül Git-be (érzékeny adatok).
- Vite-ban a `VITE_` előtagú változók kerülnek a kliensoldali kódba — a többi csak szerveroldalon érhető el.
- **Figyelem**: titkos kulcsokat soha ne tegyük a kliensoldali kódba — bárki láthatja!

Deploy lehet'ségek

- **Vercel**, **Netlify**: egy Git push-ra automatikusan deployol, ingyen elérhető próbaprojektekhez — a legegyszerűbb indulás.
- **GitHub Pages**: statikus oldalakhoz, nyílt forráskódú projektekhez kényelmes.
- **Saját szerver / CDN**: Nginx-szel statikus fájlokat tálalunk; fontos az SPA fallback beállítása (minden URL-t az `index.html`-re irányítunk, a React Router veszi át).

Biztonsági alapok

- **XSS (Cross-Site Scripting) elleni védelem:** a React alapból escape-eli a JSX-ben megjelenített szövegeket — a `dangerouslySetInnerHTML` viszont kikerüli ezt, csak tisztított adattal szabad használni.
- **Auth token kezelés:** a JWT-t ne `localStorage`-ban tároljuk — JavaScript elérheti. HTTP-only cookie biztonságosabb, mert JavaScriptből nem olvasható.
- **Hibamonitorozás:** éles alkalmazásban szükséges eszköz (pl. Sentry), hogy a felhasználóknál fellépő hibákról értesüljünk.

UI komponens könyvtárak

Mire valók?

Kész, előre megírt és stílusozott UI komponensek gyűjteményei — gombok, táblázatok, modalok, értesítések stb. Nem kell nulláról megírni a vizuális elemeket.

- **MUI (Material UI)**: Google Material Design stílus, nagyon széleskörű komponenskészlet.
- **Headless UI**, **Radix UI**: csak a viselkedési logikát adják, a megjelenést teljes egészében mi írjuk — maximális szabadság, Tailwind-del kombinálható.
- **shadcn/ui**: Radix + Tailwind alapú, másolható komponensek — egyre népszerűbb.

Állapot és adatkezelés ökoszisztéma

- **Redux Toolkit**: a Redux modern, kevésbé terjengős változata. Komplex állapotlogikához és nagy csapatoknak ideális.
- **Zustand**: egyszerű, minimális API-jú globális store. Kis és közepes projekteknél kiváló alternatíva.
- **Jotai**: atomi állapotkezelés — az állapot apró, egymásra épülő atomokból áll. Fine-grained reaktivitáshoz jó.
- **TanStack Query**: szerverállapot kezelésére specializált — cache, háttérfrissítés, újrapróbálkozás.
- **Zod**: típusbiztos adatvalidáció sémákkal — API válaszok, form adatok ellenőrzéséhez.

Összefoglalás — a React ökoszisztéma

- **Alap**: komponensek, props, state, hookok, JSX.
- **Interakció**: eseménykezelés, űrlapok, navigáció.
- **Adatok**: useEffect, Axios / TanStack Query, Context.
- **Minőség**: TypeScript, tesztelés, teljesítmény.
- **Kiadás**: build, deploy, biztonság, monitorozás.

Tanács kezdőknek

Ne akarjuk egyszerre megtanulni az összes eszközt. Kezdjük az alapokkal (komponens, state, effect), és szükség szerint bővítsük a tudást.