Programozási Projekt

Weblap Debreceni Cégnek - Oknoplast

OKTÁV Szakképzési Iskola
Szoftverfejlesztő és tesztelő

Kovács Kinga
Szeles Viktor
Kabai Tibor
2025 Szeptember

Bevezetés

Ez a projekt az Oknoplast Debrecen, egy nyílászárókkal foglalkozó vállalkozás számára készült, ahol a meglévő, elavult honlap modernizálása és funkcionális bővítése volt a cél. A projekt fő motivációja egy olyan professzionális, reszponzív weboldal létrehozása volt, amely hatékonyan támogatja az ügyfélszerzést, részletesen bemutatja a termékeket és szolgáltatásokat, valamint leegyszerűsíti az árajánlatkérési folyamatokat. Célunk egy vizuálisan vonzó és felhasználóbarát platform kialakítása volt, amely modern technológiai alapokon nyugszik és minden eszközön kiváló felhasználói élményt nyújt.

A megvalósítás során a **Next.js modern React framework-et** használtuk, kihasználva annak fejlett routing rendszerét és típusbiztonsági előnyeit. A projekt egy teljeskörű megoldást nyújt, amely magában foglalja a Supabase adatbázis integrációt a termékek és adatok kezeléséhez, egy működő email kapcsolati űrlapot, dinamikusan generált termék aloldalakat, valamint egy dedikált admin felületet a tartalom egyszerű menedzseléséhez. A **Tailwind CSS** alkalmazásával gyors és reszponzív stílusozást érhettünk el, garantálva a honlap optimális megjelenését mobil, tablet és desktop eszközökön egyaránt.

Tervezés

Projekt kiválasztása

A projekt témájának kiválasztásakor fontos szempont volt számunkra, hogy olyan feladatot találjak, amely nem csupán elméleti, hanem a gyakorlatban is hasznosítható eredményt hoz létre. Kezdetben egy játékfejlesztésen gondolkodtunk, mivel ez tűnt egyszerűbb és szórakoztatóbb kiindulópontnak. Azonban hamar rájöttünk, hogy egy játék készítése önmagában nem biztosít kellő szakmai kihívást, és gyakorlati haszna is korlátozott. Ezért végül egy olyan témát kerestünk, amely közvetlenül kapcsolódik egy valós igényhez, és amelyen keresztül a programozás, a webfejlesztés és a modern informatikai megoldások alkalmazása jól bemutatható.

A döntésünk végül az Oknoplast Debrecen nyílászárókkal foglalkozó vállalkozás új honlapjának elkészítésére esett. A cég jelenlegi weboldala elavult, nem tükrözi a mai elvárásokat sem megjelenésben, sem funkciókban. Ez egyértelmű igényt teremt egy modern, letisztult és felhasználóbarát honlap elkészítésére, amely segíti a vállalkozás online jelenlétét, ügyfélszerzését és tájékoztatási folyamatait.

A projekt fő célja egy olyan honlap megtervezése és elkészítése, amely tartalmazza:

- A vállalkozás részletes bemutatását.
- A nyílászáró termékek és szolgáltatások ismertetését.

- Egy áttekinthető és frissített árlistát.
- Kapcsolatfelvételi lehetőséget.
- Valamint olyan kiegészítő funkciókat, amelyek megkönnyítik a felhasználók számára az információk elérését.

Ez a projekt tehát több szempontból is hasznos számunkra: valódi igényre épül, így a végeredmény gyakorlati értéket képvisel, másrészt pedig alkalmat ad arra, hogy elmélyítsük programozási és webfejlesztési ismereteinket. A honlap elkészítése során gyakorlatot szerezni a frontend és backend fejlesztésben, az adatkezelésben, valamint a felhasználói élmény tervezésében is.

Összességében azért választottuk ezt a témát, mert így egy olyan projektet valósíthatók meg, amelynek kézzelfogható eredménye van, hozzájárul egy működő vállalkozás fejlődéséhez, és egyben lehetőséget biztosít számunkra a szakmai fejlődésre és az informatikai készségek fejlesztésére.

Felhasználói igények felmérése

Az ügyfelünk az Oknoplast Debrecen vezetője, aki kifejezetten igényli, hogy vállalkozása számára egy modern, professzionális és reszponzív céges honlap készüljön. A cég jelenlegi weboldala elavult, nem tükrözi a mai elvárásokat sem megjelenésben, sem funkciókban. Emiatt egy új, naprakész és jól strukturált weboldal létrehozása vált szükségessé, amely egyszerre szolgál információforrásként, ügyfélszerzési felületként és kommunikációs csatornaként.

Az első fázisban igényfelmérést végeztünk az ügyféllel történt egyeztetés keretében. Ennek során világossá vált, hogy a honlap elsődleges célja a leendő ügyfelek megszólítása, a termékek részletes bemutatása, valamint az ajánlatkérés folyamatának leegyszerűsítése.

Emellett fontos, hogy a weboldal vizuális arculata harmonizáljon az Oknoplast márka nemzetközi színvonalával, ugyanakkor kifejezetten a debreceni kirendeltséget reprezentálja.

A honlapnak különböző eszközökön egyaránt hibátlanul kell működnie. Mivel a potenciális ügyfelek egy része mobileszközről (telefon, tablet) böngészi majd az oldalt, elengedhetetlen a reszponzív tervezés. Ugyanakkor a cég üzleti partnerei és irodai felhasználói gyakran nagyobb képernyőn (laptop, asztali számítógép, 4K monitor) fogják látogatni az oldalt, ezért a weboldalnak minden felbontásban esztétikusnak és könnyen kezelhetőnek kell maradnia. Az ügyféllel folytatott megbeszélések alapján a következő funkciókra mutatkozott egyértelmű igény:

• **Kapcsolatfelvételi űrlap:** Egyszerű és gyors mód az érdeklődők számára a cég elérésére, üzenetküldésre.

• **Termékekhez kapcsolódó aloldalak:** Külön al-oldal minden terméktípusnak (például ablakok, redőnyök, zsaluk, bejárati ajtók), egyedi Landing Page jellegű kialakítással, részletes leírással, fotókkal és paraméterekkel.
• **Keresőoptimalizált tartalom (SEO):** A honlap struktúrája és szövegezése úgy kerüljön kialakításra, hogy a vállalkozás könnyen megtalálható legyen a Google-ben a releváns kulcsszavakra („ablak Debrecen”, „Oknoplast árak”, „műanyag nyílászáró Debrecen” stb.).
• **Multimédiás tartalmak:** Képgalériák, referenciamunkák, esetleg videós bemutatók, amelyek vizuálisan is megerősítik a termékek minőségét.

A honlap várható használati forgatókönyve az ügyfél által vázolt marketingstratégia alapján a következőképpen írható le:

1) 1. A cég Facebook- és Google-hirdetéseket futtat, amelyek a potenciális vásárlókat a honlapra irányítják.
2) 2. A látogató megérkezik a weboldalra, ahol áttekinthetően megtalálja a keresett terméket (például ablak).
3) 3. A termék oldalról továbblépve a látogató egyszerűen
4) 4. Az ügyfélkapcsolat ezt követően e-mailen keresztül folytatódik, amely már személyre szabott ajánlatadást és kommunikációt tesz lehetővé.

Ez a rendszer nemcsak a felhasználói élményt javítja, hanem az ügyfél számára is mérhető üzleti előnyt hoz, hiszen a honlap így közvetlenül támogatja az ügyfélszerzést és a bevételnövelést.

Összességében a felhasználói igények felmérése világossá tette, hogy az új honlapnak nem pusztán információforrásnak kell lennie, hanem komplex marketing és értékesítési eszköznek, amely korszerű technológiai háttérrel, modern megjelenéssel és felhasználóbarát funkcionalitással szolgálja az Oknoplast Debrecen üzleti érdekeit.

Vázlatolás

Az ügyféllel történt igényfelmérés és a funkcionális követelmények tisztázása után a következő lépés a honlap struktúrájának és vizuális elrendezésének vázlatolása volt. Ez a fázis kulcsfontosságú ahhoz, hogy a fejlesztés előtt egyértelmű képünk legyen a weboldal felépítéséről és a kulcsfontosságú interakciókról.

Honlap felépítése és vizuális elemek:

A honlap vázlatolása során figyelembe vettük a szükséges linkeket, al-oldalakat, szöveges tartalmakat és képeket, amelyeket egy hipotetikus "excel táblában" rögzítettük. A vizuális elrendezést illetően a következő főbb szakaszok kerültek meghatározásra, amelyek az oldal legfelső pontjától a lábléc-ig terjednek:

- **Legfelső Kapcsolati Sáv:** Ez a sáv a honlap tetején helyezkedik el, és tartalmazza a legfontosabb elérhetőségeket, például telefonszámot és e-mail címet, amelyek azonnal láthatóak a látogatók számára.
- **Menü Sáv:** A kapcsolati sáv alatt található a fő navigációs menü, amely a honlap legfontosabb oldalaira mutat. A főbb menüpontok várhatóan a következők lesznek: Főoldal, Termékeink, Szolgáltatások, Rólunk, Kapcsolat.
- **Üdvözlő Banner** (Hero Szekció): Ez a honlap egyik legkiemelkedőbb része, amely egy nagy, látványos képet tartalmaz prémium minőségű nyílászárókról, egy figyelemfelkeltő hívószóval (pl. "Minőség, megbízhatóság, innováció"), valamint egy cselekvésre ösztönző gombbal (pl. "Tekintse meg kínálatunkat!").
- **Bemutatkozás Szekció:** Rövid bevezetés a cégről, annak értékeiről és küldetéséről. A cég története, értékei (minőség, megbízhatóság, innováció, ügyfélközpontúság) és csapata is megjelenhet itt.
- **Kiemelt Termékek Szekció:** Rövid ízelítő a cég termék palettájából, bemutatva a legnépszerűbb vagy legfontosabb nyílászárókat. Ez magában foglalhatja az ablakokat, ajtókat, redőnyöket és egyéb kiegészítőket.
- **Szolgáltatások Szekció:** Az Oknoplast Debrecen által nyújtott szolgáltatások bemutatása, mint például az ingyenes felmérés és tanácsadás, egyedi méretre gyártás, szakértő beépítés és garancia.
- **Visszajelzések Szekció:** Ügyfélvélemények és ajánlások, amelyek növelik a cég megbízhatóságát és hitelességét.
- **Kapcsolati Rész:** A legfontosabb elérhetőségi adatok (telefon, e-mail, cím) és egy rövid, de hatékony kapcsolati űrlap.
- **Lábléc (Footer):** Az oldal alján található, általában tartalmazza a navigációs menü másolatát és egyéb jogi vagy kiegészítő információkat.

Részletes terv

Az igényfelmérés és a vázlatolás fázisát követően, a projekt megvalósításának alapjaként egy részletes tervet dolgoztunk ki. Ez a terv képezi a Figma-ban elkészítendő grafikai terv (design) alapját, pontosítva a főoldal.

1. Főoldal Részletes Terve (Figma Design Alapja)
A főoldal tervezése során az alábbi szekciókat és azok tartalmát pontosítottuk, figyelembe véve a vázlatban megfogalmazottakat és az Oknoplast Debrecen márka elvárásait:

• Legfelső Kapcsolati Sáv:

- *Elhelyezkedés:* Az oldal legtetején, fixen.

◦ *Tartalom:* Jól látható telefonszám (+36 30 456 0621) és e-mail cím (poliprofil@oknoplast.com.pl).

◦ *Cél:* Azonnali és könnyű kapcsolatfelvételi lehetőség biztosítása.

• Menü Sáv (Header):

◦ *Cégnév/Logó:* Oknoplast Debrecen felirat és logó.
◦ *Mottó:* Lásd a különbséget!.
◦ *Navigációs menüpontok:*

▪ **Főoldal** `(/)`
▪ **Termékeink** `(/termekeink)`
▪ **Szolgáltatások** `(/szolgaltatasok)`
▪ **Rólunk** `(/rolunk)`
▪ **Kapcsolat** `(/kapcsolat)`

◦ *Funkcionalitás*: A menü reszponzív kialakítást kap, mobilnézetben hamburger menüként funkcionál. A mobile menü linkre kattintáskor automatikusan bezáródik.

• Üdvözlő Banner (Hero Szekció):

◦ *Vizuális elem:* Nagy, látványos, prémium minőségű nyílászárókat bemutató kép. Például a `oknoplast-windows.jpg` vagy `okna-dom-parterowy-oknoplast.jpg` képekhez hasonló.
◦ *Hívószó:* Minőség, megbízhatóság, innováció.
◦ *Cselekvésre ösztönző gomb:* Tekintse meg kínálatunkat! **• Bemutatkozás Szekció (Rólunk):**

◦ *Tartalom:* Rövid bevezetés a cégről, annak 15+ éves történetéről, küldetéséről és értékeiről (minőség, megbízhatóság, innováció, ügyfélközpontúság).
◦ *Kiemelt adatok:* 15+ Év Tapasztalat, 5000+ Elégedett Ügyfél, 10000+ Beépített Ablak, 100% Elégedettség vizuális megjelenítése.
◦ *CTA:* Gomb az Összes szolgáltatás vagy Tudjon meg többet rólunk oldalra.

• Kiemelt Termékek Szekció:

◦ *Tartalom:* Rövid ízelítő a cég főbb termék palettájából: Ablakok, Ajtók, Redőnyök és Kiegészítők. Minden termék típushoz egy-egy kép és rövid leírás tartozik, linkkel a részletes termék al-oldalra.

• Szolgáltatások Szekció:

◦ *Tartalom:* Az Oknoplast Debrecen által nyújtott legfontosabb szolgáltatások bemutatása, kiemelve az Ingyenes Helyszíni Felmérés, Professzionális Beépítés, Megbízható Szállítás és Logisztika, Hosszú Távú Garancia, Gyors Ügyintézés és Ügyfélszolgálat.
◦ *Cél:* Az ügyfelek bizalmának erősítése és a teljes körű szolgáltatás bemutatása.

• Visszajelzések Szekció:

◦ *Tartalom:* Valós ügyfélvélemények és ajánlások (pl. Google vélemények).
◦ *Cél:* A cég megbízhatóságának és hitelességének növelése.

• Kapcsolati Rész:

◦ Elérhetőségi adatok: Telefonszám, e-mail, debreceni üzlet címe, nyitvatartás.
◦ Kapcsolati űrlap: Egyszerű form, név, e-mail, tárgy, üzenet mezőkkel, üzenetküldés gombbal. Az űrlap API-n keresztül küld e-mailt a cégnek a Nodemailer és Gmail App Password használatával.
◦ Térkép: Az üzlet helyének megjelenítése, útvonaltervező linkkel.

• Lábléc (Footer):

◦ *Tartalom:* A fő navigációs menüpontok ismétlése, jogi információk (pl. adatvédelmi nyilatkozat linkje), szerzői jogi megjegyzések.

3. Új nyílászáró és kiegészítők hozzáadása:

◦ A modul lehetővé teszi több nyílászáró hozzáadását egyazon ajánlatkéréshez.
◦ A felhasználó választhat különböző kiegészítőket, mint például szúnyogháló, párkány, vagy árnyékolók (redőny, zsaluzia).

4. Szolgáltatások kiválasztása:

◦ A folyamat utolsó lépéseiben a felhasználó ki választhatja a szükséges szolgáltatásokat, például helyszíni felmérést vagy a professzionális beépítést.

5. Összegzés és Kapcsolatfelvétel:

◦ A felhasználó megadja kapcsolati adatait (név, e-mail, telefonszám).
◦ A rendszer automatikusan e-mailt küld a felhasználónak és az Oknoplast Debrecennek is, összefoglalva a megadott adatokat, kiválasztott termékeket, kiegészítőket és szolgáltatásokat. Ez az e-mail lesz az alapja a személyre szabott árajánlat elkészítésének.

3. Funkciók Pontosítása és Leírása

A honlaphoz tervezett főbb funkciók, részletes leírásokkal:

• Reszponzív Design és Mobilbarát Felület:

◦ A weboldal minden eszközön (mobiltelefon, tablet, laptop, 4K monitor) tökéletesen és esztétikusan jelenik meg és működik.
◦ A fejlesztés során a mobile first megközelítést alkalmazzuk, Tailwind CSS breakpoint-eket `(md: prefix)` használva az adaptív elrendezéshez.

◦ Modern CSS layout technikák, mint a Flexbox és Grid biztosítják a rugalmas elrendezést.

• Keresőoptimalizált Tartalom (SEO):

◦ A honlap struktúráját és tartalmát úgy tervezzük meg, hogy a Google keresőmotorokban a releváns kulcsszavak (pl. „ablak Debrecen”, „Oknoplast árak”, „műanyag nyílászáró Debrecen”) könnyen megtalálható legyen.
◦ Ez magában foglalja a megfelelő metaadatok, címsorok és a tartalom optimalizálását.

• Multimédiás Tartalmak és Képgaléria:

◦ A termékoldalak és a főoldal minőségi képgalériánk tartalmaznak a nyílászárókról és referencia munkákról.
◦ `Next.js` Image komponenst használunk a képek optimalizált betöltésére.

• Adatbázis-kezelés (Supabase):

◦ A termékek, azok specifikációi és egyéb dinamikus adatok tárolására és kezelésére egy PostgreSQL alapú Supabase adatbázist alkalmazunk.
◦ Az adatbázis tábla struktúráját SQL scriptekkel definiáljuk `(products tábla)`.
◦ Az adatok lekérdezése `server-side` és `client-side` Supabase klienseken keresztül történik.

• Email Kapcsolati Form és Értesítések:
◦ A Kapcsolat oldalon egy űrlap áll rendelkezésre üzenetek küldésére.
◦ Az e-mail küldést Nodemailer és egy Next.js API Route kezeli, hitelesítéshez a Gmail App Password-öt használva.

• Dinamikus Termék Al-oldalak:

◦ A Next.js App Router dinamikus routing funkcióját alkalmazzuk, így minden egyes termék külön al-oldalon (pl. `/termekeink/mueanyag-ablakok` vagy `/termekeink/[slug]`) jelenhet meg részletes leírással, képekkel és specifikációkkal.
◦ A termékek adatbázisból töltődnek be, a `generateStaticParams f`üggvény segítségével statikusan generálva az al-oldalakat.

• Felhasználói Felület (UI) Komponensek:

◦ A modern és egységes megjelenés érdekében a `shadcn/ui` komponens könyvtárat integráljuk.
◦ A stílusozás a Tailwind CSS keretrendszerrel történik, egységes színséma és tipográfia alkalmazásával.

• Képfeltöltés és Kezelés (Supabase Storage):

◦ Egy dedikált oknoplast-images nevű Supabase Storage bucket szolgál a termékfotók tárolására.
◦ Az admin felületen keresztül lehetőség lesz képek feltöltésére, törlésére és termékekhez rendelésére, unique fájlnevek generálásával.
◦ A képek URL-jei a `getPublicUrl` függvénnyel generálódnak, megfelelő Storage policy-k biztosítják az hozzáférést.

• Admin Felület a Termékek Kezeléséhez:

◦ Egy jelszóval védett adminisztrációs felületet fejlesztünk `(/admin útvonalon)`, amely lehetővé teszi a termékek kezelését.
◦ Autentikáció: Egyszerű jelszó alapú bejelentkezés LocalStorage alapú session kezeléssel.
◦ CRUD Műveletek: Az admin felületen keresztül lehetőség van új termékek hozzáadására, meglévők szerkesztésére, megtekintésére és törlésére a Supabase adatbázisban.
◦ Validáció és Hibakezelés: A beviteli mezők, különösen a JSON alapú műszaki adatok és kép URL-ek validálása megvalósul, felhasználói visszajelzéssel a hibákról.
Ez a részletes terv szolgált alapul a projekt kivitelezési fázisának, amely biztosítja, hogy minden korábban meghatározott igény és funkcionalitás megvalósuljon egy modern, hatékony és felhasználóbarát weboldal formájában.

Cél összegzése

A tervezési fázis lezárultával a projekt célja egyértelműen meghatározásra került, és minden előkészület megtörtént a megvalósításhoz. Az Oknoplast Debrecen számára készülő új, modern és reszponzív céges honlap fő célja, hogy a vállalkozás online jelenlétét erősítse, potenciális ügyfeleket szólítson meg, részletesen bemutassa a termékeknek.

Az ügyféllel folytatott igényfelmérés során pontosan tisztáztuk a szükséges funkciókat és a felhasználási forgatókönyveket. Ennek értelmében a honlapnak tartalmaznia kell:

1. • Egy felhasználóbarát email kapcsolati űrlapot.
2. • Dinamikus termék al-oldalakat (Landing Page jellegű kialakítással minden termék típushoz).
3. • Keresőoptimalizált tartalmat (SEO) a Google-ben való könnyű megtalálhatóság érdekében.
4. • Multimédiás tartalmakat (képgalériák, referenciák) a termékek vizuális megerősítéséhez.
5. Ezenkívül a honlapnak minden eszközön (mobiltelefon, tablet, laptop, 4K monitor) reszponzívan és hibátlanul kell működnie.

Az oldal vizuális és funkcionális felépítését részletes vázlatok és tervek (melyek a Figma design alapját képezik) rögzítették, pontosan definiálva a főoldali szekciókat (például

legfelső kapcsolati sáv, menü, hero banner, bemutatkozás, kiemelt termékek, szolgáltatások, visszajelzések, kapcsolati rész, lábléc).

Ezáltal minden szükséges információ és tervezési minta rendelkezésre áll a projekt sikeres kivitelezéséhez.

Technológia Kiválasztása

Keretrendszer kiválasztása: React és `Next.js`

Mivel a rendszernek sok funkciót kellett biztosítania és megbízhatónak kellett lennie, egy elterjedt és egyszerű keretrendszer használata vált szükségessé.

A választás a Reactra és a `Next.js-re` esett a következő indokok miatt:

1. • **Modern React framework:** A `Next.js` egy modern React keretrendszer, amely számos hasznos funkciót biztosít `"out-of-the-box".`
2. • **Egyszerűség és elterjedtség:** Hasonló a tiszta JavaScripthez, egyszerűnek mondható, és széles körben elterjedt.
3. • **Dokumentáció és támogatás:** Könnyen találni az interneten bemutatókat és segítséget olyan kulcsfontosságú feladatokhoz, mint az email küldés, adatbázishoz való kapcsolódás és termékek kezelése.
4. • **Beépített funkciók és eszközök:** A `Next.js` projekt telepítésekor számos előre konfigurált eszközt és beállítást kapunk, amelyek felgyorsítják a fejlesztést és biztosítják a kód minőségét:

◦ *TypeScript:* A típus biztonság garantálása érdekében. Bár a kezdeti telepítés után voltak TypeScript hibák a `@/ import alias` felismerésével kapcsolatban, ezeket a `tsconfig.json` fájl megfelelő beállításával orvosolták. Később a React típusokkal kapcsolatos hibák megoldására további típusokat kellett telepíteni.

◦ *Tailwind CSS:* Gyors stílus-ozáshoz. Habár kezdetben problémák adódtak a Tailwind CSS működésével, a `tailwind.config.ts` fájl tartalom útvonalainak beállításával sikerült megoldani. Később a `Shadcn/ui` komponensek használatához is frissíteni kellett ezt a fájlt. A globális stílusok az `app/globals.css` fájlban importálódik.

◦ *ESLint:* Kódminőség ellenőrzéséhez.

◦ *App Router:* A Next.js 13+ új routing rendszerét használták, ami az oldalak definiálását az `app/` mappában teszi lehetővé a korábbi `pages/` mappa helyett. Ez a dinamikus route-ok `(pl. termék aloldalak)` beállításánál is kulcsfontosságú volt.

◦ *Import alias* `(@/ prefix)`: A fájlok importálásának egyszerűsítésére.

Egyéb fontos technológiák és megoldások:

• ***UI komponensek*** **`(Shadcn/ui):`** A szép és egységes felhasználói felület érdekében telepítették a `Shadcn/ui-t`. Ez létrehozta a components.json fájlt és a szükséges konfigurációt.

• ***Adatbázis-integráció (Supabase):*** A hardcoded termékek nem voltak praktikusak. Ezért a projekt átállt egy adatbázis alapú megoldásra, a Supabase-re. A Supabase egy modern, `open-source` Firebase alternatíva, amely PostgreSQL adatbázist és valós idejű funkciókat biztosít. A Supabase projekt létrehozásával, csomagok telepítésével, környezeti változók beállításával, valamint kliens-oldali `(lib/supabase/client.ts)` és szerver-oldali `(lib/supabase/server.ts)` kliensek konfigurálással integrálták a rendszert. Később a Supabase Storage-t is beállították a képek tárolásához.

• ***Email küldés (Nodemailer és Gmail App Password):*** A kapcsolati űrlap implementálásához egy API route-ot hoztak létre az email küldéshez. Ehhez telepítették a Nodemailer-t és beállították a szükséges környezeti változókat. A Gmail App Password használata tette lehetővé a Gmail fiókon keresztül történő email küldést.

• ***Képkezelés (Next.js Image komponens és Supabase Storage):*** A képek optimalizált betöltéséhez a Next.js Image komponenst használták. A képek feltöltését és kezelését a Supabase Storage segítségével valósították meg, létrehozva egy `oknoplast-images` bucket-et.

• ***Lucide React ikonok:*** Ikonok megjelenítésére szolgáló könyvtár (bár kezdetben probléma volt a betöltésével).

Miért nem más technológiák?

A forrás megemlíti, hogy a projektválasztás egy jó kihívás volt, és utal arra, hogy fontolóra vették más alternatívákat is (pl. sima CSS, HTML, JS, vagy WordPress), de a `Next.js` mellett döntöttünk. Bár a pontos okok nincsenek részletezve a forrásban, a korábbi indokokból (megbízhatóság, funkciók komplexitása, modern elvárások) kikövetkeztethető, hogy a `React/Next.js` robusztusabb, skálázhatóbb és a fejlesztő számára nagyobb kihívást jelentő, de egyben több lehetőséget kínáló megoldásnak bizonyult egy ilyen modern céges weboldal megvalósításához.

Részletes kivitelezés

Projekt telepítése és kezdeti beállítások

Amikor eljött az ideje, hogy belevágjunk a kurzus záró projektjébe, az első dolgunk az volt, hogy kiválasztottuk, milyen technológiával dolgozzunk. Mivel hallottuk, hogy a `Next.js` egy modern React keretrendszer, és sok mindent tud "`out-of-the-box`", úgy gondoltuk, ez jó választás lesz.

1. Next.js projekt létrehozása

Az első igazi lépés egy új `Next.js` alkalmazás létrehozása volt. Ezt egy paranccsal tettünk meg a terminálban. Ez a parancs rögtön beállított néhány fontos dolgot:

- **TypeScript:** Ezt a típust biztonság miatt választottunk, hogy kevesebb hibával találkozzunk menet közben. Gondoltuk, így tisztább lesz a kód is.
- **Tailwind CSS:** A gyors stílusozáshoz javasolták, és tényleg egyszerűnek tűnt.
- **ESLint:** Ez segít a kód minőségének ellenőrzésében.
- **App Router:** Ez egy új út választási rendszer volt a Next.js 13+-ban, és szerettük volna kipróbálni.
- **Import alias:** A fájlok importálásához `@/` előtagot kaptuk, ami azt jelenti, hogy könnyebb hivatkozni a fájlokra, nem kell bonyolult relatív útvonalakat írni.

2. Projekt struktúra és függőségek

A telepítés után létrejött egy alapvető mappastruktúra. Ezt nagyjából úgy hagytuk, ahogy volt, mert a `Next.js` elég jól rendszerezi magát. A `package.json` fájlban láttuk, hogy milyen sok csomagot telepített fel a rendszer automatikusan. Ez jó volt, mert nem nekünk kellett kézzel hozzáadnunk mindent.

3. Fejlesztői szerver indítása

Ezután elindítottuk a fejlesztői szervert egy egyszerű paranccsal. Ez a `http://localhost:3000` címen tette elérhetővé a frissen létrehozott, még üres honlapot. Izgatottak voltunk, hogy láthatjuk, ahogy működésbe lép.

Első problémák és megoldások
Persze, nem volt minden tökéletes azonnal. Ahogy az lenni szokott, belefutottunk pár hibába:

- **TypeScript hibák:** Az egyik első dolog az volt, hogy a TypeScript nem ismerte fel a `@/ import alias-t`. Kicsit bosszantó volt, mert azt hittük, ez automatikus. Meg kellett találnunk a `tsconfig.json fájlt`, és ott kellett beállítani, hogy tudja, mit jelent ez az előtag. Kisebb fejtörés volt.

- **Tailwind CSS nem működött:** A Tailwind CSS sem úgy működött, ahogy vártuk. Nem jelentek meg a stílusok. Rájöttünk, hogy a `tailwind.config.ts` fájlban be kellett

állítanunk a content útvonalakat, hogy a Tailwind tudja, melyik fájlokban keresse a használt osztályokat.
Ez is egy apróság volt, de elvett egy kis időt a kezdeti beállításokból.

Következő lépések.

Miután ezeket a kezdeti akadályokat legyőztük, a következő lépésünk az volt, hogy létrehozzuk az első oldalt. Ezzel akartuk kipróbálni, hogy a projekt alapjai tényleg rendben vannak-e, és minden beállítás megfelelően működik. Így mehet tovább a munka!

Első oldal létrehozása (Hello World)

Miután az előző lépésekben beállítottuk és elindítottuk a `Next.js` projektet, az első igazi feladatunk az volt, hogy létrehozzunk egy nagyon egyszerű Hello World oldalt. Ez azért volt fontos, hogy megbizonyosodjunk róla, minden rendben működik az alapokkal.

App Router használata A `Next.js` 13+-ban az App Router az új módja az oldalak kezelésének. Ez azt jelenti, hogy az oldalak az app/ mappában vannak, nem pedig a korábbi `pages/` mappában. Kicsit más volt, mint amit korábban tanultunk.

1. Főoldal létrehozása Az első oldalt az `app/page.tsx` fájlban hoztuk létre. Ez lett a honlap főoldala. Ide került a "Hello World" szöveg, amit látni akartunk.

2. Layout fájl beállítása Az `app/layout.tsx` fájlban definiáltunk az alapvető HTML struktúrát. Ez olyan, mint egy keret, ami körülveszi az összes oldalt.

Stílusozás Tailwind CSS-sel A globális stílusokat, beleértve a Tailwind CSS-t is, az `app/globals.css` fájlban importáltuk. Így tudtuk használni a Tailwind osztályait az oldalon.

Első tesztelés Nagyon izgatottak voltunk, hogy kipróbáljuk!

1. Szerver indítása: Elindítottuk a fejlesztői szervert.
2. Böngészőben ellenőrzés: Megnyitottuk a böngészőben a `http://localhost:3000` címet.
És láttuk is, hogy:

- A "Hello World" szöveg megjelent.
- A Tailwind CSS stílusok működtek (a szöveg kék volt és középre igazítva).
- A Google Fonts (Inter) betűtípus is betöltődött

.

Problémák és megoldások (mert nem minden ment simán)

• **Probléma:** Hot reload nem működött Egy idő után észrevettük, hogy ha módosítottuk a kódot, nem frissült azonnal az oldal a böngészőben. Kicsit megijedtünk, hogy valami komolyabb baj van.

Megoldás: Végül rájöttünk, hogy csak újra kellett indítani a fejlesztői szervert. Ezután már szépen működött a hot reload. Elég egyszerű hiba volt.

• **Probléma:** TypeScript hibák Kaptunk néhány TypeScript hibát a React típusokkal kapcsolatban. Ezek eleinte zavaróak voltak, mert nem tudtuk pontosan, mit hiányol.

Megoldás: A hibaüzenetek alapján telepíteni kellett a szükséges `@types/react` és `@types/react-dom csomagokat`. Utána eltűntek a hibák.

Következő lépések Most, hogy az alap Hello World oldal elkészült és működött, a következő lépés az volt, hogy létrehozzuk a weboldal alapvető layout struktúráját, beleértve a navigációs sávot és a láblécet. Ez már egy kicsit összetettebb feladatnak ígérkezett, de izgatottan vártuk.

Oldal layout készítése

Miután az előző részben sikerült a "Hello World" oldalt elindítani és láttuk, hogy az alapok működnek, a következő lépésünk az volt, hogy létrehozztuk a weboldal alapvető szerkezetét, vagyis a layout-ját. Gondoljatok csak bele, minden weboldalon van felül egy menü (header) és alul egy lábléc (footer), ami az összes oldalon megjelenik. Ezt kellett most elkészíteni.

Layout komponensek létrehozása A `Next.js-ben` az a jó, hogy feloszthatjuk a nagyobb részeket kisebb, önálló darabokra, amiket komponenseknek hívunk. Ezért külön-külön készítettünk el a fejlécet és a láblécet.

1. **Fejléc (Header) komponens:** Létrehoztunk egy `components/header.tsx` fájlt, és ebbe írtuk meg a felső navigációs sáv kódját. Ide került a cégnév és a menüpontok.
2. **Lábléc (Footer) komponens:** Hasonlóan, elkészítettük a `components/footer.tsx` fájlt is, ebbe pedig a honlap alján megjelenő információk kerültek.

Layout integrálása Miután a header és footer komponensek készen voltak, be kellett illeszteni őket a fő layout fájlba. Frissítettük az `app/layout.tsx fájl`t, hogy minden oldalt automatikusan tartalmazza a fejlécet és a láblécet. Ez a fájl az, ami "körbeöleli" az összes többi oldalt.

Stílusozás és reszponzív design Nagyon fontos volt, hogy a honlap jól nézzen ki minden eszközön, legyen az telefon, tablet vagy számítógép. Ehhez a Tailwind CSS-t használtuk, ami nagyon megkönnyíti a stílusozást.

• A `tailwind.config.ts` fájlban beállítottuk az alapvető színeket, hogy a weboldal egységes legyen.

• A reszponzív design-t is figyelembe vettük. Ez azt jelenti, hogy először a mobil nézettel foglalkoztunk (ezt hívják "mobile first" megközelítésnek), majd a `md: prefix`

segítségével állítottuk be, hogyan nézzen ki az oldal nagyobb képernyőkön, például tableten vagy asztali gépen.
• Használtunk a modern CSS technikákat is, mint a Flexbox és a Grid, amikkel szépen rendezhetők az elemek az oldalon.

Problémák és megoldások (mert nem minden ment simán, miért is menne?)

• **Probléma:** Lucide React ikonok nem működtek Szerettem volna szép ikonokat használni a menüben és máshol, ezért választottuk a Lucide React ikonokat. Azonban eleinte nem akartak megjelenni. Pár órás Google keresés és próbálkozás után végül sikerült beállítani őket, de bevaljuk, ez egy kicsit megizzasztott minket. Nem volt egyértelmű azonnal, mi hiányzik.
• **Probléma:** Mobil menü nem záródott be A mobil menü (amit a kis hamburger ikonra kattintva nyílik meg) szépen megjelent, de amikor rá-kattintottunk egy menüpontra, nem záródott be automatikusan. Ott maradt nyitva, ami egy kicsit zavaró volt a felhasználónak.
Megoldás: Végül rájöttünk, hogy hozzá kell adni egy onClick eseményt a linkekhez, ami bezárja a menüt, ha valaki rákattint. Ezt persze simán elfelejtettük először, mert annyira az alap funkciókra koncentráltunk. Tipikus kezdő hiba!

Következő lépések Most, hogy a fejléc és a lábléc is a helyén volt, és az alap layout stabilnak tűnt, a következő feladatunk az volt, hogy létrehozzuk a honlap főbb oldalait, mint például a "Termékeink" vagy a "Szolgáltatások" oldalakat. Ez már egy kicsit izgalmasabb, tartalmi résznek ígérkezett.

Főbb oldalak létrehozása

Miután az előző lépésekben elkészítettük a honlap alapvető keretét, a fejlécet és a láblécet, a következő nagy feladat az volt, hogy létrehozzuk a honlap főbb oldalait. Ezek azok az oldalak, amiken keresztül a látogatók ténylegesen böngészni fogják a tartalmat, megismerkednek a céggel és a termékekkel.

Oldalak tervezése Először is leültünk, és átgondoltuk, milyen oldalakra lesz szükségünk a projekt befejezéséhez. A következő főbb oldalakban gondolkodtuk:

- **Főoldal `(/)`** – Ez az, amit az ember először lát. Ide került egy "Hero" szekció (egy nagy, figyelemfelkeltő rész) és a termékek rövid bemutatója.
- **Termékeink `(/termekeink)`** – Itt lesz az összes termék listázása.
- **Szolgáltatások `(/szolgaltatasok)`** – A cég által nyújtott szolgáltatások leírása.
- **Blog `(/blog)`** – Itt lehet majd cikkeket olvasni, ha később lesznek.
- **Rólunk `(/rolunk)`** – A cég bemutatása, története, értékei.
- **Kapcsolat `(/kapcsolat)`** – Kapcsolati adatok és egy űrlap.

Főoldal fejlesztése A főoldal mindig a legfontosabb, úgyhogy ezzel kezdtük. Ezt is igyekeztünk darabokra bontani, hogy átláthatóbb legyen.

1. Hero komponens létrehozása: Először megcsináltuk egy `components/hero.tsx` fájlt, amibe a főoldal tetején látható, nagy, látványos részt raktunk. Ide került egy hívószó, gomb, stb., ami megfogja az embereket.

2. Termékek előnézet komponens: A főoldalra szerettük volna egy kis ízelítőt a termékekből, ezért létrehoztunk egy `components/products-preview.tsx` komponenst. Ez röviden bemutat pár terméket, amire aztán rá lehet kattintani.

3. Főoldal összeállítása: Végül az `app/page.tsx` fájlban összeraktuk a Hero és a Products Preview komponenseket, így lett egy teljes főoldal.

Termékeink oldal Utána következett a Termékeink oldal, ahol az összes terméket szerettük volna megjeleníteni.

1. Termékeink oldal létrehozása: Létrehoztunk az `app/termekeink/page.tsx` fájlt. Ez az oldal egyelőre csak az alapokat tartalmazta, később fogjuk feltölteni a valódi termék listával.

UI komponensek telepítése Hogy az oldalak szépek és egységesek legyenek, és ne kelljen minden apró stílust kézzel megírni, eldöntöttük, hogy használok egy UI könyvtárat. A választásom a `Shadcn/ui-ra` esett, ami React komponenseket biztosít Tailwind CSS-szel.

1. `Shadcn/ui` telepítése: A parancssorban telepítettük a `Shadcn/ui-t`. Ez létrehozta a szükséges konfigurációs fájlokat.

2. Szükséges komponensek telepítése: Utána hozzáadtuk azokat a specifikus komponenseket, amikre szükségem volt, például gombokat, kártyákat, amikkel a termékeket akartam megjeleníteni.

Problémák és megoldások (mert persze itt is voltak!)

• **Probléma:** Képek nem töltődtek be Amikor megpróbáltuk képeket hozzáadni az oldalakhoz, egyszerűen nem jelentek meg. Emiatt persze bosszankodtunk , mert azt hittük. valami bonyolult dologról van szó.
Megoldás: Végül rájöttünk, hogy csak annyi volt a baj, hogy létre kellett hozni egy `public/images/` mappát, és ide kellett tenni a placeholder képeket. Ez egy nagyon alapvető dolog, de könnyen el lehet felejteni az elején.

• **Probléma:** `Shadcn/ui` komponensek nem működtek Hiába telepítettem a `Shadcn/ui-t`, a komponensek nem akartak úgy kinézni, ahogy kellett volna, vagy nem működtek rendesen.
Megoldás: Kiderült, hogy frissítenünk kellett a `tailwind.config.ts` fájlt, hogy a Tailwind CSS megfelelően felismerje és használja a `Shadcn/ui` stílusait. Ez is egy konfigurációs dolog volt, amit eleinte nem vettünk észre.
Következő lépések Most, hogy a főoldalak elkészültek és már `Shadcn/ui` komponensekkel is tudtunk dolgozni, a következő feladatunk az volt, hogy létrehozzunk a

termék al-oldalakat. Ez azért fontos, mert minden egyes terméknek szüksége van egy saját, részletes bemutató oldalra.

Termék aloldalak készítése

Miután a honlap alapvető struktúrája és a főbb oldalak elkészültek, az volt a feladatunk, hogy minden terméknek legyen egy saját, részletes bemutató oldala. Gondoljatok bele, ha valaki rákattint egy ablakra a "Termékeink" oldalon, akkor látnia kell egy oldalt, ami csak arról az ablakról szól, minden fontos információval. Ehhez használtunk a `Next.js` egyik szuper funkcióját, a dinamikus routingot.

Dinamikus routing beállítása A dinamikus routing azt jelenti, hogy nem kell minden egyes termékhez külön-külön oldalt létrehoznunk. Ehelyett egyetlen sablon oldal elég, ami majd az URL alapján tudja, melyik termék adatait kell megjelenítenie.

1. Dinamikus route létrehozása: Létrehoztam egy speciális mappát és fájlt az `app/termekeink/[slug]/page.tsx` útvonalon. A `[slug]` rész a lényeg, ez jelzi, hogy itt egy változó érték (például a termék azonosítója vagy egyedi neve) fog szerepelni az URL-ben.
Termék típusok definiálása Hogy a kódom rendezett és biztonságos legyen, TypeScript interfészeket definiált a `lib/product-utils.ts` fájlban. Ez segít abban, hogy mindig tudjuk, milyen adatokat várunk egy terméktől (pl. név, ár, leírás, kép URL-ek), és elkerüljük a gépelési hibákat.
Képek kezelése Persze minden termékoldalra kellenek képek!

1. Placeholder képek hozzáadása: Először is, a `public/images/` mappába tettük néhány alap képet. Ezek ilyen "ideiglenes" képek voltak, amiket addig használtunk, amíg nem volt valós kép a termékekhez. Például `oknoplast-windows.jpg` vagy `okna-dom-parterowy-oknoplast.jpg`.

2. Next.js Image komponens használata: A képek betöltésének optimalizálásához a Next.js saját Image komponensét használtuk. Ez azért jó, mert automatikusan méretezi és optimalizálja a képeket, ami gyorsabbá teszi az oldalt.

Problémák és megoldások (mert nem megy minden rögtön elsőre)

• **Probléma:** Dinamikus route nem működött Eleinte nem értettük, miért nem működik a dinamikus oldalam. Bármilyen URL-t írtam be a `/termekeink/` után, az oldal nem töltődött be rendesen. Kicsit zavaró volt, hogy nem jöttünk rá azonnal.
Megoldás: Végül kiderült, hogy a fájlnevet pontosan `[slug].tsx-re` kellett neveznünk, és a params nevű prop-ot, ami az URL-ből érkező adatokat tartalmazza, megfelelően kellett típusozni. Ez egy tipikus kezdő hiba, hogy az ember nem figyel a pontos elnevezésekre és a TypeScript típusokra!

• **Probléma:** Statikus generálás hibák A `Next.js` tud statikusan, előre generálni oldalakat, ami nagyon gyorssá teszi őket. Ezt a generateStaticParams függvénnyel lehet megcsinálni, de nálunk hibákat dobott.
Megoldás: A probléma az volt, hogy a termékeket egy külön fájlba kellett tenni, és onnan kellett importálni őket a generateStaticParams függvénybe.

• **Probléma:** Képek nem töltődtek be Bár már a fő-oldalaknál is volt ilyen gond, itt is belefutottunk, hogy a termék al-oldalon sem akartak megjelenni a képek.
Megoldás: Megint csak ellenőrizni kellett a fájlneveket és a képek relatív útvonalait a public mappában. Kicsit kellemetlen, hogy kétszer is belefutottunk ebbe, de legalább megtanultuk alaposan.

Következő lépések Most, hogy a termék aloldalak elkészültek, rájöttünk egy fontos dologra: a termékeket most még a kódban `"hardkódoltan"` tároltunk. Ez azt jelenti, hogy ha hozzáadunk egy új ablakot, vagy megváltozna az ára, akkor minden alkalommal módosítani kellene a kódot, és újra kiadni az oldalt. Ez így nem praktikus.
Szóval a következő lépés az volt, hogy át-álljunk egy adatbázis alapú megoldásra, hogy sokkal könnyebben tudjuk kezelni a termékeket.

Hardcoded Termékek Javítása Adatbázisra

Miután az előző lépésben elkészültek a termék aloldalak, belefutottunk egy elég nagy problémába, amiről őszintén szólva, eleinte nem is tudtunk, hogy probléma. A termékek adatait, mint például a nevük, leírásuk vagy áruk, közvetlenül a kódba írtuk be. Ezt hívják `"hardcoded"` termékeknek.

Probléma felismerése Rájöttünk, hogy ez így nem fenntartható. Ha később hozzá akarunk adni egy új ablakot, vagy megváltozik egy meglévő termék ára, akkor minden egyes alkalommal módosítani kellene a kódot, és újra ki kellene adni az egész weboldalt. Ez rengeteg felesleges munka és hibalehetőség, főleg ha sok termék van. Ezért világossá vált, hogy egy adatbázisra van szükségünk, ahonnan a weboldal dinamikusan le tudja kérni a termékeket.

Adatbázis tervezése Mielőtt bármit is csináltunk volna, leültünk és átgondoltuk, hogy nézzen ki ez az adatbázis.

1. Termékek tábla struktúra: Először is megterveztük egy products nevű táblát. Ez a tábla tartalmazza az összes fontos információt egy termékről: név, leírás, ár, kép URL-je, és talán még valamilyen műszaki specifikáció is. Ehhez létrehoztam egy SQL fájlt, a `scripts/create-products-table.sql` nevű fájlt.

2. Adatok betöltése: Ahhoz, hogy legyen is valami az adatbázisban, létrehoztunk egy másik SQL fájlt, a `scripts/seed-products-data.sql` nevűt, ami betöltött néhány kezdeti, teszt adatot a products táblába.

Adatbázis integráció Ezután következett az adatbázis bekötése a weboldalba. `Next.js-t` és Supabase-t használtam, ami egy PostgreSQL alapú adatbázis szolgáltatás.

1. Termékek lekérdezése: Meg kellett változtatni azt a részt a kódba, ami eddig a `hardcoded` termékeket olvasta. Frissítettem a `lib/product-utils.ts` fájlt, hogy most már a Supabase adatbázisból kérje le a termékeket.

2. Oldalak frissítése: Végül frissítettük a Termékeink oldal `(app/termekeink/page.tsx)` és a dinamikus termék aloldalak `(app/termekeink/[slug]/page.tsx)` kódját is, hogy ezek is az adatbázisból vegyék az adatokat, ne pedig a kódból.

Problémák és megoldások (mert persze itt sem ment minden zökkenőmentesen!)

• **Probléma:** Supabase kapcsolat nem működött. A legelső akadály az volt, hogy a weboldal egyszerűen nem tudott kommunikálni a Supabase adatbázissal. Vártuk, vártuk, de nem jött válasz. Először azt gondoltunk, valami bonyolult hálózati beállításról van szó.
Megoldás: Végül rájöttünk, hogy elfelejtettük beállítani a környezeti változókat (`.env.local` fájlban) a Supabase API URL-jét és anon kulcsát. Ez egy klasszikus kezdő hiba, de ilyenkor az ember hajlamos túlgondolni a dolgot.

• **Probléma:** Server-side rendering hibák Mivel a `Next.js` szerver-oldalon is rendereli az oldalakat, ez bonyolultabbá tette az adatbázis kapcsolatot. Különböző hibákat kaptunk a konzolon, ami arra utalt, hogy a szerver nem tudja rendesen lekérdezni az adatokat.
Megoldás: A megoldás az volt, hogy létre kellett hozni egy külön `lib/supabase/server.ts` fájlt, ami kifejezetten a szerver-oldali Supabase klienst konfigurálja. Ezt specifikusan kell beállítani `Next.js`-ben.

• **Probléma:** JSON parsing hibák A termékekhez olyan adatokat is akartunk tárolni, mint a "specifikációk", amik elég összetettek, így JSON formátumban tettünk az adatbázisba. Amikor próbáltuk kiolvasni ezeket, hibákat kaptunk, hogy nem tudja értelmezni a JSON-t.
Megoldás: Kicsit megijedtem, hogy nekünk kell majd kézzel `parse-olnom` a JSON-t, de kiderült, hogy a Supabase automatikusan kezeli a JSONB típusokat. Csak ellenőriznünk kellett, hogy az adatbázisban tényleg JSONB típusként van-e beállítva az oszlop. Ez egy szerencsés felismerés volt, mert sok időt spórolt meg nekünk!

Következő lépések Most, hogy a termékek már adatbázisból jönnek, és sokkal könnyebben tudtul őket kezelni (legalábbis elméletben), a következő feladatunk az volt, hogy létrehozzuk az email kapcsolati formot. Ez egy nagyon fontos funkció, mert így a látogatók könnyen tudnak majd üzenetet küldeni a cégnek.

Email Kapcsolati Form Beállítása

Miután a `hardcoded` termékek problémáját megoldottuk az adatbázis integrációval, a következő fontos lépés az volt, hogy létrehozztuk a kapcsolati formot. Gondoltuk, ez nagyon fontos, mert így a honlap látogatói könnyen tudnak majd üzenetet küldeni a cégnek, és ez elengedhetetlen a potenciális ügyfelek számára.

Kapcsolati oldal létrehozása Először is, létrehoztuk egy teljesen új oldalt a weboldalon, ez lett az `app/kapcsolat/page.tsx` fájl. Ez az oldal fogja tartalmazni magát a formot, amit a felhasználók kitöltenek.

Email küldés implementálása Az email küldéshez több opciót is megnéztem.

1. EmailJS vagy API Route: Először az EmailJS-t próbáltuk volna használni, mert azt hallottuk, hogy egyszerű, de aztán rájöttünk, hogy egy saját API route is elég lesz, és az jobban illeszkedik a `Next.js` projekthez. Ez azt jelenti, hogy a saját szerverünkön keresztül fogjuk elküldeni az e-maileket.
2. API Route létrehozása: Létrehoztuk egy `app/api/contact/route.ts` fájlt, ami egy szerver oldali végpont (API route) lesz az email küldéshez. Ez a fájl fogja ténylegesen fogadni a form adatait és elküldeni az e-mailt.
3. Nodemailer telepítése: Ahhoz, hogy a szerver oldalon tudjuk emailt küldeni, telepíteni kellett a Nodemailer nevű csomagot. Ez egy népszerű könyvtár, ami egyszerűvé teszi az email küldést `Node.js-ben`.
4. Környezeti változók beállítása: Muszáj volt beállítani a `.env.local` fájlban a `GMAIL_APP_PASSWORD` és a `GMAIL_EMAIL_ADDRESS` változókat. Ez azért fontos, mert ezek tartalmazzák az email küldéshez szükséges titkos adatokat, és nem szabad ezeket a kódban tárolni.

Gmail App Password beállítása (ez egy kis fejtörést okozott) Ez volt az egyik trükkösebb része a feladatnak, mert a Google biztonsági beállításai miatt nem lehetett csak úgy simán a Gmail jelszavát használni.

1. Google fiók beállítások: Először is, be kellett kapcsolni a kétlépcsős hitelesítést a Google fiókomban. Ez alapfeltétel volt az alkalmazásjelszavak létrehozásához.

2. App Password létrehozása: Utána tudtuk létrehozni egy "App Password"-ot, ami egy speciális, alkalmazásspecifikus jelszó. Ezt választottuk a "Mail" alkalmazáshoz, és a generált jelszót kimásoltuk a `.env.local` fájlba. Ez volt az, amit a Nodemailer használt a bejelentkezéshez.

Form frissítése API hívással Végül, frissítettem a kapcsolati formom kódját, hogy amikor a felhasználó rákattint a "Küldés" gombra, az adatokat elküldje az újonnan létrehozott API route-nak. Az API route pedig elküldi az e-mailt.

Email tesztelése Miután mindent beállíthatunk, megnyitottunk a kapcsolati oldalt, kitöltöttük a formot, és izgatottan vártuk. Szerencsére az email sikeresen megérkezett a Gmail fiókba. Az e-mail tartalmazta a küldő nevét, e-mail címét, telefonszámát (ha megadta), az üzenet tartalmát és egy időbélyeget is. Ez egy jó érzés volt, hogy működik!

Problémák és megoldások (mert nem megy minden rögtön elsőre)

• **Probléma:** Gmail App Password nem működött: Ez eleinte nagyon zavaró volt. Bármilyen beállítással próbálkoztunk, a Nodemailer nem tudott bejelentkezni.

Megoldás: Kiderült, hogy tényleg engedélyezni kellett a kétlépcsős hitelesítést, utána kellett létrehozni az alkalmazás jelszót, és ami még fontosabb, nagyon pontosan ki kellett másolnom a generált jelszót. Egyetlen rossz karakter, és már nem működött. Ez egy tipikus apró hiba volt, amin sok idő elment.

• **Probléma:** CORS hibák: Aggódtam a CORS (Cross-Origin Resource Sharing) hibák miatt, amik gyakran előfordulnak, amikor egy frontend alkalmazás egy másik forrásról (például az API-ról) próbál adatokat lekérni.
Megoldás: Szerencsére `a Next.js` API route-ok automatikusan kezelik a CORS-t, így ezzel nem kellett külön foglalkozni. Ez egy szerencsés felismerés volt.

• **Probléma:** E-mail formátum hibák: Az első elküldött e-mailek formátuma nem volt a legszebb, a HTML elrendezése nem úgy nézett ki, ahogy szerettük volna.
Megoldás: Egyszerűen egy sokkal egyszerűbb HTML struktúrát használtunk az email tartalmához, ami végül jobban mutatott. Néha a kevesebb több!

Következő lépések Most, hogy az email kapcsolati form sikeresen működött, a következő lépésünk az volt, hogy teljesen integráljuk a Supabase-t a projektbe. Ez azt jelentette, hogy minden adat, beleértve a felhasználói interakciókat is, az adatbázisból jöjjön, és ne legyenek `"hardcoded"` elemek.

Supabase Bekötése

Miután az email kapcsolati form sikeresen működött, rájöttünk, hogy az egész projekt még jobban működhetne, ha minden adatot adatbázisból kezelnénk. Ekkor döntöttünk el, hogy teljesen integrálom a Supabase-t a projektbe. A Supabase azért tűnt jó választásnak, mert egy modern, nyílt forráskódú alternatíva a Firebase-re, ami PostgreSQL adatbázist és valós idejű funkciókat is kínál.

Supabase projekt létrehozása és telepítése:

1. Csomagok telepítése: Először is telepíteni kellett a szükséges Supabase csomagokat a projektbe.
2. Supabase projekt létrehozása online: Kinyitottuk a supabase.com oldalra, és létrehoztuk egy új projektet. Kiválasztottam a "Europe West" régiót, és vártam, amíg az inicializálódik. Ez eltartott egy ideig, de nem volt túl bonyolult.
3. Környezeti változók beállítása: Létrehoztuk vagy frissítettem a `.env.local` fájlt a Supabase API URL-jével és az anon kulccsal, amiket a Supabase adott. Ezt már tanultuk a hardcoded termékek javításánál, hogy mennyire fontos!

Supabase kliens beállítása Mivel Next.js-t használok, ami szerver-oldalon és kliens-oldalon is renderel, kétféle Supabase klienst is be kellett állítanom:

1. Kliens-oldali kliens: Létrehoztunk a `lib/supabase/client.ts` fájlt a böngészőben futó részekhez.

2. Szerver-oldali kliens: Létrehoztun a `lib/supabase/server.ts` fájlt a szerver-oldali lekérdezésekhez. Ezt már a hardcoded termékek javításakor is megcsináltuk, mert akkor is szerver-oldali hibákat kaptunk.

Adatbázis táblák létrehozása Ugyan már megterveztem a products táblát a hardcoded termékeknél, most a Supabase adatbázisban is létre kellett hoznom.

1. products tábla létrehozása: A Supabase SQL Editor-jában lefuttattuk a `scripts/create-products-table.sql` scriptet.

2. Kezdeti adatok betöltése: Majd lefuttattuk a `scripts/seed-products-data.sql` scriptet is, hogy legyen pár teszt termékem.

Custom Hook létrehozása Hogy könnyebben tudjam használni a termék adatokat a frontend oldalon, létrehoztuk egy saját React hook-ot.

1. useProducts hook: Létrehoztuk a `hooks/use-products.ts` fájlt, ami a termékek lekérdezését végzi a Supabase-ból.

Storage bucket beállítása A termékek képeinek tárolására is szükségem volt.

1. Storage bucket létrehozása: A Supabase Storage felületén létrehoztunk egy `oknoplast-images` nevű `"bucket"-et.` Ez olyan, mint egy mappa a felhőben, ahová feltöltöttük a képeket.
2. Storage policy beállítása: Be kellett állítani a megfelelő `"policy"-ket` is, hogy a képek publikusan elérhetőek legyenek, és a weboldal meg tudja jeleníteni őket.

Tesztelés Természetesen tesztelni kellett, hogy minden működik-e:

1. Adatbázis kapcsolat tesztelése: Létrehoztuk egy nagyon egyszerű teszt-oldalt, csak hogy lássuk, tudjuk-e egyáltalán csatlakozni az adatbázishoz.
2. Termékek megjelenítése: Ellenőriztük, hogy a "Termékeink" oldalon (amit korábban már frissítettünk, hogy adatbázisból olvassa a termékeket) megfelelően betöltődnek-e az adatok.

Problémák és megoldások (mert nem volt ez sem zökkenőmentes!)

• **Probléma:** RLS (Row Level Security) hibák Ez volt az első és legnagyobb falat. Hiába volt fent minden az adatbázisban, a weboldal nem tudta lekérdezni a termékeket, "engedély megtagadva" (permission denied) üzeneteket kaptuk. Eleinte azt hittük, valami bonyolult kódot rontottunk el.
Megoldás: Kiderült, hogy a Supabase adatbázis biztonsági beállításai, az úgynevezett RLS `policy-k` nem engedték a lekérdezést. Be kellett menni a Supabase `dashboard-jába`, a `"Authentication" -> "Policies"` részre, és ott beállítani egy `policy-t a products` táblához, ami engedélyezi a `"select"` (lekérdezés) műveletet a `"public"` szerepkör számára. Kicsit kínos volt, hogy egy ilyen alap dolog miatt akadtunk el, de legalább megtanultuk, hogy mindig nézzük meg az adatbázis jogosultságait!

• **Probléma:** JSONB mezők nem működtek.A specifications (műszaki adatok) mezőket JSONB formátumban tároltuk, és ismét volt egy kis aggodalom, hogy vajon rendesen tudjuk-e majd kezelni.
Megoldás: Szerencsére rájöttünk, hogy a Supabase automatikusan kezeli a JSONB típusokat, ahogy azt már a `hardcoded` termékeknél is láttuk. Csak ellenőrizni kellett, hogy az adatbázisban tényleg JSONB típusúként van-e beállítva az oszlop.

• **Probléma:** CORS hibák Ismét egy kis riadalom a CORS hibák miatt, amelyek gyakran előfordulnak különböző rendszerek közötti kommunikáció során.
Megoldás: Ahogy a `Next.js` API route-ok esetében, úgy a Supabase is alapból kezeli a CORS-t, szóval nem kellett vele külön foglalkozni, csak ellenőrizni a projekt beállításokat.

Következő lépések Most, hogy a Supabase teljesen integrálva van, és az adatok adatbázisból jönnek, a következő nagy kihívás az volt, hogy létrehozzuk az admin felületet. Ez azért fontos, mert így majd mi magunk is tudjuk kezelni a termékeket (hozzáadni, módosítani, törölni) anélkül, hogy közvetlenül az adatbázisban kellene matatni. Ez sokkal felhasználóbarátabb lesz!

Admin Felület Fejlesztése

Futtattuk az admin felület alapvető oldalát. Ez az `app/admin/page.tsx` fájl lett, ami az admin felület „belépő pontja”.

Admin funkciók Az admin felületen belül több funkciót is megvalósítottunk:

• **Bejelentkezés:** Ehhez egy egyszerű jelszó alapú autentikációt készítettünk. A bejelentkezést követően a rendszer a felhasználó adatait (vagy inkább a session állapotát) a böngésző LocalStorage-jában tárolja, ami segíti a bejelentkezett állapot megőrzését. Fontos volt, hogy minden admin művelet előtt ellenőrizzük a bejelentkezést.

• **Termékek kezelése:**
  ◦ *Megtekintés:* Az összes termék listázása egy helyen.
  ◦ *Szerkesztés:* Lehetőség van a termékek adatainak módosítására.
  ◦ *Törlés:* Egy kattintással eltávolítottunk egy terméket.
  ◦ *Új termék hozzáadása:* Új termékeket vihetek fel a rendszerbe.

• **Adatbázis műveletek:** Ezek mind a Supabase-en keresztül történnek (CRUD műveletek), és beépítettünk a valós idejű adatfrissítést is. A hibakezelésről és a felhasználói visszajelzésről is gondoskodtunk.

Problémák és megoldások (mert itt is volt mit tanulni!)

• **Probléma:** Admin jogosultságok – Bárki hozzáférhetett az admin felülethez. Ez az elején egy kicsit ijesztő volt, mert rájöttünk, hogy az admin oldalhoz elvileg bárki hozzáférhet-ne, ha ismeri az URL-t. Ez nyilván nem biztonságos.
Megoldás: Egy egyszerű jelszó alapú autentikációt implementáltam, és a bejelentkezett állapotot a LocalStorage-ban tároltam. Most már minden admin művelet előtt ellenőrzünk,

hogy a felhasználó be van-e jelentkezve. Nem a legprofibb megoldás (pl. nincs felhasználókezelés, csak egy fix jelszó), de egy kezdő projekthez teljesen megfelelő és sokkal jobb, mint a semmi!

• **Probléma:** JSON validáció – Hibás JSON a műszaki adatoknál. A specifications (műszaki adatok) mező egy JSONB típusú oszlop az adatbázisban. Előfordult, hogy a felhasználó (én) véletlenül rossz formátumú JSON-t vitt be.
Megoldás: Egy try-catch blokkot használtunk a JSON.parse művelet körül. Így, ha valaki rossz JSON-t próbál meg beírni, a program nem omlik össze, hanem csak akkor frissíti az adatokat, ha érvényes JSON-t kap. Ráadásul kap a felhasználó (azaz én) egy visszajelzést a hibáról.

• **Probléma:** Kép URL validáció – Nem ellenőriztük a kép URL-eket. A termékekhez tartozó kép URL-ek mezőjébe bármit be lehetett írni, ami később problémákat okozhatott volna a képek megjelenítésénél.
Megoldás: Bár nem egy teljes körű URL validáció, de hozzáadtunk egy egyszerű ellenőrzést az URL formátumára. Emellett, ha valamilyen okból nincs érvényes URL megadva, akkor egy `placeholder` képet használunk. Ez nem tökéletes, de megakadályozza, hogy hibás linkek törjék el az oldalt.

Következő lépések Most, hogy az admin felület készen áll, és tudok termékeket kezelni, a következő nagy feladat az volt, hogy implementáljuk a képek feltöltését és kezelését közvetlenül a Supabase Storage-ban. Így nem kellene manuálisan feltölteni a képeket és beilleszteni az URL-eket, hanem mindent az admin felületen keresztül tudnánk intézni.

Képek Feltöltése és Kezelése

Az admin felület elkészítése után rájöttünk, hogy az admin felület csak akkor az igazi, ha közvetlenül tudjuk kezelni a termékekhez tartozó képeket is, nemcsak az URL-eket beírni. Ezért a következő lépés az volt, hogy implementáljuk a képek feltöltését és kezelését a Supabase Storage-ban. Így a képeket is központilag tudjuk kezelni, és a weboldal is dinamikusabb lesz.

Storage Integráció és Beállítások

1. Storage bucket létrehozása: A Supabase dashboard-ban létrehoztuk egy oknoplast-images nevű `"bucket"-et`. Ez olyan, mint egy felhőalapú mappa, ahová feltölthettünk a képeket. Ezt már a Supabase bekötésekor is említettük, de most konkrétan a képek miatt csináltuk meg.
2. Storage `policy-k` beállítása: Be kellett állítani a megfelelő biztonsági szabályokat (`policy-ket) is a bucket-hez`. Ez biztosítja, hogy a képek publikusan elérhetők legyenek, vagy épp fordítva, csak az admin tudja őket kezelni. Eleinte ezzel volt egy kis gond, ahogy a "Problémák és megoldások" részben is látni fogod.

Admin Felület Kiegészítése Képkezelő Funkciókkal Az admin felületet bővíteni kellett, hogy tudjuk a képeket feltölteni és kezelni:

- **Fájlkezelő tab hozzáadása:** Az admin felületen belül létrehoztuk egy külön "Fájlkezelő" fület. Ez egy saját kis galéria, ahol láttuk az összes feltöltött képet.
- **Fájl feltöltés funkció:** Implementálttuk a fájl feltöltés gombot és a mögötte lévő logikát, ami feltölti a kiválasztott képet a Supabase Storage-ba.
- **Termék kép feltöltés és hozzárendelés:** Amikor egy új terméket adok hozzá vagy egy meglévőt szerkesztek, most már közvetlenül feltölthettük a képet hozzá. Ez sokkal egyszerűbb, mint manuálisan URL-eket másolgatni.
- **Fájl törlés funkció:** Természetesen a feltöltött képeket törölni is tudnunk kellett az admin felületen keresztül.
- **Galéria kép kiválasztása:** A Fájlkezelő tab-ban tudjuk képeket kiválasztani a termékekhez, amiket korábban feltöltöttünk.

UI Komponensek Frissítése Ezekhez a funkciókhoz persze frissíteni kellett a felhasználói felületet (UI komponenseket) is. Hozzáadtuk a fájlkezelő tabot és a kép feltöltő gombokat, hogy minden látható és használható legyen.

Setup Scriptek Létrehoztuk egy SQL scriptet is, a `scripts/setup-storage-bucket-v3.sql` fájlt, ami a Storage bucket beállításait automatikusan el tudja végezni. Ez hasznos, ha újra kell kezdeni a projektet, vagy más környezetben is be kell állítani.

Problémák és Megoldások (mert ez sem volt zökkenőmentes!)

- **Probléma:** Bucket nem létezett vagy rossz `policy-k:` Amikor először próbáltuk feltölteni a képeket, az alkalmazás hibát dobott, hogy a `"bucket"` nem létezik, vagy nincs hozzá jogosultságunk. Kicsit frusztráló volt, mert azt hittük, már létrehoztuk.
  Megoldás: Kiderült, hogy nem állítottuk be megfelelően a `public` read `policy-t`, vagy nem a megfelelő bucket névre hivatkoztunk. Visszamentünk a Supabase dashboardba, és ott ellenőriztük, hogy a `oknoplast-images` nevű bucket tényleg létezik és publikusan elérhető, vagy legalábbis, hogy a megfelelő jogosultságokkal rendelkezik a feltöltésre és olvasásra. A hibakeresésnél az admin felületre is be-építettünk egy kis ellenőrzést, hogy ne dobjon azonnal hibát, ha valami nincs rendben.

- **Probléma:** Fájl feltöltés hibák: Előfordult, hogy a feltöltött fájlok nem akartak megjelenni, vagy a rendszer "túl nagy fájl" hibát dobott.
  Megoldás: Rá kellett jönni, hogy a Supabase Storage-ban is vannak fájlméret-korlátozások. A feltöltés előtt ellenőriztük a fájl méretét és típusát a kliens oldalon, hogy elkerüljük a felesleges hálózati forgalmat. Emellett beépítettünk egy kis trükköt is: minden feltöltött fájlnak generálunk egy egyedi nevet, ami tartalmazza az aktuális időbélyeget `(timestamp)`, így elkerüljük a név ütközéseket. Persze, ha valami mégis elromlik, akkor egy megfelelő hibakezeléssel `(error handling)` értesítse a felhasználót (vagyis minket) a problémáról.

- **Probléma:** Kép URL-ek nem működtek: Feltöltöttük a képeket, de amikor megpróbáltuk megjeleníteni őket a weboldalon, nem látszottak. Üres képeket vagy törött linkeket kaptunk.
  Megoldás: Ez is egy jogosultsági probléma volt. Ellenőrizzük, hogy a Supabase bucket valóban `"public"` beállítással rendelkezik-e. Ha nem, akkor a getPublicUrl függvény

segítségével kellett lekérdezni az URL-eket, ami kezeli a token alapú hozzáférést. Fontos volt, hogy a getPublicUrl függvényt használjam a Supabase kliensből, hogy a linkek mindig helyesek legyenek.

Tesztelés Természetesen tesztelni kellett, hogy minden új funkció megfelelően működik-e:

• **Fájl feltöltés tesztelése:** Felmentünk az admin felületre, a Fájlkezelő tabra, és feltöltöttünk pár teszt képet. Ellenőriztük, hogy megjelennek-e a listában, és hogy az URL-jük is rendben van.
• **Termék kép hozzárendelése:** Létrehoztunk egy új terméket, és feltöltöttünk hozzá egy képet. Megnézük, hogy az URL automatikusan bekerült-e a termék adatlapjába. Teszteltük azt is, hogy a már feltöltött képek közül tudunk-e választani galéria képnek.
• **Fájl törlés tesztelése:** Végül kipróbáltuk, hogy törlünk egy feltöltött fájlt. Ellenőriztük, hogy eltűnik-e a listából, és hogy a termék oldalakon nem jelenik meg többé.
Következő lépések.

Most, hogy a teljes rendszer kész van, a honlap rendelkezik minden fő funkcióval: van egy `Next.js` alapú honlapom `responsive design-nal,` Supabase adatbázis integrációval, egy admin felülettel a termékek kezeléséhez, email kapcsolati formmal, kép feltöltéssel és kezeléssel, valamint dinamikus termék al-oldalakkal.

További lépések lehetnek még: SEO optimalizálás, teljesítmény optimalizálás, további admin funkciók hozzáadása, vagy akár mobil alkalmazás fejlesztése is.

Tesztelés

Smoke tesztelés

Email küldés

Amikor a projekt kivitelezési fázisába érve az alapvető működést ellenőriztük, a "smoke tesztelés" során két fő területre koncentráltunk: az email küldési funkcióra és az oldalak általános betöltésére. Kezdő programozóként természetesen belefutottunk néhány meglepetésbe, de igyekeztünk alaposan utánajárni a problémáknak és megoldást találni rájuk.

Email küldés tesztelése:

Az első nagy lépés a kapcsolati form működésének ellenőrzése volt. A cél az volt, hogy miután valaki kitölti az űrlapot az oldalon, az üzenet sikeresen eljusson a megadott e-mail címre.

1. A kihívás: Létrehoztunk az API route-ot az email küldéshez, telepítettük a Nodemailert, és beállítottuk a környezeti változókat a `.env.local` fájlban egy Gmail adatokkal.

Izgatottan töltöttük ki az űrlapot a fejlesztői környezetben, elküldtük, majd vártunk... és vártunk. Az e-mail sehogy sem akart megérkezni. Pánikba estünk, hogy valami alapvető dolgot rontottunk el. Visszanéztük a kódot, ellenőriztük a változókat, de semmi nyilvánvaló hiba nem tűnt fel.

2. A "kezdő" megoldás keresése: Emlékeztünk, hogy a forrásanyagok is említettek hasonló problémát, mégpedig a Gmail App Password nem működött hibáját. Ez adta a tippet. Eleinte azt hittük, elég a szokásos Gmail jelszó de kiderült, hogy a Google biztonsági beállításai miatt egy speciális App Password-re van szükség az ilyen alkalmazásokhoz.

- Először is engedélyezni kellett a kétfaktoros hitelesítést a Google fiókban.
- Ezután létrehozni egy új "App Password"-ot kifejezetten a "Mail" alkalmazáshoz.
- A kulcsfontosságú lépés az volt, hogy pontosan másoljuk ezt a generált jelszót a `.env.local` fájlba, mert egy apró elírás is hibához vezetett.

3. Az eredmény: Miután ezeket a lépéseket gondosan elvégeztük, újra elküldtük a tesztüzenetet a kapcsolati oldalon. Ezúttal nagy megkönnyebbülésünkre az e-mail sikeresen megérkezett a Gmail fiókba. Ellenőriztük, hogy a küldő neve, e-mail címe, telefonszáma és az üzenet tartalma is a helyén van. Ez a tapasztalat megtanított minket arra, hogy az apró konfigurációs részletek mennyire fontosak.

Oldalbetöltés tesztelése

Az oldalak betöltésének ellenőrzése volt az első és talán legfontosabb lépés, hogy megbizonyosodjunk róla, a `Next.js` alkalmazásom alapszinten működik.

1. A kihívás: A fejlesztői szerver indítása után `(http://localhost:3000)` az első oldal, a "Hello World" elvileg meg kellett volna, hogy jelenjen. Azonban kezdetben azt vettük észre, hogy bár a szöveg látható volt, valami mégsem stimmelt a stílusokkal. A Tailwind CSS stílusok nem működtek megfelelően, vagy éppen a hot reload nem frissítette automatikusan a változtatásokat, ami nagyon lassította a fejlesztést. Ráadásul később, amikor képeket akartuk megjeleníteni, előfordult, hogy nem töltődtek be.

2. A "kezdő" megoldás keresése:

- *Tailwind CSS:* Hosszú ideig néztük a kódot és a böngésző konzolját, de nem jöttünk rá azonnal, mi a baj. Végül a forrásanyagok útmutatása segített, miszerint a `tailwind.config.ts` fájlban be kellett állítani a content útvonalakat, hogy a Tailwind tudja, hol keresse a használt osztályokat. Ez egy olyan lépés volt, amit kezdőként könnyen kihagyhatunk, mert nem tűnt azonnal nyilvánvalónak a jelentősége.
- *Hot reload:* Amikor a hot reload nem működött, a legkézenfekvőbb, de gyakran elfeledett megoldáshoz folyamodtunk: újraindítottam a fejlesztői szervert. Ez egy egyszerű, de hatékony lépés volt, ami azonnal orvosolta a problémát és lehetővé tette a gyorsabb iterációt.
- *Képek betöltése:* Mikor a képek nem akartak megjelenni, többször ellenőriztük a HTML kódokat, a kép elérési útvonalait. Végül rájöttünk, hogy a `public/images/` mappába

kellett helyezni a placeholder képeket, mert a Next.js onnan szolgálja ki a statikus fájlokat. Később még a fájlnevek és relatív útvonalak pontosságát is ellenőriztük.
3. Az eredmény: A kezdeti buktatók után az oldalak már megfelelően betöltődtek: a "Hello World" szöveg megjelent, a Tailwind CSS stílusok működtek (például a kék szín és a középre igazítás), a Google Fonts (Inter) betűtípus is betöltődött. A képek is elkezdtek megjelenni a helyükön. Ez megerősített minket abban, hogy az alapvető beállítások most már rendben vannak, és a projekt készen áll a további fejlesztésre.

Architektúra

Amikor elkezdtük ezt a projektet, az volt a célunk, hogy egy modern, profi weboldalt hozzunk létre az Oknoplast Debrecen számára, különös tekintettel az ügyfél igényeire, mint például az árajánlatkérő űrlap és a reszponzív design.
A célunk az volt, hogy egy olyan rendszert építsek, amely nem csak szép, de funkcionális és könnyen kezelhető is. A projekt során sok kihívással találkoztunk, de mindegyiket sikerült megoldanunk, és lépésről lépésre haladtam előre.

Először is, a `Next.js-t` választottunk, mert ez egy modern React framework, ami sok hasznos funkciót biztosít, és a technológia kiválasztása során fontos volt, hogy megbízható és elterjedt keretrendszerrel dolgozzak.

A projektet TypeScript-tel, Tailwind CSS-sel, ESLint-tel és az új App Router-rel állítottunk be, és már az elején egy olyan hibával találkoztunk, hogy a TypeScript nem ismerte fel a `@/ import alias-t`. Ezt úgy oldottam meg, hogy beállítottunk a `tsconfig.json` fájlt. A Tailwind CSS sem működött elsőre, ott a `tailwind.config.ts` fájlban kellett a content útvonalakat beállítani.

Ezután a következő lépésünk az volt, hogy létrehoztuk az első oldalt, egy egyszerű "Hello World"-öt, hogy megbizonyosodjunk róla, minden rendben van.
Az App Router miatt az app/ mappába került az oldal, és az `app/layout.tsx` fájlban definiáltunk az alapvető HTML struktúrát. A stílusokhoz a Tailwind CSS-t importáltuk a `globals.css` fájlba.

Amikor teszteltük, egy olyan hibával találkoztunk, hogy a hot reload nem működött megfelelően, amit a fejlesztői szerver újraindításával orvosoltunk. Később TypeScript hibákat kaptuk a React típusokkal kapcsolatban, amihez a szükséges típusokat kellett telepíteni.

Miután az alap Hello World oldal működött, a következő lépésünk az volt, hogy elkészítsük a honlap alapvető layout struktúráját, ami a navigációs sávot `(header)` és a láblécet `(footer)` tartalmazza.

Ezeket külön komponensek-ként hoztunk létre, majd integráltuk az `app/layout.tsx` fájlba. A stílusozás-nál a responsive design-re is figyeltünk, mobil-first megközelítéssel és Tailwind CSS breakpointje-ivel. Itt egy olyan hibával találkoztunk, hogy a Lucide React ikonok nem töltődtek be, és a mobil menü nem záródott be automatikusan.

Ezeket a megfelelő `npm` csomagok telepítésével és egy onClick esemény hozzáadásával sikerült megoldani.

Most, hogy a layout készen volt, a következő lépésünk a főbb oldalak létrehozása volt, mint a Főoldal, Termékeink, Szolgáltatások, Blog, Rólunk és Kapcsolat.
A Főoldalhoz hero és termék előnézet komponenseket készítettünk, és a `shadcn/ui-t` telepítettük a szebb UI komponensekhez.

Ekkor egy olyan hibával találkoztunk, hogy a képek nem töltődtek be, amit úgy oldottuk meg, hogy létrehoztunk egy `public/images/` mappát a placeholder képeknek. A `shadcn/ui` komponensekkel is voltak gondjaink, ehhez a fájlt kellett frissíteni.

Ezután a következő lépésünk az volt, hogy létrehoztuk a termék al-oldalakat, a Next.js App Router dinamikus routing funkcióját használva.
Létrehoztuk az `app/termekeink/[slug]/page.tsx` fájlt, és definiáltuk a termék típusokat TypeScript interfészekkel. A képek kezeléséhez a `Next.js` Image komponenst használtuk.

Egy olyan hibával találkoztunk, hogy a dinamikus route nem működött, amit a fájlnév `[slug]-ra` való átnevezésével és a params prop megfelelő típusozásával orvosoltuk. A statikus generálás is okozott fejfájást, mert a generateStaticParams függvény hibákat dobott, ezt a termékek külön fájlba helyezésével és importálásával oldottuk meg. A képek betöltésével is volt még egy probléma, ahol a fájlneveket és a relatív útvonalakat ellenőriztük.

Miután a termék aloldalak kész voltak, rájöttünk, hogy a hardcoded termékek nem praktikusak. Ezért a következő lépésünk az volt, hogy át álljunk egy adatbázis alapú megoldásra. Megterveztük a termékek tábla struktúráját SQL scriptekkel, és a Supabase-t választottunk az adatbázis kezelésére. Frissítettük a termékek lekérdezését a `lib/product-utils.ts` fájlban és az összes kapcsolódó oldalon.

Egy olyan hibával találkoztunk, hogy a Supabase kapcsolat nem működött, ami a környezeti változók beállításával javult. A `server-side` rendering is hibákat dobott, ehhez létre kellett hozni a `lib/supabase/server.ts` fájlt. A JSON parsing hibáknál pedig rájöttünk, hogy a Supabase automatikusan kezeli a JSONB típusokat.

Most, hogy az adatbázis integráció megvolt, a következő lépésünk egy email kapcsolati form létrehozása volt.
Először az EmailJS-t néztük, de végül egy saját API route-ot készítettem Nodemailerrel az `app/api/contact/route.ts` fájlban. Ehhez beállítottuk a környezeti változókat és a Gmail App Password-öt. Egy olyan hibával találkoztunk, hogy a Gmail App Password nem működött, mert engedélyeznünk kellett a 2-faktoros hitelesítést és új jelszót kellett generálni.

CORS hibákat is kaptam, de kiderült, hogy a `Next.js` API routes automatikusan kezelik ezt. Az email formátummal is volt gond, amit egy egyszerűbb HTML struktúrával oldottuk meg.
A Supabase teljes integrálásakor további kihívások merültek fel.

Létrehoztuk a Supabase projektet, beállítottuk a környezeti változókat és a kliens oldali, valamint szerver oldali klienseket. A termék táblát és az adatok betöltését SQL Editor-ben futtattuk le, és készítettünk egy `useProducts custom hook-ot` is.

Egy olyan hibával találkoztunk, hogy az RLS `(Row Level Security)` policy-k nem engedték a termékek lekérdezését, amit a Supabase dashboard-ban kellett megfelelően beállítani. JSONB mezőkkel és CORS hibákkal is voltak még problémák, de ezeket is sikerült orvosolni a beállítások ellenőrzésével.

Ezután a következő lépésünk az admin felület fejlesztése volt, ahol a termékeket tudjuk kezelni. Ez egy `app/admin/page.tsx` fájlban valósult meg, egyszerű jelszó alapú autentikációval. Implementáltam a termékek megtekintését, szerkesztését, törlését és hozzáadását CRUD műveletekkel a Supabase-on keresztül.
Egy olyan hibával találkoztunk, hogy bárki hozzáférhetett az admin felülethez, amit egyszerű jelszavas autentikációval és a session LocalStorage-ban való tárolásával oldottunk meg.
A JSON validáció is problémás volt a műszaki adatoknál, amit `try-catch` blokkal és validálással védtük ki. A kép URL validációt is hozzáadtuk, hogy ne legyenek hibás URL-ek.

Végül, a következő lépésünk a képek feltöltése és kezelése volt a Supabase Storage-ban. Létrehoztunk az `oknoplast-images bucket-et` a Supabase dashboard-ban és beállítottuk a `policy-ket`. Az admin felületet kiegészítettük egy fájlkezelő tab-bal, feltöltés, törlés és galéria kép kiválasztás funkciókkal. Egy olyan hibával találkoztunk, hogy a storage bucket nem létezett, vagy a fájl feltöltés nem működött megfelelően. Ezeket a `bucket` létrehozásával, megfelelő `policy-k` beállításával, egyedi fájlnevek generálásával és error handling-gel orvosoltunk. A kép URL-ek sem voltak mindig elérhetők, amit a bucket public beállításával és a getPublicUrl függvénnyel oldottuk meg.

Összefoglalva, a rendszer most teljes funkcionalitással rendelkezik. Sikerült egy `Next.js` alapú, reszponzív honlapot készítenünk, Supabase adatbázis integrációval, admin felülettel a termékek kezeléséhez, email kapcsolati formmal, kép feltöltéssel és kezeléssel, termék aloldalakkal és dinamikus routinggal. Ez egy izgalmas és tanulságos út volt, ahol rengeteget tanultunk a webfejlesztésről és a problémamegoldásról.

Telepítés

Útmutató hogyan lehet telepíteni

A `kinga-projekt.md` forrás említi, hogy a projekt dokumentációjában szerepelnie kell egy "Telepítés" fejezetnek, amely egy útmutatót tartalmaz arról, hogyan lehet lépésről lépésre telepíteni a szoftvert.

A Projekt Telepítése és Elindítása.

Ez a projekt egy modern weboldal, ami `Next.js-t` használ, ami egy nagyon népszerű technológia mostanában. Ahhoz, hogy el tudd indítani és használni tudd, néhány egyszerű lépésre lesz szükséged.

1. A Projekt Létrehozása (vagy meglévő letöltése)

◦ Először is, szükséged van egy `Next.j` s alkalmazásra. Ezt általában egy parancssorban kell megadni, mint például: `npx create-next-app@latest` project-neved. Ez a parancs egy új projektet hoz létre neked a következő beállításokkal, amik a projektben is használva vannak:

▪ **TypeScript:** Ez segít elkerülni a hibákat, mert pontosan tudja, milyen típusú adatokkal dolgozunk.
▪ **Tailwind CSS:** Ez egy nagyon gyors módja a kinézet (stílus) elkészítésének.
▪ **ESLint:** Ez ellenőrzi a kód minőségét, hogy szép és olvasható legyen.
▪ **App Router:** Ez egy újabb rendszer, ami a weboldal különböző részeinek (oldalak, útvonalak) kezelésére szolgál.
▪ **Import alias:** Ez azt jelenti, hogy könnyebben hivatkozhatsz a fájljaidra, például `@/` előtaggal.

◦ Ha a projektet valakitől kaptad meg (például letöltötted GitHubról), akkor ez a lépés már készen van, és csak le kell töltened a projektet a számítógépedre.

2. A Szükséges Programok (Függőségek) Telepítése

◦ A projekt rengeteg kis "segítő programot" használ, amik nélkül nem működne. Ezeket függőségeknek hívjuk.
◦ Ezeket a `package.json` fájlban látod felsorolva.
◦ Általában a `Next.js` projekt létrehozása után ezek automatikusan települnek. Ha egy már meglévő projektet töltöttél le, akkor csak futtasd a következő parancsot a projekt mappájában a terminálban: `npm install` (vagy `yarn install`, ha a `yarn-t` használod).

3. A Fejlesztői Szerver Elindítása

◦ Miután minden telepítve van, el kell indítanod a "fejlesztői szervert". Ez egy olyan program, ami futtatja a weboldaladat a számítógépeden, hogy láthasd, hogyan működik.
◦ Nyisd meg a projekt mappáját egy terminálban, és írd be: `npm run dev.`
◦ Ekkor el fog indulni a weboldal, és látni fogsz egy címet, valószínűleg `http://localhost:3000`. Ezt a címet beírva a böngésződbe, máris látni fogod a honlapodat!

Gyakori Kezdeti Problémák és Megoldásuk (ha találkozol velük)

Néha előfordul, hogy az első indításnál nem minden tökéletes, de ne ijedj meg, ezekre is van megoldás!

• **TypeScript hibák `@/`miatt:** Ha a TypeScript nem ismeri fel a `@/` előtagot az importoknál (azaz, amikor egy fájlt akarsz használni egy másikban), akkor valószínűleg egy kis beállításra van szükség a `tsconfig.json` fájlban. A forrás nem részletezi, mi volt a pontos beállítás, de általában a "`paths`" beállításokat kell ellenőrizni és szükség esetén kiegészíteni.

• **Tailwind CSS nem működött:** Ha a kinézet nem úgy néz ki, ahogy kellene, és a Tailwind stílusok nem működnek, akkor a `tailwind.config.ts` fájlban kell beállítani, hogy mely fájlokat figyelje a Tailwind, hogy tudja, hol használod a stílusokat. Ezt a content részben kell megtenni.

Ezekkel a lépésekkel el tudod indítani a projektet, és elkezdheted felfedezni, hogyan épül fel a weboldal!

Lépésről lépésre

Szia! Ha egy `Next.js` alapú weboldalt szeretnél elindítani, ami adatbázissal, képekkel és email küldéssel is dolgozik, akkor ez az útmutató neked szól. Lássuk, hogyan hozhatod létre és indíthatod el a projektet!

1. Projekt Alapok Beállítása
Először is létre kell hoznunk a `Next.js` projektet, és telepítenünk kell az alapvető dolgokat.

• **Next.js projekt létrehozása:** Hozz létre egy új Next.js alkalmazást. A források alapján ezt a parancsot használ-ná egy fejlesztő, és beállítana néhány dolgot már a kezdeteknél:

- **TypeScript:** A típus biztonság miatt fontos.
- **Tailwind CSS:** Hogy gyorsan tudjak stílusozni.
- **ESLint:** A kódminőség ellenőrzéséhez.
- **App Router:** Ez a Next.js 13+ új rendszere az oldalak kezelésére.
- **Import alias:** A `@/ prefix` a fájlok egyszerűbb importálásához.

• **Függőségek telepítése:** Miután a projekt létrejött, a `package.json` fájlban lévő csomagokat is telepíteni kell (valószínűleg a projekt létrehozása után ez automatikusan megtörténik, de ha valami hiányzik, akkor ezt a parancsot kell futtatni): `npm install (vagy yarn install).`

• **Fejlesztői szerver indítása:** Ezzel tudjuk majd megnézni, amit csinálunk a böngészőben.

- Futtasd a parancsot: `npm run dev (vagy yarn dev).`
- Ezután megnyithatod a böngésződben a `http://localhost:3000` címet.

2. Kezdeti Problémák Orvoslása
Ahogy elkezded, valószínűleg találkozol majd néhány hibával, de szerencsére van rájuk megoldás!

• **TypeScript import alias hibák:** Lehet, hogy a TypeScript nem ismeri fel a `@/ import alias-t.`
  ◦ **Megoldás:** Be kell állítani a `tsconfig.json` fájlban, hogy tudja, hol keresse.

• **Tailwind CSS nem működik:** Ha a stílusok nem jelennek meg.
  ◦ **Megoldás**: A `tailwind.config.ts` fájlban be kell állítani a content útvonalakat, hogy a Tailwind tudja, hol találja a használni kívánt osztályokat.

• **Hot reload hibák:** Előfordulhat, hogy a változtatások nem jelennek meg azonnal.
  ◦ **Megoldás:** Gyakran elég újraindítani a fejlesztői szervert: `npm run dev.`

• **React típusok hiánya:** Ha TypeScript hibákat kapsz a React komponenseknél.
  ◦ **Megoldás:** Telepíteni kell a szükséges típusokat: `npm install --save-dev @types/react @types/react-dom.`

• **`Shadcn/ui` komponensek beállítása:** Ha szép UI elemeket szeretnél használni.

  ◦ **Telepítsd a `shadcn/ui-t:`** valószínűleg `npx shadcn-ui@latest init` a parancs. Ez létrehozza a `components.json` fájlt és beállítja a konfigurációt.
  ◦ **Ezután telepítsd a szükséges komponenseket,** amikre szükséged van. Ha a `Shadcn/ui` komponensek nem működnek, frissítsd a `tailwind.config.ts` fájlt a megfelelő beállításokkal.

3. Supabase Adatbázis és Képkezelés Beállítása
A projekt adatokat tárol és képeket kezel, ehhez a Supabase-t használjuk.

• Supabase projekt létrehozása és csomagok telepítése:

  ◦ Menj a `supabase.com` oldalra, és hozz létre egy új projektet. Válassz egy régiót (pl. Europe West), és várj, amíg befejeződik az inicializálás.
  ◦ Telepítsd a szükséges Supabase csomagokat a projektbe.

• Környezeti változók beállítása: Ezek nagyon fontosak a Supabase kapcsolathoz!

  ◦ Hozd létre a `.env.local` fájlt a projekted gyökerében.
  ◦ Add hozzá ezeket az értékeket, amiket a Supabase projektedből kapsz meg:
    ▪ `NEXT_PUBLIC_SUPABASE_URL=a_te_supabase_url_ed`
      ▪ `NEXT_PUBLIC_SUPABASE_ANON_KEY=a_te_supabase_anon_kulcsod`
      ▪ `DATABASE_URL=a_te_adatbazis_kapcsolati_stringed` (ezt a szerver oldali klienshez kellhet).

• Adatbázis táblák létrehozása és adatok betöltése:

  ◦ A Supabase SQL Editor-ben futtasd le a `scripts/create-products-table.sql` scriptet a termékek tábla létrehozásához.

◦ Futtasd le a `scripts/seed-products-data.sql` scriptet a kezdeti adatok betöltéséhez.

• **RLS (Row Level Security) beállítása:** Ha nem tudod lekérdezni a termékeket, valószínűleg az RLS `policy-k` miatt van.

◦ Ellenőrizd és frissítsd a policy-ket a Supabase dashboard-ban, hogy engedélyezzék a lekérdezéseket.

• **Storage bucket létrehozása és policy beállítása:** Ide kerülnek a képek.
◦ A Supabase Storage-ben hozz létre egy `oknoplast-images` nevű bucket-et.
◦ Állítsd be a megfelelő policy-ket a bucket-hez, hogy a képek publikusan elérhetők legyenek, és fel lehessen tölteni őket.

4. Email Küldés Beállítása

A kapcsolati űrlaphoz email küldésre van szükségünk.

• **Nodemailer telepítése**: Ezt használjuk az e-mailek küldéséhez.
◦ *Telepítsd:* `npm install nodemailer`.
• **Gmail App Password létrehozása**: Ez egy speciális jelszó a Google fiókodban, amit alkalmazások használhatnak.
◦ Jelentkezz be a Google fiókodba, és menj a "Biztonság" szekcióba.
◦ Engedélyezd a 2-faktoros hitelesítést, ha még nem tetted meg.
◦ A "App passwords" résznél hozz létre egy új "App Password"-ot a "Mail" alkalmazáshoz. Másold ki pontosan a generált jelszót.
• Környezeti változók az e-mailhez:
◦ A `.env.local` fájlban add meg ezeket:
▪ `EMAIL_USER=a_te_gmail_cimed@gmail.com`
▪ `EMAIL_PASS=a_generalt_app_jelszavad`
▪ `NEXT_PUBLIC_EMAIL_TO=aki_kapja_az_emailt@pelda.com`

Ezeket a lépéseket követve el tudod indítani a projektet a `Next.js`-szel, Supabase-szel és email küldési funkcióval. Ne feledd, az egyes lépések során (pl. Supabase URL-ek vagy Gmail App Password) saját adataidat kell majd beillesztened!

Összefoglaló

Sziasztok! Ahogy befejeztük ezt a projektet az Oknoplast Debrecen számára, visszatekintve látjuk, mennyi mindent tanultunk és milyen sok kihívással néztünk szembe. Célunk egy **modern, reszponzív weboldal** létrehozása volt, ami segíti az ügyfélszerzést, bemutatja a termékeket és leegyszerűsíti az árajánlatkérési folyamatokat. Kezdő programozóként az volt

a célunk, hogy egy olyan rendszert építsünk, ami nemcsak jól néz ki, hanem funkcionálisan is stabil és könnyen kezelhető.

A projekt során a **Next.js modern React framework-et** választottunk, mert sok hasznos funkciót biztosít, és a stabilitása, valamint az elterjedtsége meggyőzött. A fejlesztést **TypeScript-tel** a típusbiztonságért, **Tailwind CSS-sel** a gyors stílusozásért, **ESLint-tel** a kódminőség ellenőrzéséért és az **új App Router-rel** az oldalak kezeléséért állítottunk be.

A Tanuló Útunk és a Megoldott Problémák:

Már az elején belefutottunk néhány "kezdő" hibába, amikből sokat tanultunk:

- **Telepítési és beállítási gondok**: A TypeScript eleinte nem ismertük fel a `@/` import alias-t, amit a `tsconfig.json` fájl beállításával orvosoltuk. A Tailwind CSS sem működött elsőre, itt a `tailwind.config.ts` fájlban kellett beállítani a `content` útvonalakat.
- **Alapvető funkciók**: Az első "Hello World" oldal készítésekor a `hot reload` nem működött megfelelően, ami a fejlesztői szerver újraindításával oldódott meg. Később a React típusokkal kapcsolatos TypeScript hibákhoz további csomagokat kellett telepítenünk.
- **Layout és UI**: A navigációs sáv és a lábléc kialakításakor a Lucide React ikonok nem töltődtek be, és a mobil menü sem záródott be automatikusan. Ezeket megfelelő `npm` csomagok telepítésével és egy `onClick` esemény hozzáadásával sikerült megoldani. A `shadcn/ui` komponensek integrálásakor is kellett frissítenünk a `tailwind.config.ts` fájlt, hogy minden a helyére kerüljön.
- **Képek és dinamikus oldalak**: Többször is találkoztunk azzal a problémával, hogy a képek nem töltődtek be. Rájöttünk, hogy a `public/images/` mappa létrehozása és a képek oda helyezése, valamint a fájlnevek és relatív útvonalak ellenőrzése kulcsfontosságú. A dinamikus termék aloldalak ([slug].tsx) beállításakor a fájlnév pontossága és a `params` prop megfelelő típusozása jelentett kihívást.
- **Adatbázis integráció (Supabase)**: A hardcoded termékekről való átállás adatbázisra volt az egyik legnagyobb lépés. A Supabase kapcsolattal eleinte gondjaink voltak a környezeti változók hiánya miatt. A `server-side rendering` hibák miatt külön `lib/supabase/server.ts` fájlt kellett létrehoznunk. A `Row Level Security (RLS)` policy-k beállítása a Supabase dashboardban pedig elengedhetetlen volt az adatok lekérdezéséhez.
- **Email kapcsolati űrlap**: Az email küldéshez a Gmail App Password beállítása okozott fejtörést. Rájöttünk, hogy a 2-faktoros hitelesítés engedélyezése és egy pontosan kimásolt, új alkalmazásjelszó szükséges. Szerencsére a Next.js API routes automatikusan kezelte a CORS hibákat.
- **Admin felület**: Az admin felület biztonsága érdekében egy egyszerű jelszó alapú autentikációt implementáltam, a `LocalStorage-ban` tárolva a session-t. A `JSON validáció` a műszaki adatoknál `try-catch` blokkal és `URL validációval` vált stabillá.

- **Képfeltöltés és kezelés**: A Supabase Storage-ba történő képfeltöltésnél a `bucket` megléte, a megfelelő `policy-k` beállítása, az egyedi fájlnevek generálása és a `getPublicUrl` függvény használata mind kritikus volt a sikeres működéshez.

Elért eredmények és tanulságok:

Összefoglalva, a rendszer most **teljes funkcionalitással rendelkezik**:

- **Next.js alapú honlap**
- **Reszponzív design**
- **Supabase adatbázis integráció**
- **Admin felület termékek kezeléséhez** (CRUD műveletekkel, jelszóval védve)
- **Email kapcsolati form** (Nodemailer és Gmail App Password segítségével)
- **Kép feltöltés és kezelés** (Supabase Storage-ban)
- **Termék aloldalak** (dinamikus routinggal)

Ez a projekt egy **izgalmas és rendkívül tanulságos utazás** volt számunkra. Rengeteget tanultunk a webfejlesztésről, a hibakeresésről és a problémamegoldásról. Rájöttünk, hogy a részletekre való odafigyelés mennyire fontos, és hogy minden "hiba" egy új tanulási lehetőség. A legfontosabb talán az volt, hogy **sosem szabad feladni**, még akkor sem, ha egy apróságnak tűnő beállítás órákig tartó fejtörést okoz.

Jövőbeli tervek:

A rendszer most már stabilan működik, de a jövőben még tovább fejlesztenénk: például SEO- és teljesítményoptimalizálást végeznénk, további admin funkciókat építenénk be, vagy akár egy mobil alkalmazás fejlesztésén is elgondolkodnánk.

Olyan érzés ez, mintha egy bonyolult kirakós játékot raktunk volna össze. Minden egyes darab, minden egyes funkció, amit beépítettük, egy-egy problémamegoldás volt, ami végül egy teljes, működőképes képpé állt össze.