**JEGYZŐKÖNYV**

Web technológiák I.

Féléves feladat

Teendőlista- és feladatkezelő alkalmazás

Készítette: **Ilyó-Kovács Levente**

Neptunkód: **NLFUA8**

Dátum: **2024. 12. 08.**

Tartalomjegyzék

[1. Bevezetés 1](#_Toc184568162)

[1.1 A projekt célja 1](#_Toc184568163)

[1.2 A projekt jelentősége és kontextusa 2](#_Toc184568164)

[2. A rendszer architektúrája 3](#_Toc184568166)

[2.1 Frontend és backend integráció 3](#_Toc184568167)

[Kapcsolódási folyamatok 3](#_Toc184568168)

[7. JSON formátumú válaszok 3](#_Toc184568169)

[2.2 Adatbázis és infrastruktúra 4](#_Toc184568170)

[Adatbázis felépítése 4](#_Toc184568171)

[Táblák és adatszerkezet 4](#_Toc184568172)

[A GUID (Globally Unique Identifier) előnyei 5](#_Toc184568173)

[Adatbázis-műveletek 5](#_Toc184568174)

[3. Frontend 6](#_Toc184568175)

[3.1 A React alkalmazás általános struktúrája 6](#_Toc184568176)

[3.2 Komponens alapú tervezés 7](#_Toc184568177)

[Példa komponens: Header 7](#_Toc184568178)

[Oldalspecifikus komponens: MainPage 9](#_Toc184568179)

[3.3 A React Router szerepe és konfiguráció 9](#_Toc184568180)

[3.4 Frontend API integráció 10](#_Toc184568181)

[4. Backend 13](#_Toc184568182)

[4.1 Kontroller szintű működés 13](#_Toc184568183)

[4.2 Szolgáltatási réteg és üzleti logika 14](#_Toc184568184)

[4.3 Adatbázis-kezelés és tárolt adatok 14](#_Toc184568185)

[5. MSSQL Adatbázis Létrehozása 15](#_Toc184568186)

[Összefoglalás 16](#_Toc184568187)

# 1. Bevezetés

## 1.1 A projekt célja

A projekt célja egy modern, skálázható, és könnyen karbantartható webshop rendszer létrehozása, amely megfelel a mai e-kereskedelmi platformok által támasztott technológiai és felhasználói követelményeknek. A rendszer fejlesztése során a következő konkrét célokat tűztük ki:

1. **Felhasználóbarát felület kialakítása**  
   A frontend egy könnyen kezelhető, vizuálisan vonzó interfészt nyújt, amely elősegíti a vásárlók számára a gyors navigációt, keresést és vásárlást. A **React.js** technológia használata a **Google** által nyújtott **Material UI** komponensekkel dinamikus és reszponzív felhasználói élményt biztosít.
2. **Adatbiztonság és adatkezelés**  
   A backend biztosítja a vásárlók adatainak biztonságos kezelését, beleértve a bejelentkezési adatokat, vásárlási történeteket, és a termékekkel kapcsolatos adatokat. A **.NET Core** és az **MSSQL Server** együttese garantálja az adatok integritását és gyors elérhetőségét. Az adatbázis “salted & hashed” módon tárolja a jelszavakat, ezzel a mai standardnak megfelelő biztonságos adattárolást tesz lehetővé.
3. **Skálázhatóság és rugalmasság**  
   A **Docker** konténerizációval a rendszer könnyen telepíthető és skálázható, hogy megfeleljen a növekvő felhasználói igényeknek. Ez különösen fontos az e-kereskedelem dinamikusan változó piacán.
4. **RESTful API implementáció**  
   A backend **RESTful API-k** biztosításával lehetővé teszi az egyszerű frontend-backend kommunikációt. Ez a technológia támogatja a **HTTPS (secure)** kapcsolat biztosítását, így a bejövő és küldött adatok titkosíthatóak.

A projekt végső célja, hogy egy olyan rendszert hozzunk létre, amely a fejlesztők számára átlátható és könnyen bővíthető, míg a felhasználók számára intuitív, gyors és megbízható vásárlási élményt kínál.

## 1.2 A projekt jelentősége és kontextusa

Az e-kereskedelem növekvő szerepe az életünkben megkérdőjelezhetetlen. Az internetes vásárlás mára nem csupán kényelmi funkció, hanem sok esetben az elsődleges vásárlási forma.

1. **A piaci versenyhelyzet**  
   Számtalan webshop érhető el világszerte, így egy új platform létrehozásához kiemelkedő felhasználói élményt kell nyújtani, amely gyors, stabil és esztétikailag is versenyképes. Az egyszerű vásárlási folyamat, gyors betöltési idő, és mobilbarát kialakítás mind kritikusak a piaci sikerhez.
2. **Technológiai trendek**  
   A **React.js** és a **.NET Core** a mai fejlesztői ökoszisztéma egyik legnépszerűbb és legelterjedtebb eszközei. A választásukat az motiválta, hogy biztosítsák a skálázhatóságot, a kód újrafelhasználhatóságát és a fejlesztési idő optimalizálását.
3. **Felhasználói elvárások**  
   A modern vásárlók gyors válaszidőt, könnyű navigációt és megbízható fizetési rendszereket várnak el. A projekt során ezeket az elvárásokat az előbb említett technológiák felhasználásával igyekeztem figyelembe venni.
4. **Skálázhatóság**  
   A projekt során figyelembe vettük a skálázhatóság igényét is. A **Docker** konténerizáció révén a rendszer könnyen telepíthető és hordozható, minimalizálva az üzemeltetési költségeket.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedAz általános e-kereskedelmi jelentőségen túl ez a projekt egy példa arra, hogyan lehet modern technológiákkal felépíteni egy olyan rendszert, amely egyszerre szolgálja a felhasználók igényeit és a fejlesztők szakmai fejlődését. A webshop működése, funkcionalitása és infrastruktúrája tükrözi azokat az alapelveket, amelyek a jövő technológiai megoldásait meghatározzák.

# 2. A rendszer architektúrája

A webshop rendszer architektúrája három fő komponensből áll: a **frontendből**, a **backendből**, és az **adatbázisból**. Ezek a komponensek külön rétegekben helyezkednek el, de szorosan együttműködnek annak érdekében, hogy egy jól működő, modern e-kereskedelmi rendszert alkossanak.

## 2.1 Frontend és backend integráció

A frontend és a backend közötti integráció **RESTful API-k** segítségével valósul meg. A rendszer kliens-szerver architektúrában működik, ahol a **React**-alapú frontend küldi a kéréseket a **C# ASP.NET Core** backend felé, amely ezeket feldolgozza és az eredményt visszaküldi.

### Kapcsolódási folyamatok

1. **HTTP protokoll használata**  
   A frontend az API végpontokat **HTTP protokollon** keresztül éri el. A kérések lehetnek GET, POST, PUT, vagy DELETE metódusok, amelyek egy-egy konkrét műveletet reprezentálnak.
2. **Axios vagy Fetch API használata**  
   A React frontendben az adatlekérések és adatküldések általában az **Axios** vagy **Fetch API** segítségével történnek. Példa egy termékek lekérdezésére:

* static async getAll() {  
   const response = await fetch(this.BASE\_URL);  
   return await response.json();  
  }

### JSON formátumú válaszok

A backend által visszaküldött adatok **JSON formátumban** érkeznek. Ez a formátum egyszerűen kezelhető JavaScript-ben, és kompatibilis a legtöbb modern frontend technológiával.

#### Előnyei:

* **Modularitás**: A frontend és a backend külön rétegekben található, így egymástól függetlenül fejleszthetők.
* **Rugalmasság**: Az API-t más rendszerek is használhatják, nemcsak a React frontend.
* **Egyszerű karbantarthatóság**: A RESTful API lehetővé teszi a backend logika elkülönítését, ami csökkenti az egyes komponensek frissítésének kockázatát.

#### Működési példa

1. A felhasználó a frontend felületen kiválaszt egy terméket.
2. A frontend egy GET /api/products/{id} kérést küld a backend felé.
3. A backend feldolgozza a kérést, lekérdezi az adatbázist, majd visszaküldi a termék részletes adatait **JSON formátumban**.
4. A frontend megjeleníti az adatokat a felhasználónak egy dinamikus oldalon.

## 2.2 Adatbázis és infrastruktúra

### Adatbázis felépítése

A rendszer adatbázisa egy **Microsoft SQL Server**, amely **Docker konténerben** fut. Az SQL Server választása azért előnyös, mert:

* **Erőteljes és megbízható tranzakciókezelést biztosít.**
* **Támogatja a relációs adatbázis-modelleket**, amelyek ideálisak egy webshop rendszernél.
* **Széles körben használt vállalati környezetben**, így hosszú távon is fenntartható.

### Táblák és adatszerkezet

Az adatbázis több táblából áll, amelyek egymással relációban állnak:

#### 1. Products tábla

Ez a tábla tárolja a termékek adatait, amelyek a webshopban elérhetők.

##### Oszlopok:

* **ID**:  
  Globálisan egyedi azonosító (**GUID**), amely az egyes termékeket azonosítja. Ez a formátum biztosítja, hogy az azonosító minden környezetben egyedi maradjon.
* **Title**:  
  A termék neve (pl. *“Samsung Galaxy S24 Ultra”*).
* **Body**:  
  A termék részletes leírása.
* **Price**:  
  A termék ára, **decimális formátumban** tárolva.
* **Category**:  
  A termék kategóriája (pl. *“Smartphone”*).
* **Tags**:  
  Kulcsszavak a termékhez (pl. *“phone”*, *“android”*).
* **ImageData**:  
  A termék képadatainak **bináris formátuma** (encoded base64).

#### 2. Users tábla

Ez a tábla tartalmazza a webshop felhasználóinak adatait, beleértve a hitelesítéshez szükséges információkat.

##### Oszlopok:

* **ID**:  
  Globálisan egyedi azonosító (**GUID**), amely az egyes felhasználókat azonosítja.
* **Name**:  
  A felhasználó neve.
* **Email**:  
  A felhasználó email címe, amely a bejelentkezéshez használt.
* **Password**:  
  A **PBKDF2**-vel generált, hexadecimális formátumú jelszó hash.
* **Salt**:  
  A jelszó hasheléséhez használt véletlenszerű érték.
* **Role**:  
  A felhasználó jogosultsági szintje (pl. *“admin”*, *“customer”*).

#### 3. UserQuestions tábla

Ez a tábla rögzíti a felhasználók által feltett kérdéseket, valamint azok válaszait.

##### Oszlopok:

* **ID**:  
  **GUID**, amely az egyes kérdéseket azonosítja.
* **Question**:  
  A felhasználó által feltett kérdés szövege.
* **Answer**:  
  Az adminisztrátor által adott válasz (ha van).
* **UserId**:  
  Egy külső kulcs, amely a kérdést feltevő felhasználóra mutat.

### A GUID (Globally Unique Identifier) előnyei

A GUID egy 128 bites érték, amelyet világszerte egyedinek terveztek. A rendszerben a GUID használata biztosítja:

* **Ütközésmentesség**:  
  Az azonosítók egyediek maradnak, még elosztott rendszerekben is.
* **Biztonság**:  
  Az egyedi azonosítók nehezebben megjósolhatók, mint az egyszerű, növekvő egész számok.
* **Rugalmasság**:  
  Könnyen használható adatbázis-replikáció esetén is, mivel az azonosítók nem ütköznek.

### Adatbázis-műveletek

Az adatbázis kezelését az **Entity Framework Core** végzi. Ez egy **.NET alapú ORM (Object-Relational Mapping)** eszköz, amely megkönnyíti az adatbázis-műveletek végrehajtását.

#### Példa egy termék lekérdezésére:

public override Task<IEnumerable<Product>> GetAllAsync()  
{  
 IEnumerable<Product> products = myDbContext.Products;  
 return Task.FromResult(products);  
}  
  
Ez a metódus aszinkron módon kérdezi le a termékek adatait, ami növeli a rendszer teljesítményét.

### Dockerizált infrastruktúra

#### Kapcsolódás az adatbázishoz

Az adatbáziskapcsolat az \*\*appsettings.json\*\* fájlban van definiálva:

"ConnectionStrings": {  
 "ConnectionString": "Server=localhost,1433;Initial Catalog=uniWebShop;Persist Security Info=False;User ID=sa;Password=1Secure\*Password1;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=True;Connection Timeout=30;"  
}

Ez a beállítás határozza meg a szerver elérhetőségét, a használt adatbázist és a hitelesítési adatokat.

#### Adatbiztonság

A rendszer adatbiztonságáról az alábbiak gondoskodnak:

* **Jelszóhashelés**: A felhasználói jelszavakat hash-eléssel (pl. bcrypt) tárolják.
* **Adatbázis-hozzáférés korlátozása**: Csak a szükséges szolgáltatások érhetik el az adatbázist.
* **Docker hálózat**: Az adatbázis és a backend egy privát Docker-hálózatban kommunikál, csökkentve a támadások kockázatát.

# 3. Frontend

### 3.1 A React alkalmazás általános struktúrája

A frontend **React** technológiára épül, amely modern, komponens alapú megközelítést kínál. Az alkalmazás jól strukturált, a különböző funkciók és modulok elkülönítése megkönnyíti a fejlesztést és a karbantartást.

#### Mappa struktúra:

* **src**: Az alkalmazás forráskódját tartalmazza.
  + **pages**: Az egyes oldalak komponensei, például a főoldal, termékoldal, vagy profiloldal.
  + **common-components**: Újrafelhasználható komponensek, például fejléc (*Header*), lábléc (*Footer*).
  + **api**: A backenddel való kommunikációhoz szükséges API-hívások definíciói.
  + **context-providers**: Állapotkezelést biztosító modulok, például felhasználói adatok vagy vásárlói kosár.
  + **App.js**: Az alkalmazás fő belépési pontja, amely a navigációt és a komponensek összefogását biztosítja.
  + **index.js**: Az alkalmazás inicializálását végző fájl.

Ez a struktúra biztosítja az egyszerű bővíthetőséget és karbantarthatóságot, valamint a logikai funkciók jól elkülönülnek egymástól.

### 3.2 Komponens alapú tervezés

A React egyik legnagyobb előnye a komponens alapú tervezés, amelyet ez a projekt is maximálisan kihasznál.

#### Előnyei:

* **Modularitás**:  
  A komponensek önállóak, így újrafelhasználhatók különböző helyeken.
* **Olvashatóság**:  
  A kód logikus felépítése javítja az átláthatóságot.
* **Egyszerű karbantartás**:  
  Az egyes funkciók könnyen kezelhetők és bővíthetők külön komponensek szintjén.

A komponens alapú megközelítés hatékonyan támogatja a modern webes fejlesztési elveket, és biztosítja az alkalmazás skálázhatóságát.

### Példa komponens: Header

return (  
 <>  
 <StyledHeader>  
 <StyledLogo  
 src="/images/unilogo.png"  
 alt="logo"  
 onClick={navigateToMainPage}  
 />  
 <StyledSearchDiv>  
 <MUIButton  
 variant="contained"  
 color="red"  
 sx={{ width: 1 }}  
 onClick={navigateToSearchPage}  
 >  
 Go to search page  
 </MUIButton>  
 </StyledSearchDiv>  
 <Card />  
 <StyledLinks>  
 <MUIButton variant="contained" onClick={navigateToFaqPage} color="red">  
 FAQ  
 </MUIButton>  
 <MUIButton  
 variant="contained"  
 onClick={navigateToAboutUsPage}  
 color="red"  
 >  
 About Us  
 </MUIButton>  
 <MUIButton  
 variant="contained"  
 onClick={navigateToContactUsPage}  
 color="red"  
 >  
 Contact  
 </MUIButton>  
 {loggedInUser?.id === undefined && (  
 <LoginButton setLoggedInUser={setLoggedInUser}>  
 Log in / Sign up  
 </LoginButton>  
 )}  
 {loggedInUser?.id !== undefined && (  
 <ProfileButton  
 setLoggedInUser={setLoggedInUser}  
 loggedInUser={loggedInUser}  
 />  
 )}  
 <ShoppingCart />  
 </StyledLinks>  
 </StyledHeader>  
 </>  
);

#### Leírás

Ez a komponens egy statikus navigációs menüt valósít meg, amely minden oldalon elérhető.

#### Főbb funkciók:

* **Logó**: A felhasználót a főoldalra irányítja.
* **Keresés**: Keresési oldalra navigáló gomb.
* **Navigációs linkek**:
  + **FAQ**: Gomb a GYIK oldalra.
  + **About Us**: Gomb az “Rólunk” oldalra.
  + **Contact**: Gomb a kapcsolatfelvételi oldalra.
* **Felhasználói funkciók**:
  + Ha a felhasználó nincs bejelentkezve, megjelenik a *Log in / Sign up* gomb.
  + Ha a felhasználó be van jelentkezve, megjelenik a *ProfileButton*.
* **Vásárlói kosár**: Ikon a kosár tartalmának elérésére.
* **Dinamikus működés**: Képes kezelni a bejelentkezett és nem bejelentkezett állapotokat is.

### Oldalspecifikus komponens: MainPage

const MainPage = (props) => {  
 return (  
 <StlyedMainPageDiv>  
 <Banner />  
 <BestDealsPage />  
 </StlyedMainPageDiv>  
 );  
};

#### Leírás

A **MainPage** komponens egy konténerként működik, amely összefogja az oldalt alkotó további komponenseket, mint például a **Banner** és a **BestDealsPage** (terméklista).

### 3.3 A React Router szerepe és konfiguráció

A **React Router** segítségével az alkalmazás egyszerű és dinamikus navigációt valósít meg az oldalak között.

#### Router konfiguráció

function App() {  
 return (  
 <ThemeProvider theme={MainTheme}>  
 <BodyContainer>  
 <BrowserRouter>  
 <Header />  
 <Routes>  
 <Route path="/" element={<MainPage />} />  
 <Route path="/FAQ" element={<FaqPage />} />  
 <Route path="/About-Us" element={<AboutUsPage />} />  
 <Route path="/Contact" element={<ContactPage />} />  
 <Route path="/Search" element={<SearchPage />} />  
 <Route path="/Shop" element={<ShopPage />} />  
 <Route path="/Checkout" element={<CheckoutPage />} />  
 <Route path="/AnswerQuestion" element={<AnswerQuestionPage />} />  
 <Route path="/Profile:id" element={<ProfilePage />} />  
 <Route path="/Product/:id" element={<ProductPageWithContext />} />  
 <Route path="/AddProduct" element={<AddProductPageWithContext />} />  
 </Routes>  
 </BrowserRouter>  
 <Footer />  
 </BodyContainer>  
 </ThemeProvider>  
 );  
}

#### Útvonal típusok

* **Statikus útvonalak**  
  Például:
  + Főoldal: /
  + GYIK oldal: /FAQ
  + “About Us” oldal: /About-Us
* **Dinamikus útvonalak**  
  Például:
  + /Product/:id: Egyedi termékek részleteinek megjelenítésére szolgál.
  + /Profile:id: Felhasználói profiloldal dinamikus paraméterekkel.

#### Újrafelhasználható komponensek

* **Header**: Biztosítja a globális navigációt.
* **Footer**: Az alkalmazás alsó részén elhelyezkedő tartalom.

#### Előnyei:

* **Egyszerű bővíthetőség**: Új útvonalak egyszerűen hozzáadhatók a meglévő struktúrához.
* **Dinamikus paraméterek**: Lehetővé teszik az egyedi adatmegjelenítést, például termékinformációkat vagy felhasználói adatokat.
* **Globális navigáció**: A statikus komponensek, például a Header és Footer, minden oldalról elérhetők, egységes felhasználói élményt biztosítva.

### 3.4 Frontend API integráció

Az **API-hívások implementációja** az api mappában található, ahol külön modulok felelnek az egyes funkciókért. Az API-kat aszinkron metódusokkal valósították meg, amelyek a backend **RESTful endpointjaira** küldenek kéréseket.

#### Példa: ProductAPI.js

class ProductAPI {  
 static BASE\_URL = `${process.env.REACT\_APP\_BACKEND\_ADDRESS}/Products`;  
  
 static async getAll() {  
 const response = await fetch(this.BASE\_URL);  
 return await response.json();  
 }  
  
 static async getById(id) {  
 const response = await fetch(`${this.BASE\_URL}/${id}`);  
 return await response.json();  
 }  
  
 static async add(product) {  
 await fetch(this.BASE\_URL, {  
 method: "POST",  
 headers: { "Content-Type": "application/json" },  
 body: JSON.stringify(product),  
 });  
 }  
  
 static async remove(id) {  
 await fetch(`${this.BASE\_URL}/${id}`, { method: "DELETE" });  
 }  
  
 static async update(product) {  
 await fetch(this.BASE\_URL, {  
 method: "PUT",  
 headers: { "Content-Type": "application/json" },  
 body: JSON.stringify(product),  
 });  
 }  
}  
  
export default ProductAPI;

##### Leírás

* Az osztály biztosítja a termékek:
  + **listázását**: getAll
  + **részletezését**: getById
  + **új termék hozzáadását**: add

#### Felhasználók kezelése: UserAPI.js

const BASE\_URL = `${process.env.REACT\_APP\_BACKEND\_ADDRESS}/User`;  
  
class UserAPI {  
 static async getAll() {  
 const response = await fetch(BASE\_URL);  
 return await response.json();  
 }  
  
 static async getById(id) {  
 const response = await fetch(`${BASE\_URL}/${id}`);  
 return await response.json();  
 }  
  
 static async add(user) {  
 const response = await fetch(BASE\_URL, {  
 method: "POST",  
 headers: {  
 "Content-Type": "application/json",  
 },  
 body: JSON.stringify(user),  
 });  
 return response.status;  
 }  
  
 static async login(userData) {  
 const response = await fetch(`${BASE\_URL}/login`, {  
 method: "POST",  
 headers: {  
 "Content-Type": "application/json",  
 },  
 body: JSON.stringify(userData),  
 });  
 if (response.ok) {  
 return response.json();  
 }  
 return null;  
 }  
  
 static async remove(id) {  
 await fetch(`${BASE\_URL}/${id}`, {  
 method: "DELETE",  
 });  
 }  
  
 static async update(user) {  
 await fetch(BASE\_URL, {  
 method: "PUT",  
 headers: {  
 "Content-Type": "application/json",  
 },  
 body: JSON.stringify(user),  
 });  
 }  
}  
  
export default UserAPI;

##### Funkciók

* **Felhasználók listázása**: getAll
* **Egyedi felhasználó lekérése azonosító alapján**: getById
* **Új felhasználó hozzáadása**: add
* **Felhasználói bejelentkezés**: login
* **Felhasználó eltávolítása**: remove
* **Felhasználói adatok frissítése**: update

# 4. Backend

### 4.1 Kontroller szintű működés

A backend rendszer egyik fő komponense a **kontroller réteg**, amely az API végpontokat valósítja meg. A kontrollerek célja, hogy fogadják a klienstől érkező HTTP kéréseket, validálják azokat, és a megfelelő szolgáltatási rétegnek továbbítsák.

#### Kontroller architektúra

A rendszerben három fő kontroller található:

* **ProductsController**: A termékek kezeléséért felelős.
* **UserController**: Felhasználói műveleteket, például regisztrációt és bejelentkezést kezel.
* **UserQuestionController**: A felhasználói kérdések és válaszok kezelését biztosítja.

Ezek a kontrollerek az **ASP.NET Core** által biztosított [ApiController] attribútumot használják, amely megkönnyíti a RESTful API-k implementációját. Az útvonalak konfigurálását a [Route] attribútum végzi.

#### Főbb funkciók és működés

* **Adatok lekérdezése**:
  + Összes termék listázása.
  + Egy adott termék lekérése az azonosító alapján.
* **Adatok módosítása**:
  + Új termék hozzáadása.
  + Meglévő frissítése vagy törlése.
* **Felhasználói műveletek**:
  + Hitelesítés, regisztráció és profilkezelés.

A kontrollerek minimalizálják a logikát, és az üzleti műveleteket a szolgáltatási rétegnek delegálják. Ezáltal a kontroller réteg tisztán a klienssel való kommunikációra összpontosít.

### 4.2 Szolgáltatási réteg és üzleti logika

A **szolgáltatási réteg** a backend rendszer központi eleme, amely az üzleti logikát implementálja és az adatbázis-műveleteket koordinálja. Ez a réteg elválasztja az adatkezelést és az API-k implementációját, biztosítva a kód újrafelhasználhatóságát és tisztaságát.

#### Szolgáltatási réteg működése

Minden szolgáltatás egy adott entitáshoz kapcsolódik, például:

* **ProductService**: A termékekkel kapcsolatos összes műveletet tartalmazza.
* **UserService**: A felhasználók kezeléséért felelős.
* **QuestionService**: A kérdések és válaszok logikáját valósítja meg.

A szolgáltatások az **Entity Framework Core** segítségével hajtják végre az adatbázisműveleteket, minimalizálva a nyers SQL használatát.

### 4.3 Adatbázis-kezelés és tárolt adatok

Az alkalmazás adatbázisát egy **Microsoft SQL Server** biztosítja, amely egy **Docker konténerben** fut. Az adatok kezelését az **Entity Framework Core** végzi, amely egy modern ORM technológia a .NET-hez.

#### Adatbázis architektúra

A rendszer adatbázis-modellje moduláris és normalizált.

##### Főbb táblák:

* **Products**: A webshop termékeit tárolja.
* **User**: A regisztrált felhasználók adatait tartalmazza.
* **UserQuestions**: A felhasználók által feltett kérdéseket és válaszokat tárolja.

##### Táblák közötti kapcsolatok:

* **Felhasználó és kérdések kapcsolata**: Egy felhasználó (Users) több kérdést is feltehet, amit a UserQuestions tábla UserId idegen kulcs mezője biztosít, amely a Users tábla Id oszlopára hivatkozik.
* **Kérdések és válaszok kapcsolata**: Egy kérdéshez egy opcionális válasz (Answer) tartozhat a UserQuestions táblában, amely közvetlenül tárolja a válaszokat az adott kérdés mellett.

#### Migrációk és adatbázis-frissítések

Az **Entity Framework** migrációs rendszert biztosít, amely lehetővé teszi az adatbázis séma kezelését és frissítését.

**Példa egy migrációra:**

protected override void Up(MigrationBuilder migrationBuilder)  
{  
 migrationBuilder.AddColumn<byte[]>(  
 name: "ImageData",  
 table: "Products",  
 type: "varbinary(max)",  
 nullable: true);  
}

Ez a migráció egy új **ImageData** oszlopot ad hozzá a Products táblához.

# 5. MSSQL Adatbázis Létrehozása

Az MSSQL adatbázis **Docker alapú inicializálása** a projekt részét képező **create\_db.bat** fájl segítségével történik. Ez a megoldás automatizált, így minimális manuális beavatkozást igényel.

#### Folyamat lépései:

1. **Docker Konténer Indítása**  
   Az MSSQL szerver indítása az alábbi paraméterekkel történik:
   * **Felhasználónév és jelszó**:  
     Az SA felhasználó számára előre beállított jelszó.
   * **Volume**:  
     Az adatállományok egy konténer volume-on belül kerülnek tárolásra, biztosítva a perzisztenciát.
   * **Port**:  
     A szerver a **1433-as** porton érhető el.
2. **Elérhetőség Ellenőrzése**  
   A script ciklikusan próbál kapcsolódni az MSSQL szerverhez. Amint az adatbázis elérhető, a folyamat továbblép a következő lépésre.
3. **Migrációk Alkalmazása**  
   A rendszer az **Entity Framework** segítségével, a következő parancsot használva alkalmazza a migrációkat:

* dotnet ef database update
* Ez a parancs létrehozza és frissíti az adatbázis-struktúrát, a kódban meghatározott migrációk alapján.

#### Előnyei:

* **Automatizált folyamat**:  
  Csökkenti a manuális beavatkozás szükségességét, gyorsítva a fejlesztési és telepítési folyamatot.
* **Konzisztens környezet**:  
  Minden fejlesztő ugyanazt az adatbázis-környezetet kapja, minimalizálva az eltérésekből adódó hibákat.
* **Rugalmasság**:  
  Könnyen integrálható CI/CD pipeline-okba, támogatva a modern fejlesztési és telepítési eljárásokat.

# Összefoglalás

A projekt egy modern, skálázható, és könnyen karbantartható webshop rendszer létrehozására összpontosít, amely a legújabb technológiai megoldásokra épül. A React alapú frontend, az ASP.NET Core backend, és a Dockerrel támogatott MSSQL adatbázis egységes rendszerként működik, amely kielégíti mind a felhasználói, mind a fejlesztői igényeket. Az automatizált adatbázis-kezelés, a moduláris komponensfelépítés és a RESTful API-k biztosítják a bővíthetőséget, a karbantarthatóságot és a teljesítményoptimalizálást. Ez a megközelítés garantálja, hogy a rendszer hosszú távon is versenyképes és fenntartható maradjon az e-kereskedelem gyorsan változó világában.

### Indítási lépések:

* Az alkalmazás indításához szükséges a Docker Desktop feltelepítése, majd a create\_db.bat parancsfile indításával létrehozhatjuk az MSSQL konténerünket és lefuttathatjuk a migrációkat.
* Ezután a backend solution fileját megnyitva futtatnunk kell a projektet
* Majd a frontend mappán belül egy terminált használva futtassuk az npm i && npm run start parancsot.
* Ez le fogja tölteni a szükséges függőségeket majd elindítja a projektet fejlesztői módban.
* Az adatbázis feltöltéséhez használhatjuk az Azure Data Studio adatbáziskezelő szoftvert, mellyel beimportálható a backend mappán található dbcontent.bak file, miután felcsatlakoztunk az adatbázisra.