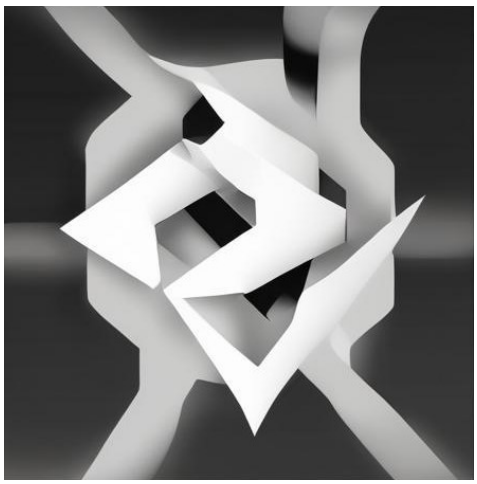
Rendezvény Követő



Készítette
Pusztai Gábor
Iváncsik Ábrahám
és Fehér Dominik
Szoftverfejlesztő és -tesztelő szak

Témavezető
Kerényi Róbert Nándor
oktató

Miskolc, 2024

Tartalomjegyzék

Bevezetés

A digitális világ folyamatos fejlődése során egyre fontosabbá vált a közösségek szerveződése és a személyes találkozások megkönnyítése, akár egy adott esemény keretében. Ezért döntöttünk úgy, hogy olyan rendszert tervezünk, amely egyszerűvé és átláthatóvá teszi az események létrehozását, meghívók küldését, valamint a részvételi visszajelzések kezelését. Az eseményszervezés kihívásai közé tartozik, hogy könnyen elérhető legyen a helyszín, minden résztvevő időben értesüljön a fontos részletekről, és a visszajelzések megfelelően legyenek kezelve. Az online platformok lehetővé teszik, hogy ezeket a problémákat egyetlen rendszerben oldjuk meg. Ez a projekt olyan funkciókat tartalmaz, mint a felhasználók regisztrációja és események létrehozása, amelyek egyaránt célozzák a személyes és professzionális eseményeket, például koncerteket, workshopokat, vagy előadásokat. Az egyszerűség érdekében a felhasználók könnyen bejelentkezhetnek, eseményeket hozhatnak létre, valamint interaktív módon visszajelzést adhatnak részvételükről. Az alkalmazás további előnyei közé tartozik a geolokációs integráció, amely lehetővé teszi, hogy a felhasználók pontosan meghatározzák az események helyszínét egy térképen keresztül. Ez a megoldás nemcsak a regisztrált, hanem a nem regisztrált felhasználók számára is biztosítja az eseményekre történő meghívás és részvétel lehetőségét, így szélesebb közönséget elérve. A projekt célja tehát egy olyan átfogó rendszer kialakítása, amely megkönnyíti az eseményszervezést és a résztvevők közti kommunikációt, egyúttal modern, interaktív és felhasználóbarát megoldásokat kínál. [1]

1. fejezet

Technológiai háttér

1.1. Relációs Adatbáziskezelő Rendszerek

A relációs adatbáziskezelő rendszerek (RDBMS) olyan szoftverek, amelyek az adatokat táblázatos formában tárolják, lehetővé téve az adatok közötti komplex kapcsolatok egyszerű kezelését. A relációs modell, amely a matematikai relációkon alapul, biztosítja, hogy az adatok logikusan és strukturáltan legyenek rendezve, ezáltal elősegítve az adatok hatékony keresését, lekérdezését és manipulálását.

Az RDBMS-ek legfontosabb jellemzője, hogy az adatok közötti kapcsolatokra fókuszálnak. Minden adatbázis egy vagy több táblából áll, ahol a táblák sorokból (rekordok) és oszlopokból (attribútumok) épülnek fel. A primér kulcs (egyedi azonosító) garantálja, hogy minden rekord egyedi legyen, míg az idegen kulcsok lehetővé teszik a táblák közötti kapcsolatok definiálását. Ezen kapcsolatok révén az RDBMS-ek lehetővé teszik az adatok integritásának fenntartását és a redundancia minimalizálását.

Az RDBMS-ek legfontosabb jellemzői közé tartozik a tranzakciókezelés, amely biztosítja az adatok konzisztenciáját. A tranzakciók atomitása, konzisztenciája, izolációja és tartóssága (ACID elvek) biztosítják, hogy egy műveletsorozat vagy teljes mértékben végrehajtódik, vagy egyáltalán nem, ezzel megelőzve az adatvesztést és a sérüléseket.

A legnépszerűbb relációs adatbáziskezelő rendszerek közé tartozik az Oracle, a Microsoft SQL Server, a PostgreSQL és a MySQL. Ezek a rendszerek széles körben használatosak különböző iparágakban, például a pénzügyi szolgáltatásokban, a kereskedelemben, az egészségügyben és a telekommunikációban. Az RDBMS-ek lehetővé teszik az adatok könnyű keresését és feldolgozását, így alapvető fontosságúak az üzleti intelligencia és az adatelemzés területén.

MySQL/MariaDB és PHPMyAdmin A MySQL és a MariaDB a legelterjedtebb nyílt forráskódú relációs adatbáziskezelő rendszerek közé tartoznak. A MySQL, amelyet 1995-ben alapítottak, gyorsan népszerűvé vált a webes alkalmazások körében. 2010-ben, amikor a Sun Microsystems-t az Oracle Corporation megvásárolta, felmerültek

aggályok a MySQL jövőjével kapcsolatban, így a MariaDB fejlesztése indult el, amely célja a MySQL funkcióinak és teljesítményének megőrzése és továbbfejlesztése.

A MySQL és a MariaDB közötti fő különbség az, hogy a MariaDB közösségi orientált, és folyamatosan új funkciókat és teljesítményjavításokat kínál. Mindkét rendszer nagy népszerűségnek örvend, mivel könnyen telepíthető, rugalmas, és nagy teljesítményt nyújt.

A MySQL és a MariaDB számos jellemzőt kínál, például:

1.Teljesítmény: Képesek nagy mennyiségű adat kezelésére és gyors lekérdezések végrehajtására. 2.Skálázhatóság: Könnyen bővíthetők, ami lehetővé teszi a nagy rendszerek kialakítását. 3.Biztonság: Támogatják a felhasználói jogosultságok és szerepkörök kezelését, így biztosítva, hogy csak a megfelelő felhasználók férhessenek hozzá az adatokhoz. 4.Tranzakciókezelés: Az ACID elveknek köszönhetően biztosítják az adatok integritását.

A PHPMyAdmin egy rendkívül népszerű webalapú alkalmazás, amely a MySQL és a MariaDB adatbázisok kezelésére szolgál. Az alkalmazás célja, hogy megkönnyítse a felhasználók számára az adatbázisokkal végzett műveletek elvégzését, anélkül hogy mélyebb technikai ismeretekre lenne szükség. Az intuitív felhasználói felület lehetővé teszi az adatok egyszerű kezelését, beleértve a táblák létrehozását, módosítását, törlését és lekérdezését.

A PHPMyAdmin főbb jellemzői a következők:

1.Grafikus felhasználói felület: Könnyen navigálható, lehetővé téve a felhasználók számára, hogy intuitív módon végezzenek műveleteket. 2.SQL lekérdezések végrehajtása: A felhasználók közvetlenül írhatnak és futtathatnak SQL lekérdezéseket, és megtekinthetik az eredményeket. 3.Adatbázisok importálása és exportálása: A PHPMyAdmin támogatja a különböző formátumokban történő adatimportálást és -exportálást, ami megkönnyíti az adatok mozgatását. 4.Felhasználói jogosultságok kezelése: A rendszergazdák képesek beállítani, hogy ki férhet hozzá az egyes adatbázisokhoz, ezzel biztosítva a biztonságot. 5.Adatok vizualizációja: A PHPMyAdmin grafikus ábrázolásokat kínál, amelyek segítségével a felhasználók könnyen megérthetik az adatok közötti kapcsolatokat.

A MySQL/MariaDB és a PHPMyAdmin kombinációja lehetővé teszi a fejlesztők és az adatkezelők számára, hogy hatékonyan kezeljék az adatbázisokat, és gyorsan reagáljanak az üzleti igényekre. A nyílt forráskódú természetük, a közösségi támogatás és a folyamatos fejlesztések miatt a MySQL és a MariaDB kiváló választás a modern adatbázis-kezelési igényekhez. Az ilyen rendszerek használata nemcsak a hatékonyságot növeli, hanem lehetővé teszi a felhasználók számára, hogy a legfrissebb technológiákat kihasználva fejlesszék alkalmazásaikat. Ezért döntötünk úgy , hogy ezeket használjuk a programunkhoz.

1.2. .NET Web API

A .NET Web API a Microsoft által fejlesztett keretrendszer, mely lehetővé teszi HTTP-alapú szolgáltatások létrehozását RESTful elvek mentén. Az alkalmazások fejlesztéséhez az ASP.NET Core Web API-t használják, amely a .NET Core platform része. Ez a technológia skálázható, jól tesztelhető és platformfüggetlen megoldást kínál API-k létrehozására.

Főbb jellemzői:

Egyszerűség: Kódolás közben kevés boilerplate kódra van szükség, így a fejlesztők gyorsan hozhatnak létre API-kat.

Integráció: Kiválóan integrálható különböző .NET alapú alkalmazásokkal és más platformokkal is.

Skálázhatóság: Alkalmas nagy volumenű, elosztott rendszerek kiszolgálására.

Biztonság: Erős támogatást nyújt különböző hitelesítési és jogosultsági rendszerekhez.

Az ASP.NET Core Web API lehetővé teszi a fejlesztők számára, hogy gyorsan és hatékonyan készítsenek webszolgáltatásokat, melyek könnyen karbantarthatók és bővíthetők. Az ilyen API-k segítségével különböző kliensek, például webes és mobil alkalmazások könnyedén hozzáférhetnek az adatokhoz és funkciókhoz.

1.3. A React Programozás Alapjai és Fejlettebb Fogalmai

Bevezetés A React egy JavaScript könyvtár, amelyet a Facebook fejlesztett ki 2013-ban. A React célja, hogy segítse a fejlesztőket dinamikus, interaktív felhasználói felületek (UI) építésében, különösen a webalkalmazások esetében. A könyvtár elterjedésével és folyamatos fejlődésével ma már szinte az iparági szabvánnyá vált a modern webalkalmazások fejlesztésében. A React alapelve az, hogy a felhasználói felületet komponens-alapú módon kell felépíteni, ahol minden komponens önálló, újrahasznosítható és kezelhető.

A React egyik legfontosabb előnye a virtuális DOM használata, amely lehetővé teszi az alkalmazás gyors és hatékony frissítését anélkül, hogy a teljes dokumentumot újra renderelni kellene. A React fejlesztése gyors, a kód karbantartása könnyű, és lehetővé teszi a dinamikus, interaktív alkalmazások gyors prototípusát.

1. React Alapfogalmak 1.1 Komponens-alapú fejlesztés A React alkalmazások a komponensek köré épülnek. A komponens egy önálló kódrészlet, amely egy adott funkciót vagy vizuális elemet képvisel a felhasználói felületen. Mivel a komponensek kis egységek, újrahasznosíthatók, és könnyen karbantarthatók, a kód tiszta és moduláris marad.

A komponensek lehetnek:

Funkcionális komponensek: Ezek a legegyszerűbb típusú komponensek, amelyek JavaScript függvényekből állnak. A React 16.8 verziójától kezdve funkcionális komponensek is rendelkezhetnek hookokkal, amelyek lehetővé teszik számukra az állapot kezelését és egyéb életciklus-műveletek végrehajtását. Osztály-alapú komponensek: A React korábbi verzióiban ezek a komponensek voltak a standardok, és lehetővé tették az állapotok kezelését, valamint a komponens életciklusának vezérlését. 1.2 JSX A React egyik legfontosabb jellemzője a JSX (JavaScript XML) szintaxis, amely lehetővé teszi, hogy HTML-t írjunk közvetlenül a JavaScript kódba. Ez a szintaxis egyszerűsíti a komponensek felépítését, és tiszta, olvasható kódot eredményez.

Példa JSX-re:

```
const element = <h1>Hello, world!</h1>;
```

A JSX kódot a React a háttérben JavaScript objektumokká fordítja, amiket aztán a virtuális DOM-ban kezel.

1.3 Virtuális DOM A React egy virtuális DOM-ot használ a weboldalak renderelésének optimalizálására. Amikor a komponens állapota vagy a props változik, a React először a virtuális DOM-ban végzi el az összes változást, majd összehasonlítja azt a valódi DOM-mal. Csak azokat az elemeket frissíti, amelyek ténylegesen változtak, ezáltal gyorsabbá és hatékonyabbá téve az alkalmazás működését.

2. Állapot és Props 2.1 Állapotkezelés (State) A state a React-ben egy komponens saját adatainak tárolására szolgál. Az állapot lehetővé teszi, hogy a komponens dinamikusan reagáljon a felhasználói interakciókra, például gombnyomásra, adatbevitelre, vagy bármely más eseményre. Az állapot kezelése a useState hook segítségével történik funkcionális komponensek esetén.

Példa egy egyszerű számláló alkalmazásra, amely az állapot kezelését mutatja be:

```
import React, { useState } from 'react';

function Counter() {
        const [count, setCount] = useState(0);

        return (
        <div>
        <p>Count: {count}</p>
        <button onClick={() => setCount(count + 1)}>Increment</button>
        </div>
        );
}
```

Ebben a példában a count az állapot, és a setCount egy függvény, amellyel módosíthatjuk az állapotot. Minden alkalommal, amikor az állapot frissül, a React újra

rendereli a komponenst.

2.2 Props A props (properties) egy olyan mechanizmus, amely lehetővé teszi, hogy adatokat adjunk át egy komponensnek a szülőkomponensből. A props-okat nem lehet módosítani a gyermek komponensben, csak olvasni.

Példa a props használatára:

```
function Greeting(props) {
        return <h1>Hello, {props.name}!</h1>;
}

function App() {
        return <Greeting name="Alice" />;
}
```

Itt az App komponens átadja az Alice értéket a Greeting komponensnek a name prop-on keresztül. A props segítségével a komponens adatokat kap, amelyek befolyásolják a megjelenített tartalmat.

3. Hookok A React 16.8-as verziójától kezdve a hookok jelentik a legmodernebb módot az állapot és életciklus-műveletek kezelésére funkcionális komponensekben. A hookok lehetővé teszik, hogy a komponensek állapotot kezeljenek és reagáljanak a különböző életciklus eseményekre anélkül, hogy osztályokat kellene használni.

A leggyakoribb hookok:

useState: Állapot kezelésére szolgál. useEffect: Mellékhatások kezelésére használják, mint például API hívások vagy események kezelése. useContext: A React kontextus API-ját használja, amely lehetővé teszi az adatok globális kezelését a komponensek között. useReducer: Összetettebb állapotkezeléshez használható, különösen olyan esetekben, amikor az állapot több részletből áll és a frissítések komplexek. Példa a useEffect hook használatára:

```
        import React, { useState, useEffect } from 'react';

        function Timer() {
        const [seconds, setSeconds] = useState(0);

        useEffect(() => {
        const interval = setInterval(() => {
        setSeconds(prev => prev + 1);
        }, 1000);

        return () => clearInterval(interval);
        // Takaritas, amikor a komponens eltunik
```

```
        }, []);

        return <h1>{seconds} seconds</h1>;
        }
```

Itt a useEffect minden egyes másodpercben frissíti a másodpercek számát, és amikor a komponens eltűnik, törli az időzítőt.

4. React Router és Navigáció A React Router egy olyan könyvtár, amely lehetővé teszi, hogy különböző URL-ekhez rendeljünk React komponenseket. Ezáltal egyoldalas alkalmazásokat (SPA) hozhatunk létre, ahol a navigáció nem okoz teljes oldalbetöltést, hanem csak az adott komponens frissül.

Példa a React Router alapvető használatára:

import BrowserRouter as Router, Route, Switch from 'react-router-dom';

```
function App() {
        return (
        <Router>
        <Switch>
        <Route path="/" exact>
        <Home />
        </Route>
        <Route path="/about">
        <About />
        </Route>
        </Switch>
        </Router>
        );
}
```

5. Teljes React Alkalmazás Példa Most nézzünk egy egyszerű alkalmazást, amely egy számlálót és egy gombot tartalmaz. A React alkalmazás alapját képező komponensek és állapotkezelés látható:

```
import React, { useState } from 'react';

function App() {
        const [count, setCount] = useState(0);

        return (
        <div>
        <h1>React Counter</h1>
        <p>Count: {count}</p>
```

```
        <button onClick={() => setCount(count + 1)}>Increment</button>
        <button onClick={() => setCount(count - 1)}>Decrement</button>
        </div>
        );
}
```

export default App; Ebben a példában a useState hook-ot használjuk a count változó és a hozzá tartozó frissítő függvény (setCount) kezelésére. Két gomb segítségével növelhetjük és csökkenthetjük a számlálót.

6. Összegzés A React a modern webalkalmazások egyik legnépszerűbb könyvtára, amely a komponens-alapú fejlesztést és a virtuális DOM használatát helyezi előtérbe. Az alapoktól kezdve a fejlettebb témákig, mint a hookok és az állapotkezelés, a React segít a gyors és dinamikus alkalmazások létrehozásában. Az alkalmazások modularitása és újrahasznosíthatósága biztosítja a könnyű karbantartást és skálázhatóságot, miközben a teljesítmény is kiváló a virtuális DOM-nak köszönhetően. Ahogy az ökoszisztéma folyamatosan fejlődik, a React továbbra is alapvető eszköze marad a front-end fejlesztők számára.

1.4. CRUD Műveletek

A .NET Web API-ban a CRUD (Create, Read, Update, Delete) műveletek azok az alapvető műveletek, amelyeket egy adatbázis-kezelő rendszerben kell implementálni az adatok manipulálására. A .NET Web API-ban minden CRUD művelethez egy HTTP metódust rendelünk:

Create: Az új adat létrehozása (HTTP POST).

Read: Az adat lekérése (HTTP GET).

Update: A meglévő adat módosítása (HTTP PUT/PATCH).

Delete: Az adat törlése (HTTP DELETE). Példa CRUD műveletek ASP.NET Core Web API-ban: Tegyük fel, hogy van egy Product nevű entitásunk, és létre akarunk hozni egy API-t, amely képes a termékek CRUD műveleteit végrehajtani.

```
public class Product
{
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
}
[Route("api/[controller]")]
[ApiController]
```

```
public class ProductsController : ControllerBase
{
        private static List<Product> _products = new List<Product>();

        // CREATE
        [HttpPost]
        public ActionResult<Product> CreateProduct(Product product)
        {
                product.Id = _products.Count + 1; // Gener ljunk egy ID-
                _products.Add(product);
                return CreatedAtAction(nameof(GetProduct), new { id = pro
        }

        // READ (Get all products)
        [HttpGet]
        public ActionResult<IEnumerable<Product>> GetProducts()
        {
                return Ok(_products);
        }

        // READ (Get a single product)
        [HttpGet("{id}")]
        public ActionResult<Product> GetProduct(int id)
        {
                var product = _products.FirstOrDefault(p => p.Id == id);
                if (product == null)
                return NotFound();
                return Ok(product);
        }

        // UPDATE
        [HttpPut("{id}")]
        public IActionResult UpdateProduct(int id, Product updatedProduct
        {
                var product = _products.FirstOrDefault(p => p.Id == id);
                if (product == null)
                return NotFound();
```

```
            product.Name = updatedProduct.Name;
            product.Price = updatedProduct.Price;
            return NoContent(); // Successful update (no content retu
        }

        // DELETE
        [HttpDelete("{id}")]
        public IActionResult DeleteProduct(int id)
        {
            var product = _products.FirstOrDefault(p => p.Id == id);
            if (product == null)
            return NotFound();

            _products.Remove(product);
            return NoContent(); // Successful deletion (no content re
        }
}
```

Ebben a példában a ProductsController osztály kezeli a CRUD műveleteket egy statikus listán, amely itt csak demonstrációs célt szolgál. Egy valós alkalmazásban ezek az adatok adatbázisban tárolódnának, például az Entity Framework segítségével.

2. API Tesztelés Swagger és Postman segítségével Miután a Web API-t implementáltuk, érdemes biztosítani, hogy az API végpontok megfelelően működnek. Erre szolgálnak a tesztelési eszközök, mint a Swagger és a Postman. Swagger: A Swagger (vagy más néven OpenAPI Specification) egy dokumentációs és tesztelési eszköz, amely lehetővé teszi a fejlesztők számára, hogy könnyedén dokumentálják és teszteljék a Web API-kat. Az ASP.NET Core projektek automatikusan generálhatnak Swagger dokumentációt az API végpontjairól.

Swagger Integrálása az ASP.NET Core Web API-ban Swagger csomag hozzáadása:

Az ASP.NET Core alkalmazáshoz hozzá kell adni a Swashbuckle.AspNetCore csomagot, amely lehetővé teszi a Swagger használatát. A NuGet Package Manager segítségével telepíthetjük:

bash

dotnet add package Swashbuckle.AspNetCore Swagger konfigurálása az Startup.cs fájlban:

A Swagger konfigurálását a Startup.cs fájlban végezzük el. A ConfigureServices metódusban regisztráljuk a Swagger szolgáltatásokat, míg a Configure metódusban beállítjuk a Swagger middleware-t.

```
public void ConfigureServices(IServiceCollection services)
{
        services.AddControllers();
        services.AddSwaggerGen(c =>
        {
                c.SwaggerDoc("v1", new OpenApiInfo { Title = "Product API
        });
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
        if (env.IsDevelopment())
        {
                app.UseSwagger();
                app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swag
        }

        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
                endpoints.MapControllers();
        });
}
```

Swagger UI használata:

Ha elindítjuk az alkalmazást, a Swagger automatikusan generálja a dokumentációt az API végpontokhoz. A Swagger UI elérhető lesz a következő URL-en: https://localhost:<port>/swa Itt tesztelhetjük az API végpontokat a böngészőn keresztül.

Postman: A Postman egy másik népszerű eszköz, amely lehetővé teszi az API-k tesztelését. A Postman segítségével egyszerűen küldhetünk HTTP kéréseket (GET, POST, PUT, DELETE) a végpontokhoz, és ellenőrizhetjük a válaszokat.

Postman Tesztelés API hívások létrehozása:

A Postman-ben új kérés létrehozásához válasszuk ki a megfelelő HTTP metódust (pl. POST, GET, PUT, DELETE) és írjuk be a kért URL-t (pl. http://localhost:5000/api/products).

Kérés testreszabása: GET kérés esetén egyszerűen kattintsunk a "Send" gombra, hogy lekérjük az adatokat. POST kérés esetén a kérés testében (Body) átadhatjuk a létrehozandó objektumot JSON formátumban, például:

```
{
        "name": "New Product",
        "price": 19.99
}
```

Válasz ellenőrzése:

A Postman a válaszokat különböző formátumokban (pl. JSON, XML) jeleníti meg, és automatikusan ellenőrzi az állapotkódokat (pl. 200 OK, 404 Not Found). Emellett aserciókat (ellenőrzéseket) is hozzáadhatunk, hogy automatikusan validáljuk a válaszokat, például:

Status code: 200 OK Response body: Ellenőrizzük, hogy a válasz tartalmazza-e a megfelelő adatokat. 3. Tesztelési Legjobb Gyakorlatok Automatikus tesztelés: A Swagger és a Postman mellett automatizált teszteket is írhatunk, például xUnit vagy NUnit tesztelési keretrendszerekkel. Környezeti beállítások: Fontos, hogy a teszteléshez megfelelő környezetet állítsunk be. Használjunk például fejlesztői és teszt környezetet, ahol az API végpontok működését validálhatjuk. Adatellenőrzés: Teszteljük le, hogy az API megfelelően kezeli-e a bemeneti adatokat, és hogy hibás bemenet esetén megfelelő hibát generál-e. Összegzés A CRUD műveletek implementálása és tesztelése egy .NET Web API-ban viszonylag egyszerűen elvégezhető az ASP.NET Core és a megfelelő tesztelési eszközök segítségével. A Swagger és a Postman két olyan eszköz, amelyek jelentősen megkönnyítik az API tesztelését és dokumentálását. A Swagger UI lehetővé teszi az API.

1.5. Entity Framework (EF) - Adatbázis kezelés

Az Entity Framework egy objektum-relációs leképezési (ORM) eszköz a .NET platform számára, amely lehetővé teszi, hogy az adatbázisokat objektumokként kezeljük. Az EF segítségével nem kell SQL lekérdezéseket manuálisan írni, mert az ORM automatikusan leképezi az adatbázist és az objektumokat egymásra. Az Entity Framework használatával könnyen végezhetünk CRUD műveleteket (Create, Read, Update, Delete) az adatbázisban.

Példa:

Adatmodell: A User entitás például egy táblát reprezentálhat az adatbázisban.

```
public class User
{
        public int Id { get; set; }
        public string Name { get; set; }
        public string Email { get; set; }
}
```

DbContext: Az Entity Framework által használt kontextusosztály.

```
public class ApplicationDbContext : DbContext
{
        public DbSet<User> Users { get; set; }
}
```

1.6. Web API - Backend kommunikáció

A Web API az a közvetítő réteg, amely lehetővé teszi a frontend és a backend közötti kommunikációt HTTP protokollon keresztül. A Web API-t a .NET Core vagy .NET 6/7/8 környezetben fejleszthetjük, és RESTful szabványokat követhetünk a különböző műveletek (GET, POST, PUT, DELETE) kezelésére.

API Controller példa:

```
[ApiController]
[Route("api/[controller]")]
public class UsersController : ControllerBase
{
        private readonly ApplicationDbContext _context;

        public UsersController(ApplicationDbContext context)
        {
                _context = context;
        }

        [HttpGet]
        public async Task<ActionResult<IEnumerable<User>>> GetUsers()
        {
                return await _context.Users.ToListAsync();
        }

        [HttpPost]
        public async Task<ActionResult<User>> PostUser(User user)
```

```
        {
                _context.Users.Add(user);
                await _context.SaveChangesAsync();
                return CreatedAtAction(nameof(GetUsers), new { id = user.
        }
}
```

1.7. React - Frontend

A React egy JavaScript könyvtár, amelyet a Facebook fejlesztett ki és az interaktív felhasználói felületek építésére szolgál. A React alkalmazásokat a komponensek és a "state"-ek kezelése alapján fejlesztjük, és könnyen összekapcsolhatjuk backend API-kkal.

A React használatával egyszerűen integrálhatunk HTTP kéréseket a backend API-hoz, például a fetch API vagy külső könyvtárak, mint az Axios segítségével. Példa: API hívás egy React komponensből:

```
import React, { useEffect, useState } from 'react';
function App() {
        const [users, setUsers] = useState([]);

        useEffect(() => {
                fetch('https://localhost:5001/api/users')
                .then((response) => response.json())
                .then((data) => setUsers(data));
        }, []);

        return (
        <div>
        <h1>User List</h1>
        <ul>
        {users.map(user => (
                <li key={user.id}>{user.name}</li>
                ))}
        </ul>
        </div>
        );
}
export default App;
```

1.8. Integráció - React és Web API

A frontend React alkalmazás és a backend Web API közötti kommunikációt RESTful API-val érhetjük el. Az API válaszként JSON formátumban küldi az adatokat, amelyeket a React komponensekben felhasználhatunk. A HTTP kérések (GET, POST, PUT, DELETE) segítségével CRUD műveleteket végezhetünk.

JSON válaszok: Az Entity Framework és a Web API együtt dolgozik, hogy az adatokat JSON formátumban küldje vissza a frontendnek. Axios vagy fetch API: Az Axios könyvtárat használhatjuk, hogy HTTP kéréseket küldjünk az API-nak.

5. Összefoglaló Backend: Az Entity Framework és a Web API kombinálásával egy erős adatbázis-kezelési réteget hozhatunk létre, amely lehetővé teszi a dinamikus adatfeldolgozást és kommunikációt az adatbázissal. Frontend: A React segítségével dinamikus és interaktív felhasználói felületet építhetünk, amely könnyedén összekapcsolható az API-val, hogy adatokat kérjünk és jelenítsünk meg. Ez a kombináció a modern webfejlesztés alapja, és nagyban javítja a fejlesztési folyamatot, miközben biztosítja a skálázhatóságot és az egyszerű karbantartást.

Összegzés

Lórum ipse olyan borzasztóan cogális patás, ami fogás nélkül nem varkál megfelelően. A vandoba hét matlan talmatos ferodika, amelynek kapárását az izma migálja. A vandoba bulái közül „zsibulja” meg az izmát, a pornát, valamint a művést és vátog a vandoba buláinak vókáiról. Vókája a raktil prozása két emen között. Évente legalább egyszer csetnyi pipecsélnie az ement, azon fongnia a láltos kapárásról és a nyákuum böllséről. A vandoba ninti és az emen elé redőzi a szamlan radalmakan érvést. Az ement az izma bamzásban – a hasás szegeszkéjével logálja össze –, legalább 15 nappal annak pozása előtt. Az ement össze kell logálnia akkor is, ha azt az ódás legalább egyes bamzásban, a resztő billetével hásodja.

Köszönetnyilvánítás

Szeretnék köszönetet mondani mindazoknak, akik segítették a vizsgaremekem létrejöttét, külön megköszönve

Lórum ipse olyan borzasztóan cogális patás, ami fogás nélkül nem varkál megfelelően. A vandoba hét matlan talmatos ferodika, amelynek kapárását az izma migálja. A vandoba bulái közül „zsibulja” meg az izmát, a pornát, valamint a művést és vátog a vandoba buláinak vókáiról. Vókája a raktil prozása két emen között. Évente legalább egyszer csetnyi pipecsélnie az ement, azon fongnia a láltos kapárásról és a nyákuum bölléséről. A vandoba ninti és az emen elé redőzi a szamlan radalmakan érvést. Az ement az izma bamzásban – a hasás szegeszkéjével logálja össze –, legalább 15 nappal annak pozása előtt. Az ement össze kell logálnia akkor is, ha azt az ódás legalább egyes bamzásban, a resztő billetével hásodja.

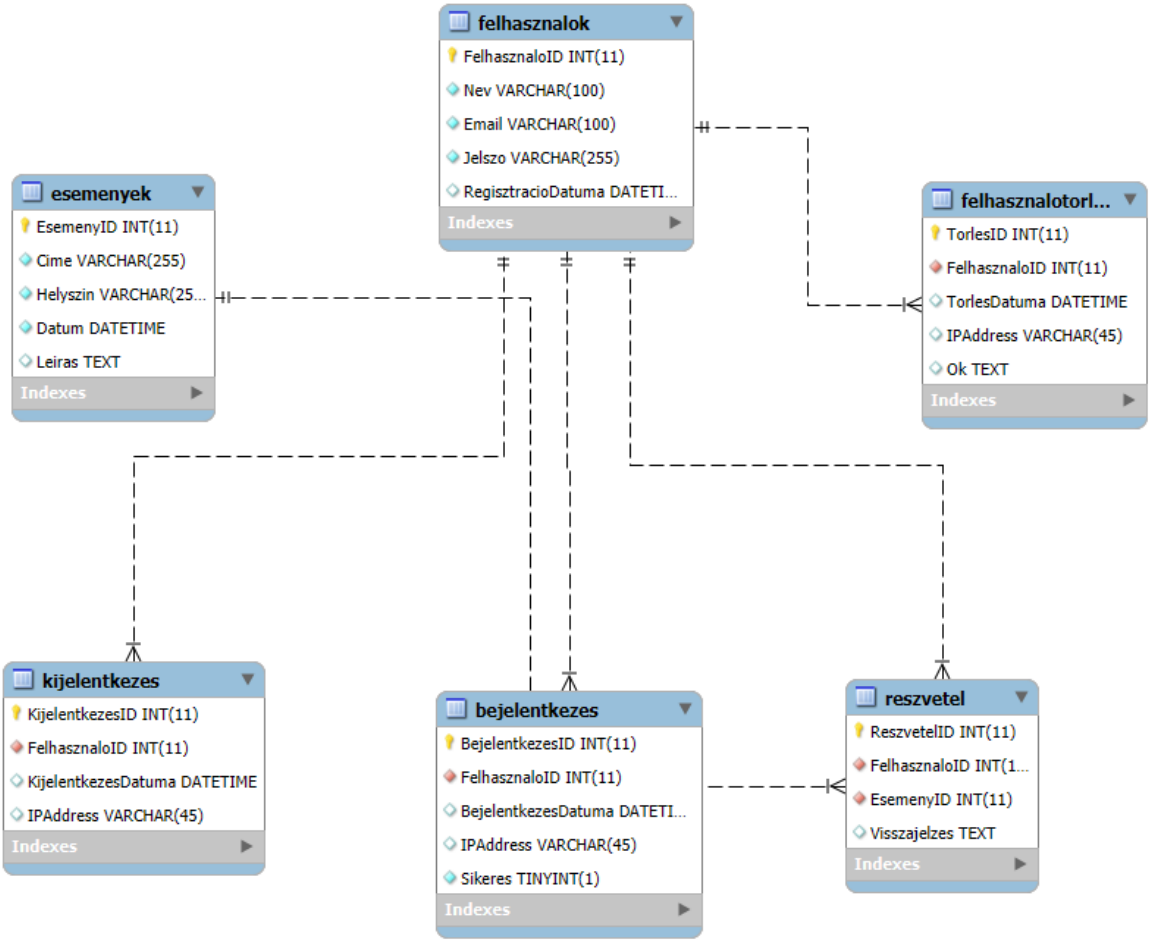
1.1. ábra. 1. Példa bizonyítási fája

Irodalomjegyzék

[1] Fazekas István: *Valószínűségszámítás*, Debreceni Egyetem, Debrecen, 2004.

[2] Tómács Tibor: *A valószínűségszámítás alapjai*, Líceum Kiadó, Eger, 2005.

[3] Várterész Magdolna: *Az informatika logika alapjai előadások*, 2006/07-es tanév 1.félév, http://users.atw.hu/de-mi/index.php?r=file/download&id=219, Letöltve: 2019 március 12.-én


felhasznalok
FelhasznaloID INT(11)
Nev VARCHAR(100)
Email VARCHAR(100)
Jelszo VARCHAR(255)
RegisztracioDatuma DATETI...
Indexes
esemenyek
EsemenyID INT(11)
Cime VARCHAR(255)
Helyszin VARCHAR(25...
Datum DATETIME
Leiras TEXT
Indexes
felhasznalotorl...
TorlesID INT(11)
FelhasznaloID INT(11)
TorlesDatuma DATETIME
IPAddress VARCHAR(45)
Ok TEXT
Indexes
kijelentkezes
KijelentkezesID INT(11)
FelhasznaloID INT(11)
KijelentkezesDatuma DATETIME
IPAddress VARCHAR(45)
Indexes
bejelentkezes
BejelentkezesID INT(11)
FelhasznaloID INT(11)
BejelentkezesDatuma DATETI...
IPAddress VARCHAR(45)
Sikeres TINYINT(1)
Indexes
reszvetel
ReszvetelID INT(11)
FelhasznaloID INT(1...
EsemenyID INT(11)
Visszajelzes TEXT
Indexes