Eseményvezérelt alkalmazások: 11. gyakorlat

A feladat egy *Todo List* (“teendő lista”) alkalmazás elkészítése, relációs adatbázis alapú perzisztenciával.

Az alkalmazás alkalmas új listák (pl. bevásárló lista, teendő lista) felvételére, azokban pedig elemek rögzítésére. Minden elemhez kötelezően megadandó neve és határideje, illetve egy opcionális, hosszabb leírása. A program lehetőséget nyújt a listák és a bennük található elemek hozzáadására, módosítására és törlésére.

Az alkalmazást MVVM architektúrában valósítjuk meg. A modell réteg lesz felelős a perzisztenciáért, az adatokat a program objektum-relációs leképezéssel (ORM), az *Entity Framework Core* keretrendszer használatával, relációs adatbázisban tárolja.

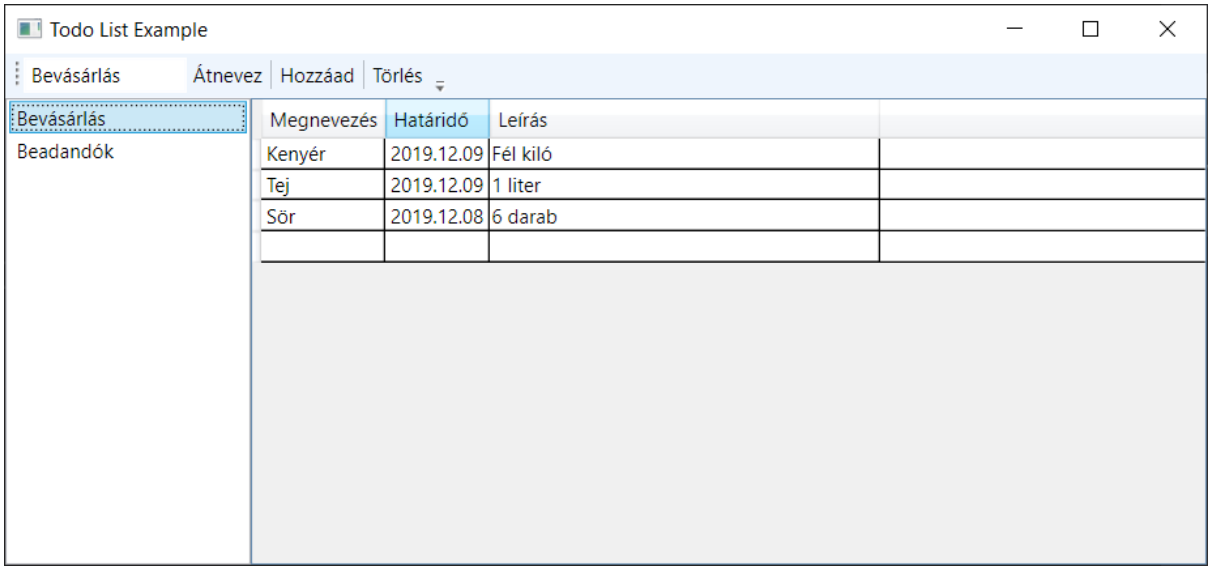

Figure 1: Todo List alkalmazás

A programot a munkafüzet *bottom-up* módszerrel az alsóbb rétegek felől felfele építkezve állítja össze.

1 Projekt létrehozása

Készítsünk egy új WPF projektet .NET Core 3.1 keretrendszer használatával. Adjuk hozzá a projektünkhöz a következő NuGet csomagokat: - `Microsoft.EntityFrameworkCore`, az Entity Framework Core használatához. - `Microsoft.EntityFrameworkCore.SqlServer`, az MSSQL adatbázismotor használatához az Entity Framework Core keretrendszeren keresztül. - `Microsoft.EntityFrameworkCore.Proxies`, a navigációs tulajdonságok lusta kiértékelésének támogatásához.

2 Modell réteg

A modell réteg lesz felelős az adatbáziskezelésért. Mivel az adatok elemi kezelésén kívül (*CRUD* = *create* / *read* / *update* / *delete*) egyéb, absztraktabb funkcionalitásra nem lesz szükségünk, ezért a modell és a

perzisztencia réteg szétválasztása most nem indokolt.

Hozzuk létre a `List` és az `Items` entitás osztályokat a séma szerint. Származtassunk egy `TodoListDbContext` adtabázis kontextus osztályt a `DbContext` osztályból, amely 2 táblát fog tartalmazni, az alábbiak szerint.

```
public class TodoListDbContext : DbContext
{
    public TodoListDbContext(DbContextOptions<TodoListDbContext> options)
        : base(options) { }

    public DbSet<List> Lists { get; set; }

    public DbSet<Item> Items { get; set; }
}
```

Az adatbázis sémáját az alábbi diagram szemlélteti:

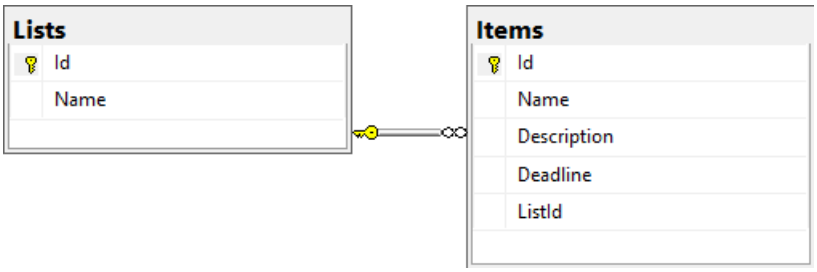

Figure 2: Adatbázis kapcsolat diagram

Definiáljuk a `List` és az `Item` entitás osztályokat a diagramnak megfelelően. Ne feledkezzünk meg az idegen kulcsokat leképező *navigációs propertyk* megadásáról sem.

2.1 Adatbázis inicializálása

Annak érdekében, hogy kezdetben ne legyen üres az adatbázisunk, hanem egy kiindulási, tesztelési célú adathalmazzal rendelkezzünk, hozzuk létre a statikus `DbInitializer` osztályt és annak statikus `async Task Initialize(TodoListDbContext context)` metódusát. Ennek feladata lesz az adtabázis létrehozása, amennyiben még nem létezik:

```
await context.Database.EnsureCreatedAsync();
```

Továbbá néhány példa adat betöltése az adatbázisba:

```
IList<List> defaultLists = new List<List>
{
    new List
    {
        Name = "Bevásárlás",
        Items = new List<Item>()
        {
            new Item()
            {
                Name = "Kenyér",
                Deadline = DateTime.Now.AddDays(1),
                Description = "Fél kiló"
            },
            /* ... */
        }
    },
    /* ...*/
```

```
};

foreach (List list in defaultLists)
    context.Lists.Add(list);

await context.SaveChangesAsync();
```

2.2 Adatbázis kapcsolat beállítása

Adjunk hozzá a projektünkhöz egy konfigurációs állományt `App.config` néven és helyezzük el benne a relációs adatbázis eléréséhez szükséges *connection stringet*:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="TodoListModel"
         connectionString="Data Source=(LocalDb)\MSSQLLocalDB;
                           Initial Catalog=TodoList;
                           Integrated Security=True;
                           MultipleActiveResultSets=True" />
  </connectionStrings>
</configuration>
```

Az `App.xaml.cs` fájlban az alkalmazás `Startup` eseményére iratkozzunk fel egy eseménykezelővel (pl. `App_Startup()`) és konstruáljunk egy új `TodoListDbContext` objektumot. Az adatbázis kontextus objektum beállításait a `DbContextOptionsBuilder` segédosztály segítségével állíthatjuk össze:

- a `UseSqlServer()` metódus meghívásával konfigurálhatjuk az MSSQL szerver használatát, egyben a *connection stringet* is az alkalmazás konfigurációs állományból betöltve;
- a `UseLazyLoadingProxies()` metódus meghívásával engedélyezve a navigációs tulajdonságok lusta kiértékelését.

Végül inicializáljuk az adtabázist a korábban elkészített `DbInitializer.Initialize()` statikus metódus meghívásával.

```
private async void App_Startup(object sender, StartupEventArgs e)
{
    DbContextOptionsBuilder<TodoListDbContext> optionsBuilder =
        new DbContextOptionsBuilder<TodoListDbContext>()
            .UseSqlServer(ConfigurationManager.ConnectionStrings["TodoListModel"]
                .ConnectionString)
            .UseLazyLoadingProxies();

    model = new TodoListDbContext(optionsBuilder.Options);
    await DbInitializer.Initialize(model);
}
```

3 Nézetmodell réteg

A nézetmodell rétegben ugyan felhasználhatnánk a modell réteg entitás típusait (`List` és `Item`), azonban ezzel kettős problémát vezetnénk be:

- Egyrészt a modell réteg típusait (és objektumait) közvetlenül használhatná fel majd a nézet réteg, megsértve ezzel az MVVM architektúra szigorú szabályát, amely szerint minden réteg csak a közvetlenül alatta lévőt ismerheti.
- Másrészt amennyiben később módosítanánk a modell entitás típusait, a módosítást mindenképpen végig kellene vezetni egészen a nézet rétegig, még akkor is, ha a változás ezt nem indokolná.

Ezért a `List` és `Item` entitás típusokhoz egy-egy csomagoló (*wrapper*) osztályt készítünk `ListViewModel` és `ItemViewModel` néven. A `ViewModelBase` osztály az előadás példákban is szereplővel megegyező.

```
public class ListViewModel : ViewModelBase
{
    private List list;

    public List Entity
    {
        get { return list; }
    }

    [Required]
    [StringLength(50)]
    public String Name
    {
        get { return list.Name; }
        set { list.Name = value; OnPropertyChanged(); }
    }

    public ObservableCollection<ItemViewModel> Items
    {
        get
        {
            return new ObservableCollection<ItemViewModel>(
                list.Items.Select(item => new ItemViewModel(item))
            );
        }
    }

    public ListViewModel(List list)
    {
        this.list = list;
    }
}
```

Hasonló módon készítsük el a `ItemViewModel` osztályt is.

Az implementáció így elsőre redundánsnak tűnhet, de más módon a rétegek elvárt szeparációja nem érhető el. Nagyobb alkalmazásokban egy entitás típus és a nézetmodell típusa jobban eltérhet egymástól, a kódredundancia pedig kódgenerátorok segítségével mérsékelhető.

Tipp: például az AutoMapper NuGet csomag használatával automatizálható az EF entitás típusok és a csomagoló nézetmodell osztályok közötti megfeleltetés. A Fody NuGet csomag alkalmazásával pedig közel teljesen elkerülhető a `PropertyChanged` esemény manuális kiváltása.

Az alkalmazás nézetmodelljét a `TodoListViewModel` osztály fogja adni, amelyet szintén a `ViewModelBase` absztrakt osztályból származtassunk. Az osztály aggregálja a következő adatokat:

- `context`: az adatbázis kontextus egy példánya (amelyet konstruktoron keresztüli függőség befecskendezéssel kap meg);
- `Lists`: a tárolt teendő listák, `ObservableCollection<ListViewModel>` típusú;
- `Items`: az aktuálisan kiválasztott lista elemei, `ObservableCollection<ItemViewModel>` típusú;
- `currentList`, `CurrentListName`: az aktuálisan kiválasztott lista, valamint neve. Utóbbi az alkalmazás felületén szerkeszthető lesz. Típusuk ennek megfelelően `ListViewModel` és `string` lesz.

3.1 Teendő listák változáskezelése

A nézetmodellben rögzítsünk 4 parancsot is:

- `SelectCommand`: a kijelölt teendő lista megváltoztatására lefutó parancs. Módosítsa az `Items`, a `currentList`, `CurrentListName` tagok értékét.
- `RenameListCommand`: a kijelölt teendő lista (`currentList`) nevét a felületen megadottra (`CurrentListName`) módosítja. A módosításokat az adatbázisba is perzisztálja.

- `NewListCommand`: a felületen megadott névvel (`CurrentListName`) felvesz egy új listát, menti, majd az adatokat az adtabázisból újra betölti.
- `DeleteListCommand`: az aktuálisan kiválasztott (`currentList`) listát törli, az állapotváltozást menti, majd az adatokat az adtabázisból újra betölti.

A parancsok legyenek a `DelegateCommand` osztály példányai, amely az előadás példákban is szereplővel megegyező.

3.2 Teendő elemek változáskezelése

A teendő elemek változásait (hozzáadás, módosítás, törlés) az őket tároló `Items` gyűjtemény változásának detektálásával valósítjuk meg. Az `ObservableCollection<T>` típus rendelkezik egy `CollectionChanged` eseménnyel, amellyel a gyűjtemény változása (elem hozzáadása, törlése, stb.) kezelhető, azonban a tárolt objektumok belső állapotának változása nem kerül kiváltásra. Szükséges ezért az `Items`-ben tárolt `ItemViewModel` típusú objektumok `PropertyChanged` eseményére is feliratkoznunk.

A megjelenítendő teendő elemek a `SelectCommand` hatására így a következő módon frissíthetőek:

```
// korábbi eseménykezelés eltávolítása
Items.CollectionChanged -= OnItemsChanged;
// gyűjtemény kiürítése
Items.Clear();
// új elemek felvétele
if (list != null)
{
    foreach (var item in list.Items)
    {
        Items.Add(item);
        // ha bármely teendő elem változik, szinkronizáljunk az adatbázissal
        item.PropertyChanged += (o, args) => { context.SaveChanges(); };
    }
}
// teendő elem gyűjteményének változáskezelése
Items.CollectionChanged += OnItemsChanged;

// az aktuális lista frissítése
currentList = list;
CurrentListName = currentList != null ? currentList.Name : String.Empty;
```

A teljes gyűjtemény változáskezelése:

```
private void OnItemsChanged(object sender, NotifyCollectionChangedEventArgs e)
{
    // amennyiben vannak új teendő elemek
    if(e.NewItems != null)
        foreach (ItemViewModel item in e.NewItems)
        {
            // az új elemet az aktuálisan kiválasztott listába kell felvenni
            currentList.Entity.Items.Add(item.Entity);
            // ha a teendő elem később változik, szinkronizáljunk az adatbázissal
            item.PropertyChanged += (o, args) => { context.SaveChanges(); };
        }

    // amennyiben vannak törlendő teendő elemek
    if(e.OldItems != null)
        foreach (ItemViewModel item in e.OldItems)
        {
            // a törlendő elemet az adatbázis kontextus felé törlendőnek jelöljük
            context.Items.Remove(item.Entity);
        }
```

```
    // adatbázis szinkronizáció
    context.SaveChanges();
}
```

4 Nézet réteg

Az alkalmazás egyetlen ablakból áll, nézetét tisztán deklaratív módon, XAML kóddal valósítjuk meg.

A felületen egy `DockPanel`-be vegyünk fel egy `ToolBarTray`, egy `ListBox` és egy `DataGrid` WPF vezérlőt, ezeket `Dock` tulajdonságát állítsuk rendre felülre, balra és jobbra igazításra.

4.1 Eszköztár

A `ToolBarTray`-re vegyünk fel egy szövegdobozt (`TextBox`), valamint három gombot (`Button`). Végezzük el az adat- illetve parancskötéseket a nézetmodell `CurrentListName`, a `RenameListCommand`, a `NewListCommand` és `DeleteListCommand` tagjaira.

4.2 Teendő listák megjelenítése

A listákat egy `ListBox` vezérlő jelenítse meg. Sablon (*template*) alkalmazásával megadhatjuk hogyan kell az egyes listákat megjeleníteni:

```
<ListBox x:Name="lists" ItemsSource="{Binding Lists}" ...>
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding Name}" />
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

A `ListBox` vezérlő kiválasztott elemének változásáról annak `SelectionChanged` eseménye révén értesülhetünk. Az eseményekre azonban nem tudunk közvetlenül parancskötést végrehajtani (és így az MVVM architektúrát megfelelően alkalmazni), ezért projektünkhöz adjuk hozzá a *Microsoft.Xaml.Behaviors.Wpf* NuGet csomagot.

Ezt követően definiáljuk az új névteret az ablak (`Window`) elemében:

```
<Window xmlns:i="http://schemas.microsoft.com/xaml/behaviors" ...>
```

És máris az adott vezérlő bármely eseményére végezhetünk parancskötést egy `EventTrigger` segítségével:

```
<ListBox x:Name="lists" ItemsSource="{Binding Lists}" ...>
    <i:Interaction.Triggers>
        <i:EventTrigger EventName="SelectionChanged">
            <i:InvokeCommandAction Command="{Binding SelectCommand}"
                CommandParameter="{Binding ElementName=lists, Path=SelectedItem}" />
        </i:EventTrigger>
    </i:Interaction.Triggers>
</ListBox>
```

4.3 Teendő elemek megjelenítése

A teendő elemeket egy adatrácsban (`DataGrid`) jelenítjük meg. Vegyük fel a megfelelő oszlopokat, ezzel egyidejűleg az automatikus oszlop generálást (`AutoGenerateColumns`) tiltsuk meg. Engedélyezzük a felhasználói hozzáadást (`CanUserAddRows`), illetve törlést (`CanUserDeleteRows`).

```
<DataGrid ItemsSource="{Binding Items}" AutoGenerateColumns="False"
          CanUserAddRows="True" CanUserDeleteRows="True">
    <DataGrid.Columns>
        <DataGridTextColumn Header="Megnevezés" Binding="{Binding Name}" />
```

```
        <DataGridTextColumn Header="Határidő"
                            Binding="{Binding Deadline, StringFormat=\{0:yyyy.MM.dd\}}" />
        <DataGridTextColumn Header="Leírás" Binding="{Binding Description}" />
    </DataGrid.Columns>
</DataGrid>
```

Tipp: az alkalmazás futtatáskor törölni a sor kiválasztásával, majd a *Delete* gombbal lehet.

Új elem létrehozásához (amely `ItemViewModel` típusú lesz) a típusnak rendelkeznie kell alapértelmezett (nulla paraméteres) konstruktorral. Ezt megadhatjuk például az alábbi módon:

```
public ItemViewModel()
{
    item = new Item()
    {
        Name = "Új elem",
        Deadline = DateTime.Now.AddDays(1)
    };
}
```

5 Alkalmazás környezet

Az `App.xaml.cs` fájlban az adatbázis inicializálását követően hozzuk létre a modellt, a nézetmodell és a nézetet, majd kapcsoljuk össze a függőségeket és jelenstsük meg az ablakot.