Bazy danych SQLite

Bartosz Miąskowski

Czym jest SQLite?

- Lekka baza danych działająca bez serwera – bez DBMS.
- Wszystkie dane są przekazywane w jednym pliku `.db` / `.db3`.

Dlaczego w ten sposób?

- Jeżeli chcemy, aby aplikacja przechowywała ustrukturyzowane dane, musimy skorzystać z bazy danych.
- Nie wolno łączyć się bezpośrednio z zdalną bazą danych z aplikacji klienckiej (*jest to niebezpieczne i drogie*).
- Chcemy aby aplikacja umożliwiała nam dostęp do danych, nawet w trybie offline.

Ograniczenia SQLite

Jako, że SQLite nie jest pełnoprawnym DBMS posiada pewne ograniczenia (*nie wynikające z platformy .NET*).

- Obsługa operacji SQL jest ograniczona – zbliżone do standardu SQL-92.
- Nie obsługuje schematów, autogenerowanych tokenów współbieżności.
- Ograniczone wsparcie dla typów danych (`decimal`, `DateTimeOffset`, `TimeSpan`, `ulong` itd.).

Co ciekawe, SQLite obsługuje między innymi:

- zapytania zagnieżdżone,
- widoki,
- transakcje,
- wyzwalacze (*tylko częściowo*),
- definiowanie własnych funkcji.

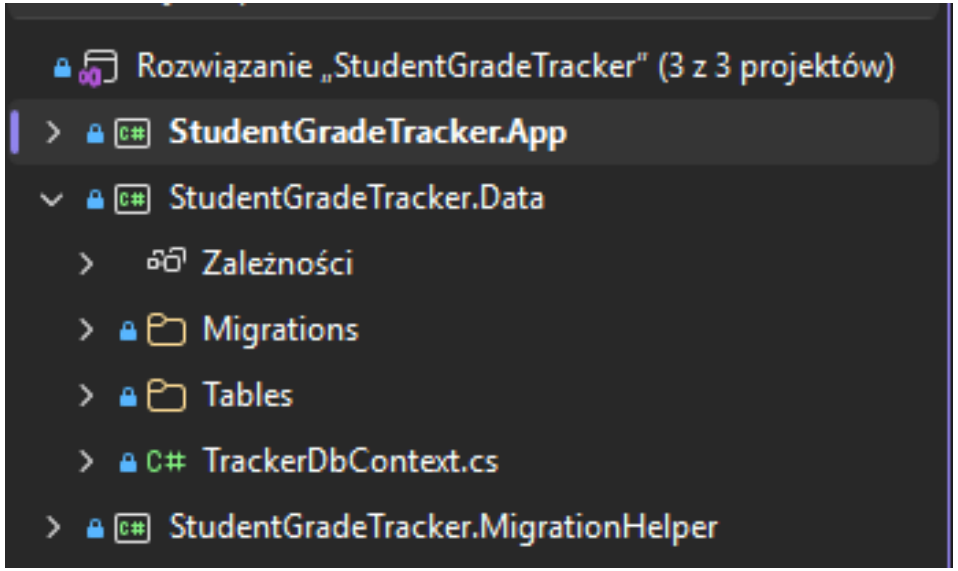
Architektura warstwy danych

- W aplikacjach .NET zgodnie z konwencją, logikę związaną z bazami danych umieszcza się w katalogu `Data` – tam znajduje się zwykle kontekst bazy danych oraz jej tabele (w katalogu `Models` lub `Tables`).

```
Data
├── ApplicationDbContext.cs   <-- Kontekst
└── Models                    <-- Tabele
    ├── AgentToken.cs
    ├── ApplicationUser.cs
    ├── FileBinding.cs
    ├── Invitation.cs
    ├── LoginAttempt.cs
    ├── NotificationSubscription.cs
    ├── RequestChange.cs
    ├── Request.cs
    ├── Specialization.cs
    ├── TrustedBrowser.cs
    └── UploadedFile.cs
```

Architektura warstwy danych

- Tworząc aplikację korzystającą z lokalnej bazy danych w MAUI, będą potrzebne trzy projekty:
  - projekt aplikacji – projekt aplikacji MAUI,
  - projekt z bazą danych – biblioteka klas, zawierająca logikę bazy danych,
  - projekt pomocniczy do migracji – aplikacja konsolowa, służąca tylko jako punkt startowy do migracji.



Architektura rozwiązania Visual Studio

Jeżeli trzymasz **bazę danych w osobnym projekcie**, kontekst może znajdować się na wierzchu.

Entity Framework Core

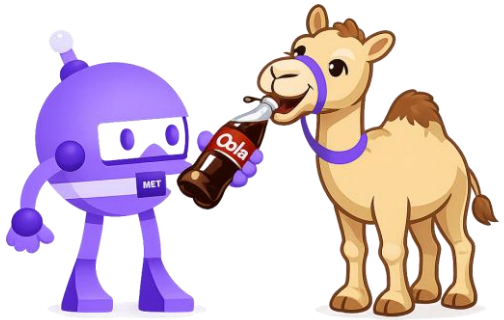
- **Przypomnienie:**
  - Entity Framework (*EF*) to otwartoźródłowa technologia mapowania obiektowo-relacyjnego (*ORM*) stworzona przez Microsoft dla platformy .NET.
  - Pozwala programistom pracować z relacyjnymi bazami danych przy użyciu obiektów C#, eliminując konieczność pisania większości zapytań SQL.
  - EF automatyzuje operacje CRUD (*tworzenie, odczyt, aktualizacja, usuwanie*) i obsługuje relacje w bazie.

- Podsumowując, EF Core to potężna biblioteka do obsługi baz danych – zamiast na zapytaniach SQL, pracujemy na obiektach C# oraz zapytaniach LINQ.

Entity Framework Core – obsługa SQLite

- Do podstawowej obsługi SQLite przez Entity Framework Core potrzebne będę następujące biblioteki:
  - `Microsoft.EntityFrameworkCore` – podstawowa biblioteka, zawierająca Entity Framework,
  - `Microsoft.EntityFrameworkCore.Sqlite` – sterownik do SQLite dla Entity Framework,
  - `Microsoft.EntityFrameworkCore.Tools` – narzędzia (*m.in. do migracji*) baz danych.

- Powyższe biblioteki instalujemy poprzez menedżer pakietów NuGet do projektu z bazą danych (`.Data`).
- Istnieje znacznie więcej bibliotek rozszerzających możliwości Entity Framework Core (*Design, Proxies itd.*), jednakże i tak nie będą one przydatne w pracy z SQLite.



Kontekst bazy danych – `DbContext`

DbContext w Entity Framework Core to najważniejsza klasa odpowiedzialna za komunikację aplikacji z bazą danych. Można powiedzieć, że jest to most między kodem C# a tabelami w bazie.

Co robi kontekst?

- łączy się z bazą danych,
- udostępnia tabele jako właściwości `DbSet<T>`,
- udostępnia dostęp do procedur oraz widoków zmaterializowanych,
- śledzi zmiany w obiektach (*dodanie, edycja, usunięcie*),
- zapisuje zmiany do bazy przez `SaveChanges()`,
- konfiguruje relacje i strukturę modeli.

Kontekst bazy danych – DbContext

```
public class TrackerDbContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        base.OnConfiguring(optionsBuilder);

        var path = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData), "trackerDb.db3");
        optionsBuilder.UseSqlite($"Data Source={path}");
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        // Każda ocena ma swój przedmiot, ale przedmiot może mieć wiele ocen
        modelBuilder.Entity<Grade>().HasOne(o => o.Subject).WithMany(m => m.Grades)
            .HasForeignKey(k => k.SubjectId).OnDelete(DeleteBehavior.Cascade);
    }

    public DbSet<Subject> Subjects { get; set; }

    public DbSet<Grade> Grades { get; set; }
}
```

Kontekst bazy danych - SQLite

Tworzenie tabel

W Entity Framework Core tabela w bazie danych jest tworzona na podstawie klasy C#. Każda właściwość klasy staje się kolumną w tabeli.

Najważniejsze atrybuty:

- `[Key]` – oznacza klucz główny (*primary key*),
- `[Required]` – pole wymagane (*NOT NULL*),
- `[MaxLength(x)]` – maksymalna długość tekstu,
- `[Column("Nazwa")]` – własna nazwa kolumny,
- `[NotMapped]` – pole nie będzie zapisane w bazie,
- `[DatabaseGenerated]` – sposób generowania wartości (*np. Identity*)

```
public class Student
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    [Required]
    [MaxLength(100)]
    public string FirstName { get; set; }

    [MaxLength(100)]
    public string LastName { get; set; }

    public DateTime BirthDate { get; set; }
}
```

Przykładowa tabela EF Core

Uwaga!
Jeżeli właściwość nazywa się **`Id`** lub **`StudentId`** – nie trzeba dodawać atrybutów `[Key]` oraz `[DatabaseGenerated]` – **EF Core robi to automatycznie**.

Dodawanie tabeli do bazy

Aby Entity Framework Core obsługiwał nową tabelę, model musi zostać dodany do klasy `DbContext` jako właściwość typu `DbSet<T>`.

Co oznacza `DbSet<T>`?

- reprezentuje tabelę w bazie danych,
- umożliwia wykonywanie operacji CRUD,
- pozwala pobierać dane za pomocą LINQ,
- informuje EF Core, że dana klasa ma być tabelą.

```
public class SchoolDbContext : DbContext
{
    public DbSet<Student> Students { get; set; }
    public DbSet<Grade> Grades { get; set; }
}
```

Prosty przykład kontekstu z dwiema tabelami

Konfiguracja połączenia z bazą danych

Tworząc kontekst bazy danych pod SQLite, połączenie z bazą danych (*connection string*) będzie znajdowało się najczęściej w kontekście bazy danych – w metodzie `OnConfiguring`.

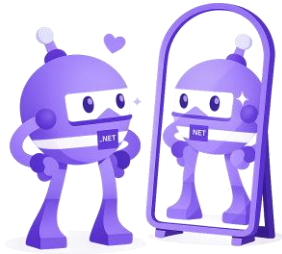
```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    base.OnConfiguring(optionsBuilder);

    var path = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData), "trackerDb.db3");
    optionsBuilder.UseSqlite($"Data Source={path}");
}
```

Konfiguracja połączenia z bazą danych

Uwaga!
Inaczej będzie wyglądała ścieżka do pliku z bazą SQLite na Windowsie oraz Androidzie. W przypadku **Windowsa**, dobrze będzie **do ścieżki dopisać katalog przeznaczony do przechowywania plików** – np. „*MyApp*”.

W przeciwnym wypadku, pliki będą zapisywane bezpośrednio w **`%AppData%\Local`** – więc bardzo szybko zrobi się śmietnik.

Connection string



- Connection string to ciąg tekstowy zawierający informacje potrzebne aplikacji do połączenia z bazą danych. Zawiera takie informacje jak: adres serwera, nazwę bazy danych, login, hasło, dodatkowe parametry połączenia.
- Connection string będzie się różnił, w zależności od DBMS.

```
User ID=root;Password=myPassword;Host=localhost;Port=5432;Database=myDataBase;Pooling=true;
Min Pool Size=0;Max Pool Size=100;Connection Lifetime=0;
```

PostgreSQL

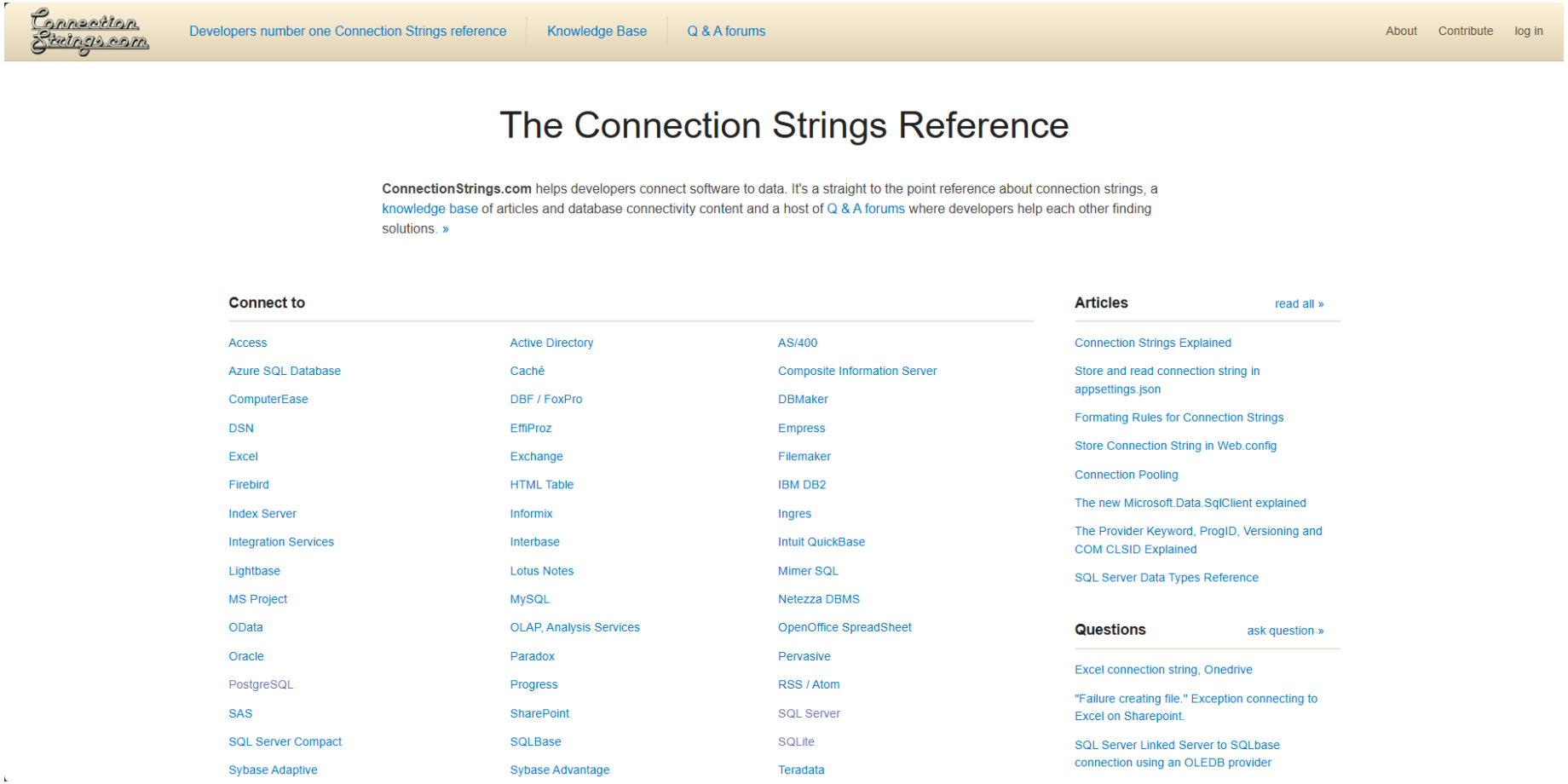
```
Data Source=c:\mydb.db;Version=3;
```

SQLite

```
Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;
```

Microsoft SQL Server

Connection string



Strona www.connectionstrings.com – baza connection stringów do różnych DBMS.

Connection string

SQLite.NET

Basic

```
Data Source=c:\mydb.db;Version=3;
```

Version 2 is not supported by this class library.

SQLite

In-Memory Database

An SQLite database is normally stored on disk but the database can also be stored in memory. Read more about SQLite in-memory databases here.

```
Data Source=:memory:;Version=3;New=True;
```

SQLite

Using UTF16

```
Data Source=c:\mydb.db;Version=3;UseUTF16Encoding=True;
```

SQLite

With password

```
Data Source=c:\mydb.db;Version=3;Password=myPassword;
```

SQLite

Using the pre 3.3x database format

```
Data Source=c:\mydb.db;Version=3;Legacy Format=True;
```

SQLite

With connection pooling

Connection pooling is not enabled by default. Use the following parameters to control the connection pooling mechanism.

```
Data Source=c:\mydb.db;Version=3;Pooling=True;Max Pool Size=100;
```

Przykładowe connection strings dla SQLite

Migracje

- Migracje w Entity Framework Core służą do aktualizowania struktury bazy danych na podstawie zmian w klasach modeli.
- EF Core porównuje aktualny kod z poprzednim jego stanem – na tej podstawie generuje pliki migracji, które pozwalają na zmodyfikowanie struktury bazy danych.
- Za pomocą migracji można między innymi:
  - tworzyć nowe tabele,
  - dodawać nowe kolumny,
  - usuwanie i dodawanie wartości w tabelach.
- Migracje celowo utrudniają takie procesy jak zmiana typu kolumny – wtedy trzeba dodać własny kod SQL do migracji.

Migracje bazy danych w .NET MAUI

- Migracje najczęściej przeprowadza się przez konsolę managera pakietów NuGet, ten sposób jednak nie działa za dobrze w współpracy z MAUI – ominiemy więc tą część.
- Alternatywna ścieżka, to skorzystanie z narzędzia dotnet-ef.

Parametry:

- `migrations add AddedGradeName` – tworzy nową migrację o nazwie `AddedGradeName`,
- `--project` – wskazuje projekt, w którym znajduje się `DbContext` i gdzie ma zostać zapisana migracja,
- `--startup-project` – wskazuje projekt uruchamiany podczas tworzenia migracji,
- `--context` – określa, którego `DbContext` użyć.

```
dotnet ef migrations add AddedGradeName --project .\StudentGradeTracker.Data
--startup-project .\StudentGradeTracker.MigrationHelper --context TrackerDbContext
```

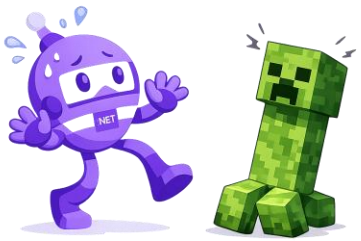
Dodanie migracji z użyciem polecenia dotnet-ef *(to jest jedna komenda)*

Relacje

- Relacje określają powiązania między tabelami w bazie danych. Dzięki nim dane mogą być logicznie połączone. W Entity Framework Core relacje tworzymy za pomocą kluczy obcych oraz właściwości nawigacyjnych.
- Relacja najczęściej będzie polegała na dodaniu odpowiednich właściwości w tabelach (*lub tylko jednej*) oraz dodanie kolumny z kluczem obcym do innej tabeli.
- Właściwości o typie innej tabeli to właściwości nawigacyjne.
- Relacje opisuje się w metodzie `OnModelCreating`, w kontekście bazy danych.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    // Każda ocena ma swój przedmiot, ale przedmiot może mieć wiele ocen
    modelBuilder.Entity<Grade>().HasOne(o => o.Subject).WithMany(m => m.Grades)
        .HasForeignKey(k => k.SubjectId).OnDelete(DeleteBehavior.Cascade);
}
```

Przykładowa konfiguracja relacji 1:N – każda ocena, ma swój przedmiot.



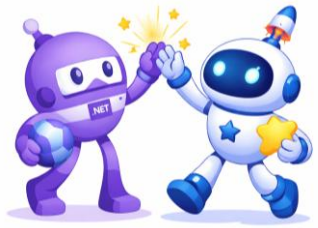
Relacje – 1:1

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

    public Profile Profile { get; set; }
}

public class Profile
{
    public int Id { get; set; }
    public string FullName { get; set; }

    public int UserId { get; set; }
    public User User { get; set; }
}
```

Tabele

```
modelBuilder.Entity<User>()
    .HasOne(u => u.Profile)
    .WithOne(p => p.User)
    .HasForeignKey<Profile>(p => p.UserId);
```

Konfiguracja



Relacje – 1:N

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

    public List<Grade> Grades { get; set; } = new();
}

public class Grade
{
    public int Id { get; set; }
    public int Value { get; set; }

    public int StudentId { get; set; }
    public Student Student { get; set; }
}
```

Tabele

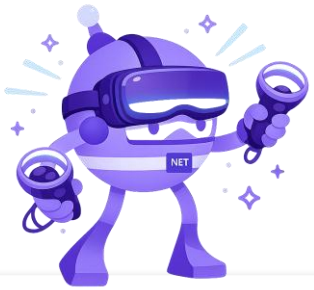
```
modelBuilder.Entity<Grade>()
    .HasOne(g => g.Student)
    .WithMany(s => s.Grades)
    .HasForeignKey(g => g.StudentId);
```

Konfiguracja

Relacje – N:N

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

    public ICollection<Course> Courses { get; set; } = new List<Course>();
}

public class Course
{
    public int Id { get; set; }
    public string Title { get; set; }

    public ICollection<Student> Students { get; set; } = new List<Student>();
}
```

Tabele



```
modelBuilder.Entity<Student>()
    .HasMany(s => s.Courses)
    .WithMany(c => c.Students);
```

Konfiguracja

Reakcja na usuwanie rekordów

- `DeleteBehavior` określa, co stanie się z rekordami powiązanymi po usunięciu rekordu głównego.

Dostępne wartości:

- `Cascade` – usuwa dane powiązane,
- `Restrict` – blokuje usunięcie,
- `SetNull` – ustawia NULL w kluczu obcym.

```
modelBuilder.Entity<Grade>()
    .HasOne(g => g.Student)
    .WithMany(s => s.Grades)
    .HasForeignKey(g => g.StudentId)
    .OnDelete(DeleteBehavior.Cascade);
```

Konfiguracja relacji wraz z usuwaniem kaskadowym

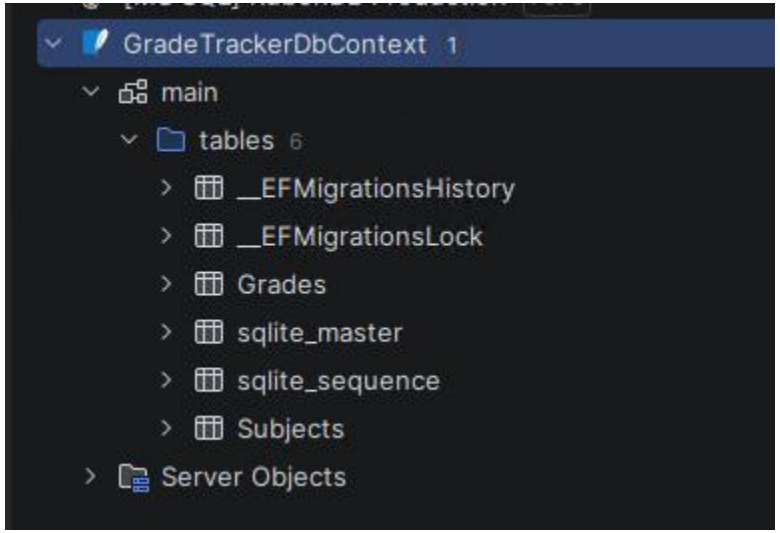
Rejestracja kontekstu bazy danych

- Kontekst bazy danych rejestrujemy w DI – tak, aby można było go wstrzyknąć bezpośrednio do ViewModeli lub innych serwisów.

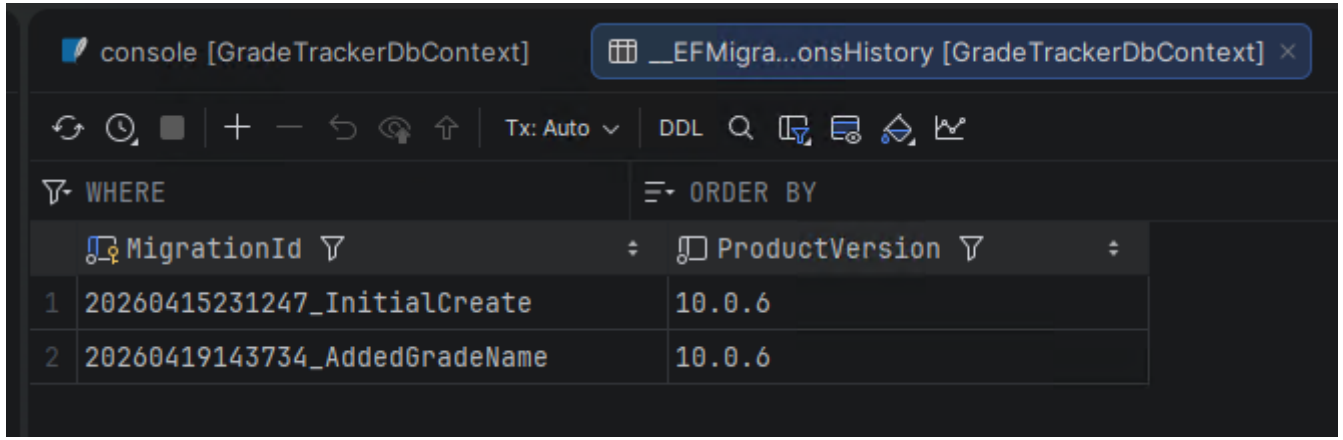
```
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder.UseMauiApp<App>().ConfigureFonts(fonts =>
        {
            fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
            fonts.AddFont("OpenSans-Semibold.ttf", "OpenSansSemibold");
        }).UseMauiCommunityToolkit();

        // Rejestracja TrackerDbContext jako singletona, aby był współdzielony w całej aplikacji
        builder.Services.AddSingleton<TrackerDbContext>();

        // Tutaj zmiana - zamiast zwracać zbudowaną aplikację, przypisujemy ją do zmiennej
        var app =  builder.Build();

        // Stosowanie migracji bazy danych
        using (var scope = app.Services.CreateScope())
        {
            var db = scope.ServiceProvider.GetRequiredService<TrackerDbContext>();
            db.Database.Migrate();
        }

        return app;
    }
}
```

`MauiProgram.cs` z zarejestrowanym kontekstem bazy oraz automatyczną migracją na starcie

Struktura bazy danych



Przykładowa struktura surowej bazy danych



Zawartość tabeli `__EFMigrationsHistory`

Operacje na danych – dodawanie rekordu

```
var student = new Student
{
    Name = "Jan Kowalski"
};

_db.Students.Add(student);
await context.SaveChangesAsync();
```

Dodawanie obiektu do tabeli Students

Operacje na danych – dodawanie rekordu

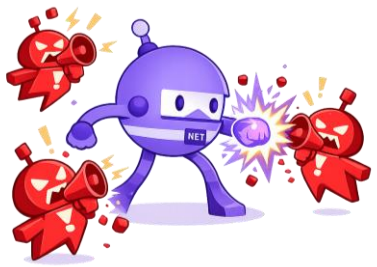
```
var student = await
context.Students.FindAsync(1);

context.Students.Remove(student);
await context.SaveChangesAsync();
```

Usuwanie obiektu z tabeli Students

Operacje na danych – aktualizacja rekordu

```
var student = await
context.Students.FindAsync(1);

student.Name = "Adam Nowak";

await context.SaveChangesAsync();
```

Aktualizacja rekordu w tabeli Students



Na co należy zwracać uwagę?

- Do aktualizacji rekordów w bazie, musimy użyć obiektu, który został z niej pobrany (*tracking EF*).
- Nie można odwoływać się do jednej instancji kontekstu bazy danych z wielu wątków na raz, ani współbieżnie – jeżeli jest rozpoczęta transakcja, musi najpierw się zakończyć.
- Migracje są klasami – w związku z tym ich nazwy piszemy jako PascalCase.
- Za zastosowanie migracji odpowiedzialna jest aplikacja, korzystająca z bazy danych – w MAUI najlepiej jest to robić na starcie programu.