Uniwersytet Bielsko-Bialski

LABORATORIUM

Programowanie dla Internetu w technologii ASP.NET

Sprawozdanie nr 3

Warstwa repozytorium

GRUPA: 2B / SEMESTR: 6 / ROK: 3

Igor Gawłowicz / 59096

Warstwa Repozytorium

wzorzec repozytorium (ang. Repository Pattern) jest popularnym wzorcem projektowym stosowanym w aplikacjach ASP.NET i innych aplikacjach opartych na architekturze MVC (Model-View-Controller). Pozwala on na separację logiki biznesowej od kodu dostępu do danych poprzez dodanie warstwy pośredniczącej między aplikacją a bazą danych.

Aby zaimplementować go w naszej aplikacji musimy zacząć od utworzenia interfejsów dla każdej z tabel w bazie

```
using System.Collections.Generic;
using System.Threading.Tasks;
using TripApp.Models;

namespace TripApp.Repositories
{
    public interface IClientRepository
    {
        Task<IEnumerable<Client>> GetAllAsync();
        Task<Client> GetByIdAsync(int id);
        Task<Client> GetByEmailAsync(string email);
        Task AddAsync(Client client);
        Task UpdateAsync(Client client);
        Task RemoveAsync(Client client);
    }
}
```

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using TripApp.Data;
using TripApp.Models;

namespace TripApp.Repositories
{
    public class ClientRepository : IClientRepository
    {
        private readonly TripContext _context;

        public ClientRepository(TripContext context)
        {
            _context = context;
        }

        public async Task<IEnumerable<Client>> GetAllAsync()
        {
            return await _context.Clients.ToListAsync();
        }
```

```
        public async Task<Client> GetByIdAsync(int id)
        {
            return await _context.Clients.FindAsync(id);
        }

        public async Task<Client> GetByEmailAsync(string email)
        {
            return await _context.Clients.FirstOrDefaultAsync(c => c.Email ==
email);
        }

        public async Task AddAsync(Client client)
        {
            _context.Clients.Add(client);
            await _context.SaveChangesAsync();
        }

        public async Task UpdateAsync(Client client)
        {
            _context.Entry(client).State = EntityState.Modified;
            await _context.SaveChangesAsync();
        }

        public async Task RemoveAsync(Client client)
        {
            _context.Clients.Remove(client);
            await _context.SaveChangesAsync();
        }
    }
}
```

```
using System.Collections.Generic;
using System.Threading.Tasks;
using TripApp.Models;

namespace TripApp.Repositories
{
    public interface ITripRepository
    {
        Task<IEnumerable<Trip>> GetAllAsync();
        Task<Trip> GetByIdAsync(int id);
        Task AddAsync(Trip trip);
        Task UpdateAsync(Trip trip);
        Task DeleteAsync(int id);
        bool Exists(int id);
    }
}
```

```
```

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using TripApp.Data;
using TripApp.Models;

namespace TripApp.Repositories
{
    public class TripRepository : ITripRepository
    {
        private readonly TripContext _context;

        public TripRepository(TripContext context)
        {
            _context = context;
        }

        public async Task<IEnumerable<Trip>> GetAllAsync()
        {
            return await _context.Trips.ToListAsync();
        }

        public async Task<Trip> GetByIdAsync(int id)
        {
            return await _context.Trips.FindAsync(id);
        }

        public async Task AddAsync(Trip trip)
        {
            _context.Trips.Add(trip);
            await _context.SaveChangesAsync();
        }

        public async Task UpdateAsync(Trip trip)
        {
            _context.Entry(trip).State = EntityState.Modified;
            await _context.SaveChangesAsync();
        }

        public async Task DeleteAsync(int id)
        {
            var trip = await _context.Trips.FindAsync(id);
            _context.Trips.Remove(trip);
            await _context.SaveChangesAsync();
        }

        public bool Exists(int id)
        {
            return _context.Trips.Any(e => e.TripId == id);
        }
    }
}
```

```
using System.Collections.Generic;
using System.Threading.Tasks;
using TripApp.Models;

namespace TripApp.Repositories
{
    public interface IReservationRepository
    {
        Task<Reservation> GetByIdAsync(int id);
        Task<IEnumerable<Reservation>> GetAllAsync();
        Task AddAsync(Reservation reservation);
        Task UpdateAsync(Reservation reservation);
        Task DeleteAsync(int id);
    }
}
```

```
using Microsoft.EntityFrameworkCore;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using TripApp.Data;
using TripApp.Models;

namespace TripApp.Repositories
{
    public class ReservationRepository : IReservationRepository
    {
        private readonly TripContext _context;

        public ReservationRepository(TripContext context)
        {
            _context = context;
        }

        public async Task<Reservation> GetByIdAsync(int id)
        {
            return await _context.Reservations.FindAsync(id);
        }

        public async Task<IEnumerable<Reservation>> GetAllAsync()
        {
            return await _context.Reservations.ToListAsync();
        }

        public async Task AddAsync(Reservation reservation)
        {
            _context.Reservations.Add(reservation);
            await _context.SaveChangesAsync();
```

```
        }

        public async Task UpdateAsync(Reservation reservation)
        {
            _context.Entry(reservation).State = EntityState.Modified;
            await _context.SaveChangesAsync();
        }

        public async Task DeleteAsync(int id)
        {
            var reservation = await _context.Reservations.FindAsync(id);
            if (reservation != null)
            {
                _context.Reservations.Remove(reservation);
                await _context.SaveChangesAsync();
            }
        }
    }
}
```

Następnie musiałem dostosować kontrolery aby wykorzystywały repozytoria zamiast kontekstu, wklejanie całego kodu mija się tutaj troszkę z celem, ponieważ byłoby go tutaj troszkę sporo, ale jakiś gigantycznych zmian tam nie było

Ostatecznie aby nasze repozytoria działały musiałem zarejestrować je w aplikacji

```
builder.Services.AddScoped<IReservationRepository, ReservationRepository>();
builder.Services.AddScoped<IClientRepository, ClientRepository>();
builder.Services.AddScoped<ITripRepository, TripRepository>();
```

Wnioski

1. **Oddzielenie logiki biznesowej od dostępu do danych**: Repozytorium oddziela logikę biznesową aplikacji od szczegółów implementacyjnych dotyczących dostępu do danych. Dzięki temu aplikacja staje się bardziej modularna i łatwiejsza do zarządzania, ponieważ zmiany w dostępie do danych nie wymagają zmian w warstwie logiki biznesowej.

2. **Łatwość testowania**: Repozytorium umożliwia łatwiejsze testowanie aplikacji poprzez wprowadzenie zależności od interfejsów repozytorium. Dzięki temu możemy stosować techniki testowania jednostkowego i mockowania, co ułatwia pisanie testów jednostkowych dla różnych warstw aplikacji.

3. **Zwiększona przejrzystość kodu**: Użycie repozytorium pozwala na bardziej czytelny i zorganizowany kod. Logika dostępu do danych jest izolowana w osobnych klasach repozytorium, co ułatwia zrozumienie i utrzymanie kodu.

4. **Uniezależnienie od konkretnej technologii dostępu do danych**: Repozytorium pozwala na uniezależnienie się od konkretnej technologii dostępu do danych. Dzięki temu możemy łatwo zmieniać dostawców baz danych lub technologie ORM, nie wpływając na logikę biznesową aplikacji.

5. **Reużywalność kodu**: Repozytorium umożliwia ponowne wykorzystanie kodu dostępu do danych w różnych częściach aplikacji. Możemy wykorzystać te same repozytoria w różnych modułach aplikacji lub nawet w innych projektach.

6. **Łatwa skalowalność**: Dzięki zastosowaniu repozytorium, aplikacja staje się łatwiejsza do skalowania. Możemy dodawać nowe funkcjonalności aplikacji lub rozszerzać istniejące, nie zmieniając istniejącej logiki biznesowej ani sposobu dostępu do danych.