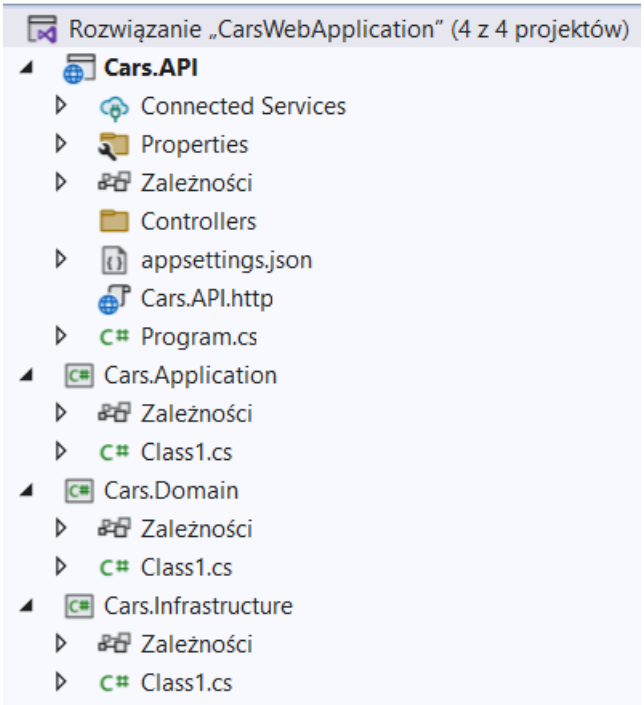
**Instrukcja tworzenia szkieletu WebApi**

1. Tworzymy w jednej solucji (CarsWebApplication) cztery projekty. Najpierw tworzymy projekt Cars.API (typu Internetowy interfejs API platformy ASP.NET Core) w wersji .NET 8.0.
2. Usuńmy automatycznie wygenerowane pliki: `WeatherForecast.cs` oraz `Controllers/WeatherForecastController.cs`
3. Dodajemy do solucji trzy nowe projekty typu biblioteka klas. Projektom zostały nadane nazwy: `Cars.Domain`, `Cars.Application` oraz `Cars.Infrastructure`. Strukturę solucji przedstawiono na poniższym rysunku.

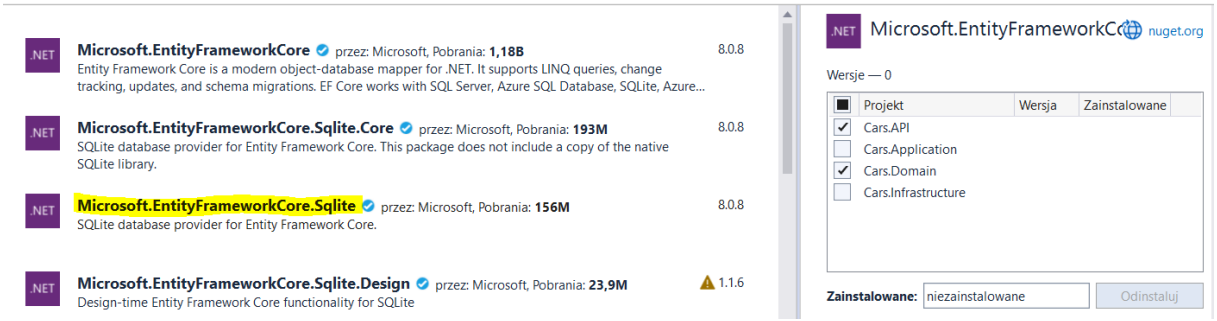



4. Dodajemy odpowiednie zależności między projektami:
   - `Cars.Domain` nie zależy od żadnego projektu,
   - `Cars.Infrastructure` zależy od `Cars.Domain`,
   - `Cars.Application` zależy od `Cars.Domain` oraz `Cars.Infrastructure`,
   - `Cars.API` zależy od `Cars.Application`.

5. Tworzymy model danych w projekcie `Cars.Domain` (zobacz poniższy rysunek).

```
namespace Cars.Domain
{
    1 odwołanie
    public enum FuelType { Petrol, Hybrid, Diesel, LPG }
    1 odwołanie
    public enum BodyType { Hatchback, Sedan, Kombi, SUV, Roadster }
    Odwołania: 0
    public class Car
    {
        Odwołania: 0
        public Guid Id { get; set; }
        Odwołania: 0
        public string Brand { get; set; }
        Odwołania: 0
        public string Model { get; set; }
        Odwołania: 0
        public int DoorsNumber { get; set; }
        Odwołania: 0
        public int LuggageCapacity { get; set; }
        Odwołania: 0
        public int EngineCapacity { get; set; }
        Odwołania: 0
        public FuelType FuelType { get; set; }
        Odwołania: 0
        public DateTime ProductionDate { get; set; }
        Odwołania: 0
        public double CarFuelConsumption { get; set; }
        Odwołania: 0
        public BodyType BodyType { get; set; }
    }
}
```

6. W celu skorzystania z narzędzia EF Core, instalujemy paczkę NuGet o nazwie: `Microsoft.EntityFrameworkCore.Sqlite` zarówno w projekcie `Cars.API`, jak i `Cars.Infrastructure`.




7. W warstwie `Cars.Infrastructure` tworzymy klasę `DataContext` dziedziczącą po klasie `DbContext` (zobacz rysunek poniżej). Obiekt `DbSet<Car> Cars` odzwierciedla odpowiednią tabelę (Cars) w bazie danych.

```
using Cars.Domain;
using Microsoft.EntityFrameworkCore;

namespace Cars.Infrastructure
{
    1 odwołanie
    public class DataContext : DbContext
    {
        Odwołania: 0
        public DataContext(DbContextOptions options) : base(options) {}

        // Cars to nazwa tabeli w bazie danych
        Odwołania: 0
        public DbSet<Car> Cars { get; set; }
    }
}
```

8. W pliku `Cars.API/Program.cs` możemy dokonać konfiguracji aplikacji poprzez dodanie lub zmianę ustawień wybranych serwisów. Odpowiada za to obiekt `builder.Services`. Dodaj serwis odpowiedzialny za kontekst bazy danych. Dodatkowo zapisz nazwę pliku bazy danych w pliku konfiguracyjnych `appsettings.json` pod kluczem `DefaultConnection` (zobacz dwa poniższe rysunki).

```
// kontekst bazy danych
builder.Services.AddDbContext<DataContext>(opt =>
{
    opt.UseSqlite(builder.Configuration.GetConnectionString("DefaultConnection"));
});
```

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "DefaultConnection": "Data source=cars.db"
  }
}
```

9. Następnie dokonujemy migracji bazy danych. W tym celu, najpierw instalujemy paczkę NuGet `Microsoft.EntityFrameworkCore.Design` dla projektu `Cars.API` . Następnie wpisz w PMC (konsola menadżera pakietów w VS) komendę:

```
dotnet ef migrations add InitialCreate -s Cars.API -p Cars.Persistence
```

Być może będzie potrzeba, aby ustawić jako projekt startowy `Cars.Infrastructure` i/lub ponownie skompilować rozwiązanie. W projekcie `Cars.Infrastructure` zostanie utworzony folder `Migrations`, a w nim plik migracji (właściwość `Id` automatycznie będzie kluczem głównym w tabeli dzięki przyjętej konwencji).
Skróty `–s` oraz `–p` odnoszą się do pełnych nazw opcji `--startup-project` oraz `--project`. W razie problemów należy stosować pełne nazwy zamiast skrótów.

10. Stwórz plik bazy danych na podstawie utworzonej migracji w kroku poprzednim. Można to zrobić, na przykład, pisząc kod w C#, w pliku `Program.cs` (punkt wejściowy aplikacji) przed instrukcją `app.Run();`. (zobacz poniższy rysunek). Jeśli wcześniej nie było bazy danych, to wówczas baza zostanie utworzona – plik z bazą znajdować się będzie w projekcie `Cars.API`.

```
35  using var scope = app.Services.CreateScope();
36  var services = scope.ServiceProvider;
37
38  // próba utworzenia bazy danych
39  try
40  {
41      var context = services.GetRequiredService<DataContext>();
42      context.Database.Migrate();
43  }
44  catch (Exception ex)
45  {
46      var logger = services.GetRequiredService<ILogger<Program>>();
47      logger.LogError(ex, "An error occured during migration");
48  }
49
50  app.Run();
```

11. Ten punkt jest opcjonalny i służy dodaniu do bazy kilku rekordów. Tworzymy klasę `Seed` (warstwa `Cars.Infrastructure`) zawierającą kilka przykładowych rekordów (zobacz poniższy rysunek). Następnie, wewnątrz bloku `try` (w pliku `Program.cs`), po instrukcji z migracją używamy metody `SeedData` - instrukcja `await Seed.SeedData(context);`

```
10      public class Seed
11      {
            1 odwołanie
12          public static async Task SeedData(DataContext context)
13          {
14              // jeśli baza ma jakieś rekordy to nic nie rób
15              if (context.Cars.Any()) return;
16
17              var cars = new List<Car>
18              {
19                  new Car
20                  {
21                      Brand = "Mazda",
22                      Model = "CX60",
23                      DoorsNumber = 5,
24                      LuggageCapacity = 570,
25                      EngineCapacity = 2488,
26                      FuelType = FuelType.Hybrid,
27                      ProductionDate = DateTime.UtcNow.AddMonths(-1),
28                      CarFuelConsumption = 18.1,
29                      BodyType = BodyType.SUV
30                  },
31                  new Car
32                  {
33                      Brand = "Renault",
34                      Model = "Clio II",
35                      DoorsNumber = 5,
36                      LuggageCapacity = 300,
37                      EngineCapacity = 1149,
38                      FuelType = FuelType.Petrol,
39                      ProductionDate = DateTime.UtcNow.AddYears(-18),
40                      CarFuelConsumption = 7.2,
```

12. Na koniec dodamy jeden kontroler z dwoma metodami zwracającymi listę wszystkich aut oraz wybrane auto po id. Tworzymy najpierw pustą klasę `BaseApiController` (w `Cars.API/Controllers/`), która dziedziczy po klasie `ControllerBase`. Przy użyciu atrybutu `[Route("api/[controller]")]` definiujemy domyślną ścieżkę dla endpointów. Wszystkie kontrolery będą dziedziczyć po klasie `BaseApiController`. Następnie tworzymy klasę `CarsController`. Klasa ta zawiera prywatne pole z kontekstem danych. Kontekst danych będzie wstrzykiwany przez konstruktor. W klasie zawarto dwie metody (powstaną dwa endpointy), jedna dla pobrania wszystkich aut oraz druga dla pobrania auta na podstawie `id`. Aplikację można przetestować w przeglądarce lub Postmanie wpisując odpowiedni adres, np. `https://localhost:7243/api/cars`

```
1  using Microsoft.AspNetCore.Mvc;
2
3  namespace Cars.API.Controllers
4  {
5      [ApiController]
6      [Route("api/[controller]")] // endpoint zawsze zaczyna się "api/..."
       1 odwołanie
7      public class BaseApiController : ControllerBase { }
8  }
```

```
6  namespace Cars.API.Controllers
7  {
       1 odwołanie
8      public class CarsController : BaseApiController
9      {
10         private readonly DataContext _context;
           Odwołania: 0
11         public CarsController(DataContext context)
12         {
13             _context = context;
14         }
15
16         [HttpGet] // api/cars
           Odwołania: 0
17         public async Task<ActionResult<List<Car>>> GetCars()
18         {
19             return await _context.Cars.ToListAsync();
20         }
21
22         [HttpGet("{id}")] // /api/cars/id
           Odwołania: 0
23         public async Task<ActionResult<Car>> GetCar(Guid id)
24         {
25             return await _context.Cars.FindAsync(id);
26         }
27     }
28 }
```

13. Należy jeszcze dodać politykę cors, aby móc korzystać np. z Postmana. W przeciwnym przypadku adres będzie blokowany.

```
20 // dodanie serwisu polityki cors
21 builder.Services.AddCors(opt =>
22 {
23     opt.AddPolicy("CorsPolicy", policy =>
24     {
25         // ufamy temu adresowi, niezależnie od nagłówka lub metody (POST, PUT, etc.)
26         policy.AllowAnyHeader().AllowAnyMethod().WithOrigins("http://localhost:7072");
27     });
28 });
```