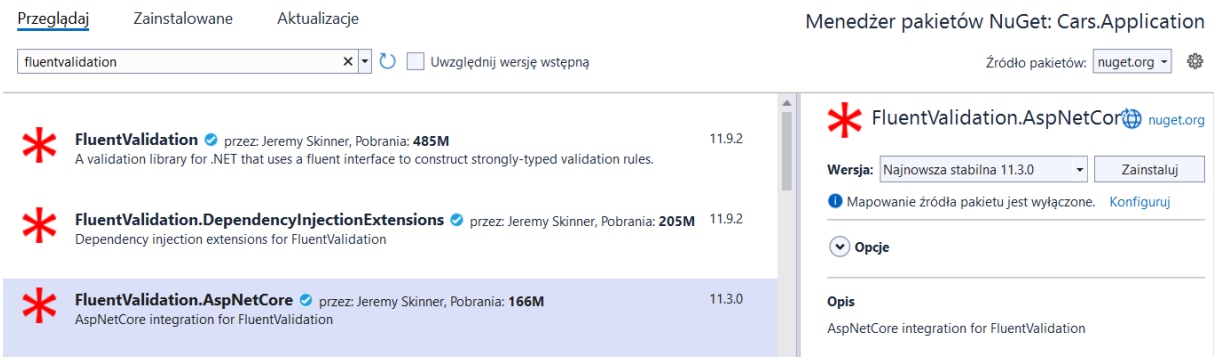
Instrukcja implementacji obsługi błędów

1. Walidację przeprowadzimy na poziomie warstwy aplikacji (`Cars.Application`). Instalujemy paczkę NuGet o nazwie: `FluentValidation.AspNetCore`.



2. Dokonaj integracji z biblioteką `FluentValidation` (plik `Program.cs`):

   - `builder.Services.AddFluentValidationAutoValidation();` - rejestruje `FluentValidation` do automatycznej walidacji modeli w aplikacji. Zazwyczaj działa to w połączeniu z walidacją modeli w ASP.NET Core (taką jak walidacja atrybutami `[Required]`). Dzięki temu, kiedy model jest przekazywany np. jako parametr do kontrolera, `FluentValidation` automatycznie przeprowadzi walidację i zwróci odpowiedni błąd, jeśli walidacja się nie powiedzie. Ułatwia to proces walidacji, ponieważ nie trzeba ręcznie wywoływać walidatorów
   - `builder.Services.AddValidatorsFromAssemblyContaining<Create>();` - rejestruje wszystkie walidatory znajdujące się w tym samym zestawie (assembly) co typ `Create`. Innymi słowy, przeszukuje assembly, w którym znajduje się klasa `Create`, w poszukiwaniu klas implementujących `IValidator<T>`, i automatycznie rejestruje je w kontenerze DI (Dependency Injection). Sprawia to, że nie trzeba rejestrować każdego walidatora ręcznie. Wystarczy, że walidatory są zdefiniowane w tym samym assembly, a ta linia zajmie się ich automatyczną rejestracją.

3. Następnie w projekcie `Cars.Application` tworzymy nową klasę `CarValidator`. Definiujemy w niej zasady walidacji. W prezentowanym przykładzie odnosi się to do sytuacji przesyłania danych do żądań create oraz edit. Zakładamy, że wszystkie właściwości klasy `Car` (poza `Id`) są wymagane i nie mogą być puste. Dodatkowo na właściwości `DoorsNumber` i `CarFuelConsumption` nałożono dodatkowe reguły walidacyjne. Następnie tworzymy klasę `CommandValidator` wewnątrz klasy `Create` oraz `Edit`. Rysunki poniżej prezentują zawartość obu klas. Przetestuj w dowolnym narzędziu (np. Postman) odpowiednie endpointy (create, edit). W przypadku nie dostarczenia, którejś z wymaganych właściwości powinniśmy otrzymać odpowiedź serwera z kodem 400 – Bad Request.

```
using Cars.Domain;
using FluentValidation;
namespace Cars.Application.Cars
{
    Odwołania: 3
    public class CarValidator : AbstractValidator<Car>
    {
        Odwołania: 2
        public CarValidator()
        {
            RuleFor(x => x.Brand).NotEmpty().WithMessage("Brand is required");
            RuleFor(x => x.Model).NotEmpty().WithMessage("Model is required");
            RuleFor(x => x.DoorsNumber).InclusiveBetween(2, 10).NotEmpty()
                .WithMessage("Doors number is required and must be between 2 and 10");
            RuleFor(x => x.LuggageCapacity).NotEmpty().WithMessage("LuggageCapacity is required");
            RuleFor(x => x.EngineCapacity).NotEmpty().WithMessage("EngineCapacity is required");
            RuleFor(x => x.FuelType).NotEmpty().WithMessage("FuelType is required");
            RuleFor(x => x.ProductionDate).NotEmpty().WithMessage("ProductionDate is required");
            RuleFor(x => x.CarFuelConsumption).GreaterThan(0).NotEmpty()
                .WithMessage("CarFuelConsumption is required");
            RuleFor(x => x.BodyType).NotEmpty().WithMessage("BodyType is required"); ;
        }
    }
}
```

```
public class CommandValidator : AbstractValidator<Command>
{
    Odwołania: 0
    public CommandValidator()
    {
        RuleFor(x => x.Car).SetValidator(new CarValidator());
    }
}
```

4. Zajmiemy się teraz obsługą żądania pobrania konkretnego auta po `Id`. W przypadku, gdy do kontrolera przyjdzie zapytanie, aby zwrócić obiekt, którego nie ma w bazie, wówczas powinniśmy zwrócić błąd 404 - Not Found. Obsługę błędów przeprowadzimy na poziomie handlerów (czyli w projekcie `Cars.Application`). Jednak, handlery nie należą do warstwy `Cars.API`, stąd nie możemy po prostu zwrócić `NotFound()`. Musimy odpowiedź ,,opakować" w odpowiedni obiekt. W tym celu wewnątrz warstwy `Cars.Application` utwórzmy klasę odpowiedzi - `Result`.
   Klasa `Result<T>` jest używana do opakowywania odpowiedzi w aplikacjach typu Web API, aby w jednolity sposób zwracać wyniki operacji oraz obsługiwać błędy. Dzięki temu podejściu zyskujemy bardziej spójny i czytelny sposób komunikacji między API a jego klientami (np. frontendem lub inną usługą korzystającą z tego API).
   `IsSuccess` – wskazuje, czy operacja zakończyła się sukcesem. Działa jako flaga informująca o tym, czy wynik operacji jest poprawny (true) czy nie (false).
   `Value` – przechowuje wynik operacji w przypadku sukcesu. Typ T pozwala na elastyczność, więc możemy zwrócić dowolny obiekt, np. `Car`, `User` lub inny typ w zależności od kontekstu.
   `Error` – przechowuje wiadomość o błędzie, jeśli operacja zakończyła się niepowodzeniem. To pole pozwala na przekazanie szczegółów dotyczących problemu, który wystąpił.
   Dodatkowa logika w metodach statycznych `Success` i `Failure` – te metody ułatwiają tworzenie obiektów `Result<T>` dla udanych i nieudanych operacji. Zamiast ręcznie tworzyć obiekt `Result<T>` i ustawiać jego właściwości, można użyć `Result<T>.Success()` lub `Result<T>.Failure()`.

`Success(T value)` – tworzy i zwraca obiekt `Result<T>`, reprezentujący pomyślny wynik operacji. Ustawia `IsSuccess` na `true` i przekazuje wartość do `Value`.
`Failure(string error)` – tworzy obiekt `Result<T>`, który reprezentuje nieudaną operację. Ustawia `IsSuccess` na `false` i przechowuje komunikat o błędzie w `Error`.

```
10      public class Result<T>
11      {
            Odwołania: 2
12          public bool IsSuccess { get; set; }
            1 odwołanie
13          public T Value { get; set; }
            1 odwołanie
14          public string Error { get; set; }
            Odwołania: 0
15          public static Result<T> Success(T Value)
16          {
17              return new Result<T> { IsSuccess = true, Value = Value };
18          }
            Odwołania: 0
19          public static Result<T> Failure(string error)
20          {
21              return new Result<T> { IsSuccess = false, Error = error };
22          }
23      }
```

5. Należy teraz odpowiednio dostosować metodę `Handle` z klasy `Details`. Już nie będzie ona zwracać obiektu `Car`, a obiekt `Result<Car>`. Należy również dostosować kontroler, aby zwracał odpowiedni kod odpowiedzi. Poniżej znajdują się zarówno kod klasy `Details`, jak również metoda `GetCar` z kontrolera.

```
[HttpGet("{id}")] // /api/cars/id
public async Task<IActionResult> GetCar(Guid id)
{
    var result = await Mediator.Send(new Details.Query { Id = id });

    if (result == null)
        return NotFound();
    if (result.IsSuccess && result.Value != null)
        return Ok(result.Value);
    if (result.IsSuccess && result.Value == null)
        return NotFound();
    return BadRequest(result.Error);
}
```

```
7   public class Details
8   {
        Odwołania: 3
9       public class Query : IRequest<Result<Car>>
10      {
            Odwołania: 2
11          public Guid Id { get; set; }
12      }
13
            1 odwołanie
14      public class Handler : IRequestHandler<Query, Result<Car>>
15      {
16          // przekazujemy kontekst danych
17          private readonly DataContext _context;
            Odwołania: 0
18          public Handler(DataContext context)
19          {
20              _context = context;
21          }
22
            Odwołania: 0
23          public async Task<Result<Car>> Handle(Query request, CancellationToken cancellationToken)
24          {
25              var car =  await _context.Cars.FindAsync(request.Id);
26
27              return Result<Car>.Success(car);
28          }
29      }
30  }
```

6. W analogiczny sposób zmień kody klas: `Create, Delete, Edit, List` oraz odpowiadające im metody w kontrolerze. Poniżej zaprezentowano metody `Handle` odpowiednio klas `Create, Delete, Edit` i `List.`

Rys. Metoda Handle dla klasy Create

```
32      public async Task<Result<Unit>> Handle(Command request, CancellationToken cancellationToken)
33      {
34          // na tym etapie dodajemy auto tylko do pamięci
35          // jeszcze nie do bazy
36          _context.Cars.Add(request.Car);
37
38          // teraz następuje zapis w bazie
39          // SaveChangesAsync zwraca liczbę uaktualnionych wierszy w bazie danych
40          var result = await _context.SaveChangesAsync() > 0;
41
42          // tak naprawdę nie zwracamy konkretnej wartości,
43          // tylko informujemy, że proces obsługi się zakończył
44          // Unit jest ''nothing object''
45          return Result<Unit>.Success(Unit.Value);
46      }
```

Rys. Metoda Handle dla klasy Delete

```
23      public async Task<Result<Unit>> Handle(Command request, CancellationToken cancellationToken)
24      {
25          var car = await _context.Cars.FindAsync(request.Id);
26
27          // usuwamy tylko z ,,pamięci"
28          _context.Remove(car);
29
30          // zapisujemy zmiany w bazie danych
31          var result = await _context.SaveChangesAsync() > 0;
32
33          if (!result) return Result<Unit>.Faluire("Failed to delete the car");
34
35          return Result<Unit>.Success(Unit.Value);
36      }
```

Rys. Metoda Handle dla klasy Edit

```
32          public async Task<Result<Unit>> Handle(Command request, CancellationToken cancellationToken)
33          {
34              // pobieramy samochód z bazy danych po id
35              var car = await _context.Cars.FindAsync(request.Car.Id);
36              if (car == null) return null;
37
38              // edytujemy wybrane pola obiektu
39              // ewentualnie instalujemy automappera
40              car.Brand = request.Car.Brand ?? car.Brand;
41              car.Model = request.Car.Model ?? car.Model;
42              car.DoorsNumber = request.Car.DoorsNumber;
43              car.LuggageCapacity = request.Car.LuggageCapacity;
44              car.EngineCapacity = request.Car.EngineCapacity;
45              car.FuelType = request.Car.FuelType;
46              car.ProductionDate = request.Car.ProductionDate;
47              car.CarFuelConsumption = request.Car.CarFuelConsumption;
48              car.BodyType = request.Car.BodyType;
49
50              // zapisujemy zmiany
51              var result = await _context.SaveChangesAsync() > 0;
52
53              if (!result) return Result<Unit>.Failure("Failed to update car.");
54
55              return Result<Unit>.Success(Unit.Value);
56          }
```

Rys. Metoda Handle dla klasy List

```
22          public async Task<Result<List<Car>>> Handle(Query request, CancellationToken cancellationToken)
23          {
24              var result = await _context.Cars.ToListAsync();
25              return Result<List<Car>>.Success(result);
26          }
```

7. Przetestuj wszystkie endpointy.