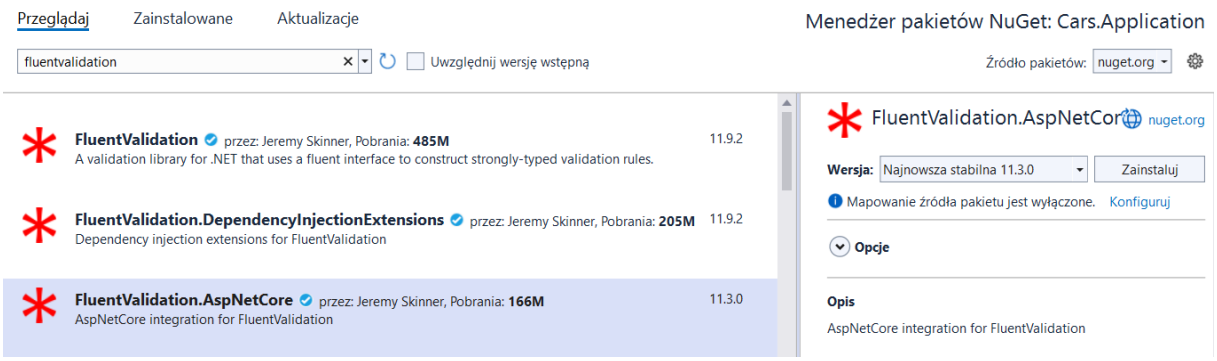
Instrukcja implementacji obsługi błędów

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.
   - `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 zestawie (assembly).

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. Na właściwości `DoorsNumber` i `CarFuelConsumption` nałożono dodatkowe reguły walidacyjne. Następnie tworzymy klasę `CommandValidator` wewnątrz klasy `Create` oraz `Edit`. Rysunki poniżej prezentują zawartość klasy `CarValidator` i odpowiedni fragment klas `Create` oraz `Edit`. 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.

```
// AbstractValidator<T> implementuje interfejs IValidator<T>
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 wywołać metody `NotFound()` (Metody `Ok()`, `NotFound()`, `BadRequest()`, `CreatedAtAction()`, `NoContent()`, `Unauthorized()` pochodzą z klasy bazowej `ControllerBase` w przestrzeni nazw `Microsoft.AspNetCore.Mvc`). 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. przeglądarką).
   `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 zwracaną 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`.

```
7   namespace Cars.Application
8   {
        Odwołania: 27
9       public class Result<T>
10      {
            Odwołania: 7
11          public bool IsSuccess { get; set; }
            Odwołania: 8
12          public T Value { get; set; }
            Odwołania: 6
13          public string Error { get; set; }
            Odwołania: 5
14          public static Result<T> Success(T value) =>
15              new Result<T> { IsSuccess = true, Value = value };
            Odwołania: 3
16          public static Result<T> Failure(string error) =>
17              new Result<T> { IsSuccess =  false, Error = error };
18      }
19  }
```

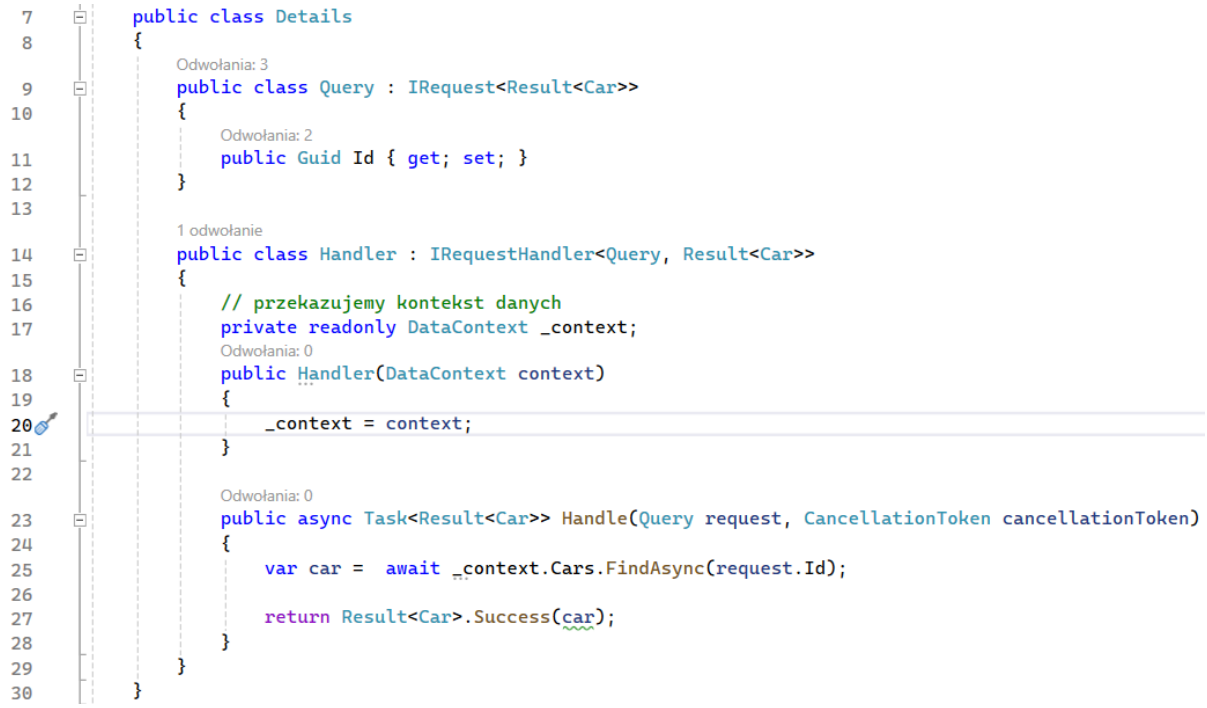
5. Należy teraz odpowiednio dostosować metodę `Handle` z klasy `Details`. Już nie będzie ona zwracać obiektu `Car`, a obiekt `Result<Car>`. Należy również dostosować kontroler, aby zwracał odpowiedni kod odpowiedzi. Poniżej znajdują się zarówno kod klasy `Details`, jak również metoda `GetCar` z kontrolera.

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

    if (result == null || result.Value == null)
        return NotFound(); // 404 – zasób nie istnieje

    if (result.IsSuccess)
        return Ok(result.Value); // 200 – znaleziono samochód

    return BadRequest(result.Error); // 400 – błąd zapytania
}
```



```
public class Details
{
    Odwołania: 3
    public class Query : IRequest<Result<Car>>
    {
        Odwołania: 2
        public Guid Id { get; set; }
    }

    1 odwołanie
    public class Handler : IRequestHandler<Query, Result<Car>>
    {
        // przekazujemy kontekst danych
        private readonly DataContext _context;
        Odwołania: 0
        public Handler(DataContext context)
        {
            _context = context;
        }

        Odwołania: 0
        public async Task<Result<Car>> Handle(Query request, CancellationToken cancellationToken)
        {
            var car =  await _context.Cars.FindAsync(request.Id);

            return Result<Car>.Success(car);
        }
    }
}
```

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`.

Kod. Metoda Handle dla klasy Create

```
public async Task<Result<Car>> Handle(Command request, CancellationToken cancellationToken)
{
	// Dodajemy nowy samochód do kontekstu (śledzony przez EF Core)
	_context.Cars.Add(request.Car);

	// Zapisujemy zmiany w bazie
	var success = await _context.SaveChangesAsync(cancellationToken) > 0;

	if (!success)
		return Result<Car>.Failure("Nie udało się zapisać samochodu do bazy danych");

	// Zwracamy obiekt, który faktycznie został utworzony (z Id)
	return Result<Car>.Success(request.Car);
}
```

Kod. Metoda Handle dla klasy Delete

```
public async Task<Result<Unit>> Handle(Command request, CancellationToken cancellationToken)
{
	var car = await _context.Cars.FindAsync(request.Id);

	// usuwamy tylko z ,,pamięci"
	_context.Remove(car);

	// zapisujemy zmiany w bazie danych
	var result = await _context.SaveChangesAsync() > 0;
```

```
        if (!result)
                return Result<Unit>.Failure("Failed to delete the   car");

        return Result<Unit>.Success(Unit.Value)
}
```

Kod. Metoda Handle dla klasy Edit

```
public async Task<Result<Unit>> Handle(Command request, CancellationToken
cancellationToken)
{
        // pobieramy samochód z bazy danych po id
        var car = await _context.Cars.FindAsync(request.Car.Id);
        if (car == null) return null;

        // edytujemy wybrane pola obiektu
        // ewentualnie instalujemy automappera
        car.Brand = request.Car.Brand ?? car.Brand;
        car.Model = request.Car.Model ?? car.Model;
        car.DoorsNumber = request.Car.DoorsNumber;
        car.LuggageCapacity = request.Car.LuggageCapacity;
        car.EngineCapacity = request.Car.EngineCapacity;
        car.FuelType = request.Car.FuelType;
        car.ProductionDate = request.Car.ProductionDate;
        car.CarFuelConsumption = request.Car.CarFuelConsumption;
        car.BodyType = request.Car.BodyType;

        // zapisujemy zmiany
        var result = await _context.SaveChangesAsync() > 0;

        if (!result) return Result<Unit>.Failure("Failed to update car.");

        return Result<Unit>.Success(Unit.Value);
}
```

Kod. Metoda Handle dla klasy List

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

Poniżej zaprezentowano również kod metody CreateCar() z kontrolera:

```
[HttpPost]
public async Task<IActionResult> CreateCar(Car car)
{
        var result = await Mediator.Send(new Create.Command { Car = car });

        if (result == null)
                return BadRequest(); // 400 – niepoprawne dane

        if (result.IsSuccess && result.Value != null)
        {
                // 201 – zasób utworzony; nagłówek Location zawiera adres
                nowego samochodu
```

```
            return CreatedAtAction(
                  nameof(GetCar),    // metoda, do której prowadzi link
                  new { id = result.Value.Id }, // parametr route
                  result.Value  // zwracany obiekt
            );
      }

      return BadRequest(result.Error);
}
```

7. Przetestuj wszystkie endpointy.