Fleet Management App

Realisatie

Bachelor in de Toegepaste Informatica

Matthias Reynders

Academiejaar 2020-2021

Campus Geel, Kleinhoefstraat 4, BE-2440 Geel

INHOUDSTAFEL

1 INLEIDING

De Fleet Management App is mijn tweede opdracht tijdens mijn stage bij AllPhi. Dit project omvat het bouwen van een applicatie voor AllPhi om hun wagenpark te beheren. De applicatie maakt het voor werknemers mogelijk om aanvragen met betrekking tot hun wagen in te dienen. Vervolgens maakt de fleetmanager van de applicatie gebruik om de aanvragen af te handelen.

In dit document worden de stappen beschreven in de ontwikkeling van de applicatie. Eerst maak ik een analyse van de opdracht. Uit de opdracht haal ik de benodigde functionaliteiten en het domeinmodel voor de applicatie. Daarna kan ik al beginnen nadenken over het uiterlijk van de applicatie en mock-ups ontwerpen. Dan begint het ontwikkelen waarbij ik eerst mijn databank opzet en de Data Access Laag. Vervolgens werk ik de functionaliteiten uit per gebruiker. Eerst de werknemer die aanvragen doet, erna de fleetmanager die aanvragen afhandelt.

2 ANALYSE

In dit hoofdstuk beschrijf ik mijn analyse van de opdracht. Hierin komen volgende onderdelen aan bod: technische vereisten, functionaliteiten, domeinmodel en mock-ups.

2.1 Technische vereisten

De technische vereisten van de opdracht bepalen in welke programmeertaal de oplossing dient geschreven te worden, m.b.v. welk framework en welke design-patterns geïmplementeerd moeten worden. Omdat AllPhi gespecialiseerd is in softwareontwikkeling met een duidelijke focus op Microsoft .NET zijn de vereisten voor deze opdracht ook .NET-gericht.

Als basis voor de applicatie gebruik ik .NET5 en MsSql server. De applicatie zelf wordt opgebouwd uit een 4 lagen systeem. De Data Acces Laag maakt gebruik van 2 ORMs EF Core en Dapper. EF Core wordt toegepast voor alle CRUD-operaties en Dapper wordt Gebruikt voor het opstellen van de meer complexere queries. Binnen de EF Core Data Access Laag wordt er gebruik gemaakt van een generic repository zodat er geen herhaling van code plaatsvind.

Alle Business logica wordt centraal beheerd in de Business Laag. Deze laag dient als centraal aanspreekpunt voor de verschillende endpoints die data naar de buitenwereld beschikbaar stellen.

Als endpoints worden er een readonly-API en write-API voorzien langs de kant van de werknemer/gebruiker. Een API die alle CRUD-operaties kan uitvoeren wordt opgesteld voor de fleetmanager.

De front-end bestaat uit 2 delen. Hierin mocht ik vrij kiezen en koos ik een Xamarin front-end voor de werknemer en een React front-end voor de fleetmanager. De verklaring voor deze keuze staat in het begin van hoofdstuk 4: Front-end.

2.2 Functionaliteiten

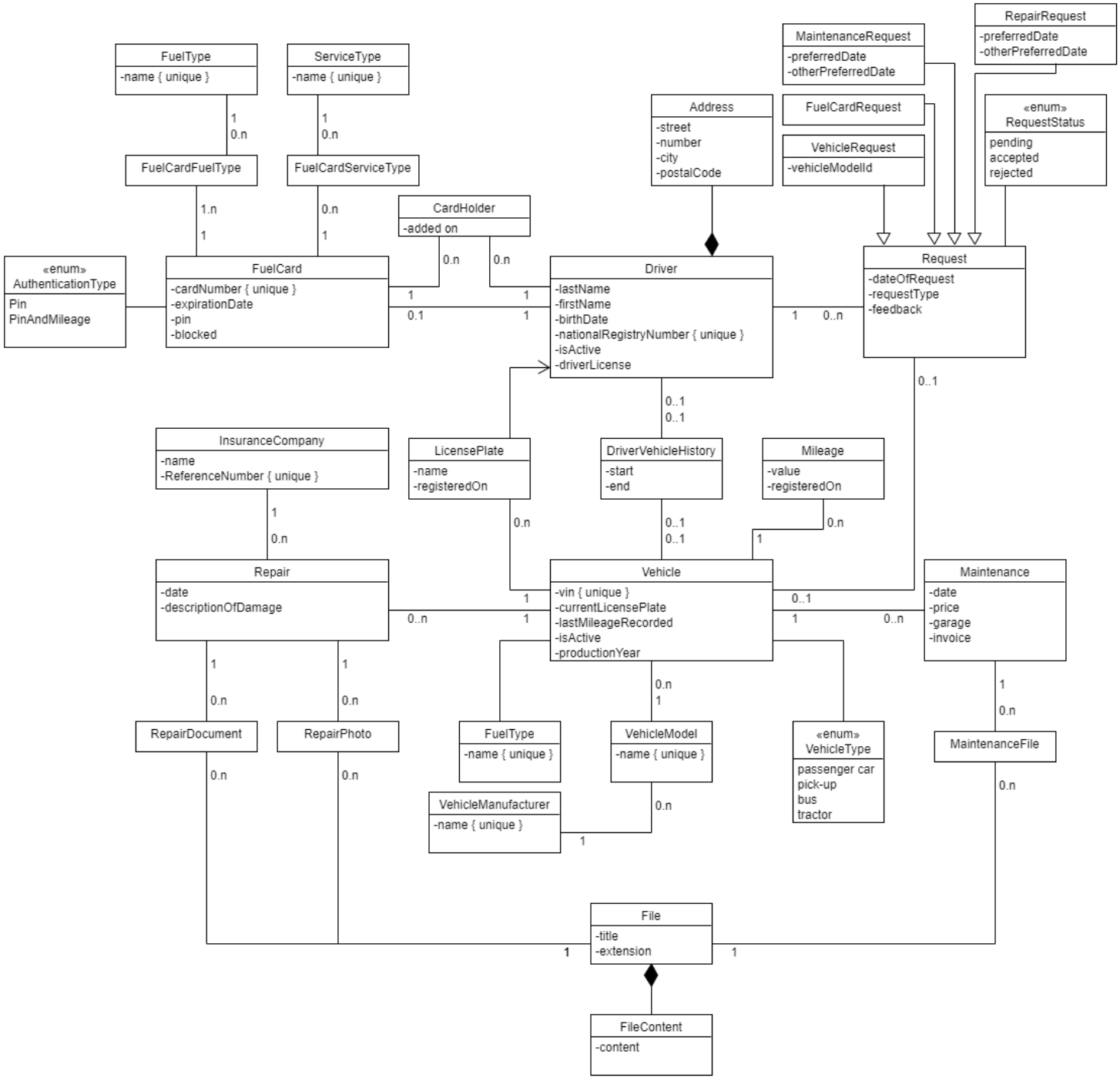
De functionaliteiten beschrijven wat het systeem kan. Op basis van het type gebruiker deel ik de functionaliteiten op in twee groepen. Deze twee groepen zijn werknemer en fleetmanager.

Voor de fleetmanager is er een overzicht nodig van alle voertuigen, tankkaarten en aanvragen. Verder nog moet een fleetmanager in staat zijn om deze aanvragen af te handelen.

Een werknemer moet aanvragen kunnen doen zoals een onderhoud/herstelling of tankkaart aanvragen. Een voertuig aanvragen staat niet in de opdracht, maar deze functionaliteit werk ik wel uit.

2.3 Domeinmodel

Het domeinmodel voor de Fleet Management App ziet er als volgt uit.



Figuur 1: domeinmodel

Dit zijn alle entiteiten die nodig zijn om de applicatie als een geheel te laten werken. De belangrijkste hiervan zijn Driver, Vehicle, FuelCard en Request. Er zijn 4 entiteiten die overerven van Request, dit zijn specifieke aanvragen zoals een FuelCardRequest om een tankkaart aan te vragen.

2.4 Mock-ups

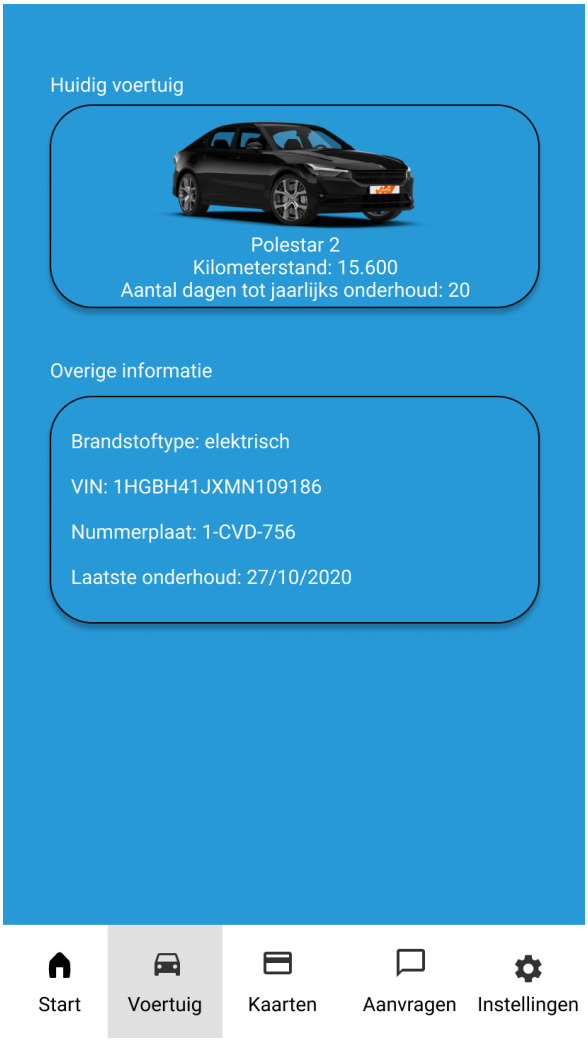
Dit hoofdstuk gaat over de mock-ups die ik tijdens de analyse heb gemaakt als hulpmiddel voor mijn functionele presentatie. Omdat ik vrij ben in hoe de applicatie er zal uit zien, zullen deze mock-ups al een goed idee geven over het eindresultaat waar ik naartoe wil werken.

2.4.1 Mock-ups werknemer front-end

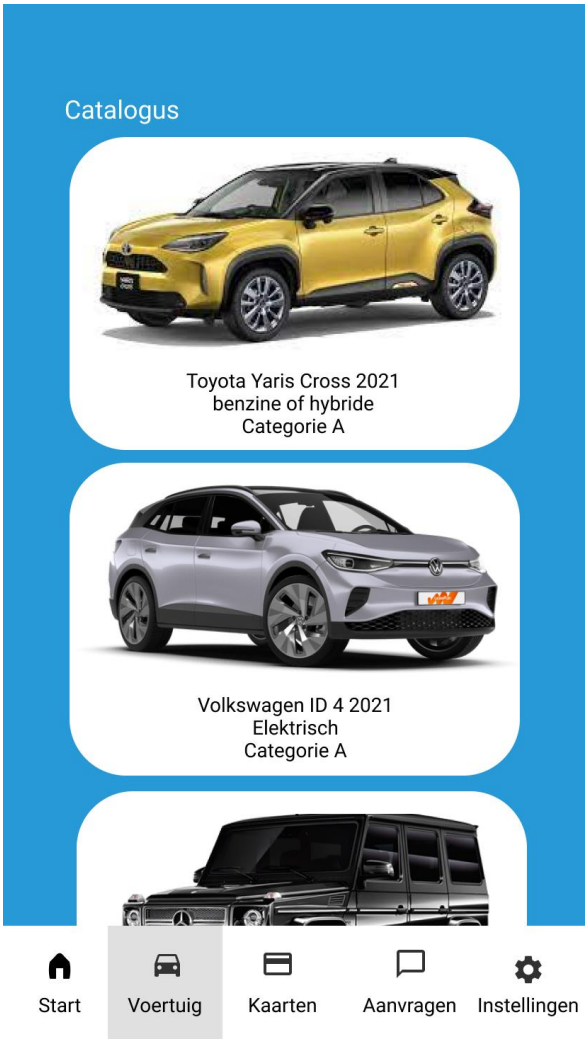
Dit zijn de mock-ups om te schetsen hoe de applicatie voor een werknemer er zal uitzien. Ik heb hier al gekozen dat de front-end voor de werknemer een mobiele app zal worden, omdat een werknemer slechts een paar acties moet uitvoeren en een overzicht van zijn/haar aanvragen moet kunnen opvragen.

Figuur 2: Inlogscherm

Figuur 3: Landingspagina



Figuur 4: VoertuigPagina



Figuur 5: CatalogusPagina

Figuur 6: TankkaartPagina

Figuur 7: TankkaartPagina

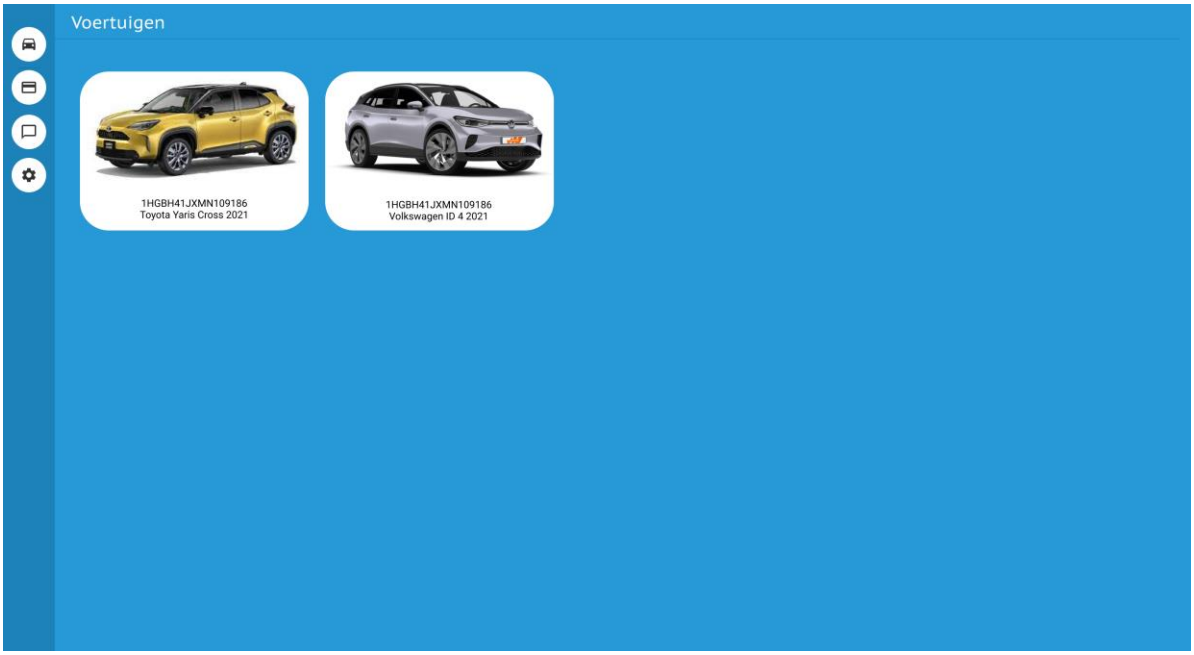
Figuur 8: AanvragenPagina

Figuur 9: InstellingenPagina

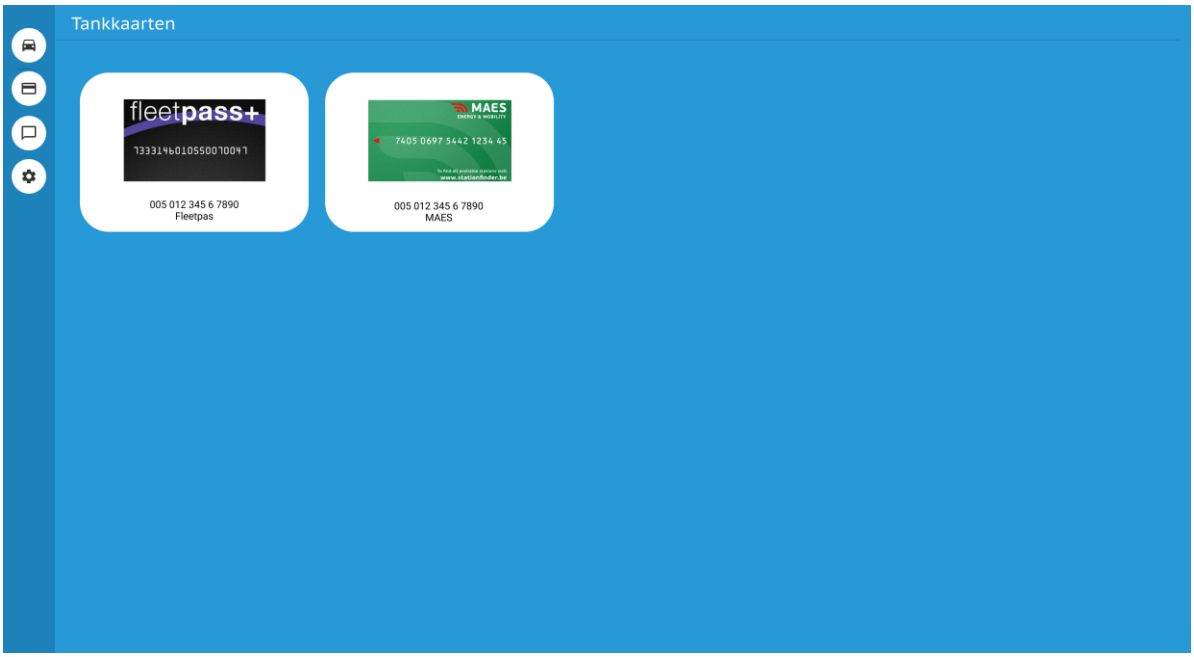
2.4.2 Mock-ups fleetmanager front-end

Deze mock-ups tonen hoe de applicatie eruit zal zien voor een fleetmanager. Een overzicht van gegevens is vooral belangrijk voor deze gebruiker en dit had ik graag op 2 manieren uitgewerkt. Een lijstweergave (voor aanvragen vooral), oftewel een kaartweergave (interessanter voor alle voertuigen te tonen).

Figuur 10: Inlogscherm



Figuur 11: Voertuigenscherm



Figuur 12: Tankkaartenscherm

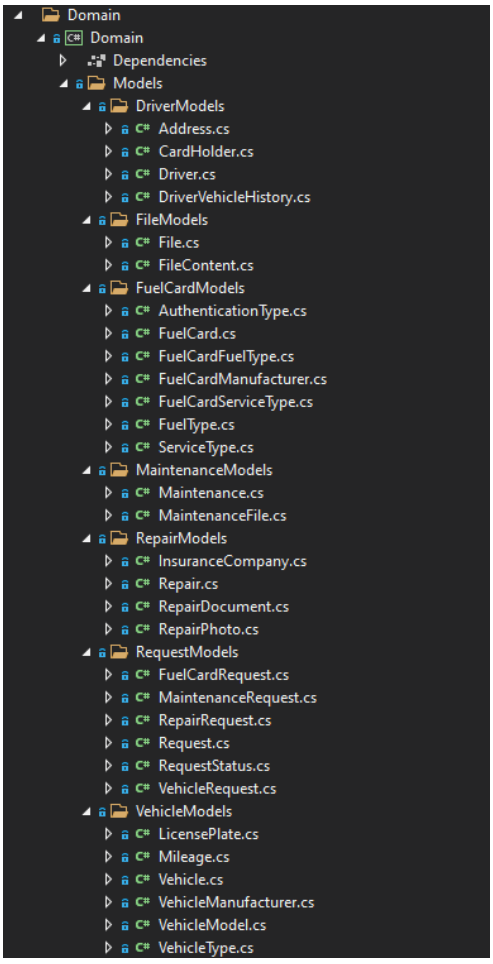
3 BACK-END

In dit hoofdstuk beschrijf ik hoe ik de back-end uit werk. Dit wil niet zeggen dat de back-end helemaal af is wanneer ik begin aan de front-ends. Integendeel voor het schrijven van de applicatie ben ik agile te werk gegaan. Een voorbeeld hiervan is een werknemer die zijn/haar voertuig opvraagt. Hierbij zorg ik er eerst voor dat de data uit de databank wordt gelezen en wordt doorgegeven naar een endpoint. Vervolgens stuurt de mobiele app een verzoek naar deze endpoint en toont het voertuig van de werknemer.

3.1 Domain

In het begin is het vooral belangrijk om de entiteiten die nodig zijn voor de applicatie te doen werken gedefinieerd worden in klassen. Ook hier ben ik agile te werk gegaan en heb ik eerst de klasse aangemaakt die nodig zijn voor een MVP (minimum viable product).

Het is van enorm belang om de relatie tussen deze entiteiten zo te definiëren dat het systeem makkelijk uitbreidbaar blijft. De opdracht impliceert verschillende veel op veel en één op één (of nul) relaties. Zo heeft een tankkaart maar één actieve relatie met een bestuurder en is er altijd minstens één bestuurder gelinkt aan een kaart. Een ander voorbeeld is een aanvraag waarbij altijd een bestuurder hoort. Voor alle informatie over de relaties raadpleeg het domeinmodel in het hoofdstuk analyse. Hieronder ziet u alle entiteiten gedefinieerd in klassen.



Figuur 13: Klassen binnen Domain

3.2 Data Access Layer

De Data Access Layer zal een MsSql-database aanspreken om gegevens uit te lezen. Dit zal op 2 verschillende manieren gebeuren (Dapper en Entity Framework Core), omdat de technische vereisten dit opleggen. De gegevens voor de werknemer front-end worden opgehaald met Dapper en het ophalen/wijzigen van gegevens voor de fleetmanager front-end met Entity Framework Core.

3.2.1 Database

De databank voor deze opdracht is een MsSql-database, zoals ik al eerder vermeldde bij de technische vereisten. Om een MsSql-server lokaal te draaien maak ik gebruik van SQL Server 2019 Developer Edition en SQL Server Management Studio.

SQL Server 2019 Developer is een volledig functionele gratis editie, die wordt gelicentieerd voor gebruik als ontwikkel- en testdatabase in een niet-productieomgeving.

SQL Server Management Studio (SSMS) is een geïntegreerde omgeving voor het beheren van elke SQL-infrastructuur, van SQL Server tot Azure SQL Database. SSMS biedt tools voor het configureren, bewaken en beheren van SQL Server-instances en databases.

3.2.2 Write Data Access Layer

Zoals eerder vermeld wordt Entity Framework Core gebruikt voor het ophalen en wijzigen van data door de fleetmanager. Wanneer een werknemer een aanvraag doet wordt dit ook gepersisteerd via deze laag.

Entity Framework Core is een open-source object-relational mapping framework. Binnen het project maak ik gebruik van EF Core 5.0 om de tabellen te genereren (ook wel bekend als code-first). Entity Framework Core wordt ook gebruikt om de database aan te spreken en het resultaat te mappen naar mijn C# klassen.

In de DbContext definieer ik de DbSets (tabellen) en de configurations (constraints en seed data). Hieronder ziet u de DbContext voor de Fleet Management App. De DbSets gaan op volgende afbeelding verder.

```
public class FleetManagementContext : DbContext
{
    0 references
    public FleetManagementContext(DbContextOptions<FleetManagementContext> options) : base(options)
    {
    }

    // DriverEntities
    0 references
    public DbSet<Address> Addresses { get; set; }
    0 references
    public DbSet<CardHolder> CardHolders { get; set; }
    0 references
    public DbSet<Driver> Drivers { get; set; }
    0 references
    public DbSet<DriverVehicleHistory> DriverVehicleHistories { get; set; }

    // FileEntities
    0 references
    public DbSet<File> Files { get; set; }
    0 references
    public DbSet<FileContent> FileContents { get; set; }

    // FuelCardEntities
    3 references
    public DbSet<FuelCard> FuelCards { get; set; }
    0 references
    public DbSet<FuelCardFuelType> FuelCardFuelTypes { get; set; }
    0 references
    public DbSet<FuelCardManufacturer> FuelCardManufacturers { get; set; }
    0 references
    public DbSet<FuelCardServiceType> FuelCardServiceTypes { get; set; }
    0 references
    public DbSet<FuelType> FuelTypes { get; set; }
    0 references
    public DbSet<ServiceType> ServiceTypes { get; set; }

    // MaintenanceEntities
    0 references
    public DbSet<Maintenance> Maintenances { get; set; }
    0 references
    public DbSet<MaintenanceFile> MaintenanceFiles { get; set; }

    // RepairEntities
    0 references
    public DbSet<InsuranceCompany> InsuranceCompanies { get; set; }
    0 references
    public DbSet<Repair> Repairs { get; set; }
    0 references
    public DbSet<RepairDocument> RepairDocuments { get; set; }
    0 references
    public DbSet<RepairPhoto> RepairPhotos { get; set; }

    // RequestEntities
    2 references
    public DbSet<Request> Requests { get; set; }
    0 references
    public DbSet<FuelCardRequest> FuelCardRequests { get; set; }
    0 references
    public DbSet<MaintenanceRequest> MaintenanceRequests { get; set; }
    0 references
    public DbSet<RepairRequest> RepairRequests { get; set; }
    0 references
    public DbSet<VehicleRequest> VehicleRequests { get; set; }
```

Hier ziet u de overige DbSets en welke configurations op de DbContext worden toegepast.

```
// VehicleEntities
0 references
public DbSet<LicensePlate> LicensePlates { get; set; }
0 references
public DbSet<Mileage> Mileages { get; set; }
4 references
public DbSet<Vehicle> Vehicles { get; set; }
1 reference
public DbSet<VehicleModel> VehicleModels { get; set; }
0 references
public DbSet<VehicleManufacturer> VehicleManufacturers { get; set; }

0 references
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlServer("Data Source=localhost;Initial Catalog=FleetManagement;I
}
```

```
0 references
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new AddressConfiguration());
    modelBuilder.ApplyConfiguration(new CardHolderConfiguration());
    modelBuilder.ApplyConfiguration(new DriverConfiguration());
    modelBuilder.ApplyConfiguration(new DriverVehicleHistoryConfiguration());
    modelBuilder.ApplyConfiguration(new FileConfiguration());
    modelBuilder.ApplyConfiguration(new FileContentConfiguration());
    modelBuilder.ApplyConfiguration(new FuelCardConfiguration());
    modelBuilder.ApplyConfiguration(new FuelCardFuelTypeConfiguration());
    modelBuilder.ApplyConfiguration(new FuelCardManufacturerConfiguration());
    modelBuilder.ApplyConfiguration(new FuelCardServiceTypeConfiguration());
    modelBuilder.ApplyConfiguration(new FuelTypeConfiguration());
    modelBuilder.ApplyConfiguration(new ServiceTypeConfiguration());
    modelBuilder.ApplyConfiguration(new MaintenanceConfiguration());
    modelBuilder.ApplyConfiguration(new MaintenanceFileConfiguration());
    modelBuilder.ApplyConfiguration(new InsuranceCompanyConfiguration());
    modelBuilder.ApplyConfiguration(new RepairDocumentConfiguration());
    modelBuilder.ApplyConfiguration(new RepairPhotoConfiguration());
    modelBuilder.ApplyConfiguration(new RepairConfiguration());
    modelBuilder.ApplyConfiguration(new RequestConfiguration());
    modelBuilder.ApplyConfiguration(new FuelCardRequestConfiguration());
    modelBuilder.ApplyConfiguration(new MaintenanceRequestConfiguration());
    modelBuilder.ApplyConfiguration(new RepairRequestConfiguration());
    modelBuilder.ApplyConfiguration(new LicensePlateConfiguration());
    modelBuilder.ApplyConfiguration(new MileageConfiguration());
    modelBuilder.ApplyConfiguration(new VehicleConfiguration());
    modelBuilder.ApplyConfiguration(new VehicleManufacturerConfiguration());
    modelBuilder.ApplyConfiguration(new VehicleModelConfiguration());
}
```

Figuur 14: FleetManagementContext

Hieronder ziet u een voorbeeld van een configuration. Een aparte configuration-klasse laat toe om de constraints hier te definiëren en seed data toe te voegen.

```
public class VehicleConfiguration : IEntityTypeConfiguration<Vehicle>
{
    0 references
    public void Configure(EntityTypeBuilder<Vehicle> builder)
    {
        builder
            .HasKey(v => v.VehicleId);
        builder
            .Property(v => v.VehicleType).HasConversion<string>();
        builder
            .HasOne(v => v.Driver)
            .WithOne(d => d.Vehicle)
            .HasForeignKey<Driver>(d => d.VehicleId);
        builder
            .HasMany(v => v.DriverHistory)
            .WithOne(dvh => dvh.Vehicle)
            .OnDelete(DeleteBehavior.Restrict);

        builder
            .HasIndex(x => x.Vin)
            .IsUnique();

        builder
            .HasData(
                //polestar 2 single motor
                new
                {
                    VehicleId = 1,
                    Vin = "2HNYD18821H068766",
                    CurrentLicensePlate = "1-ABC-001",
                    FuelTypeId = 3,
                    VehicleType = VehicleType.Sedan,
                    LastMileageRecorded = 300,
                    IsActive = true,
                    ProductionYear = "2022",
                    VehicleModelId = 8
                },
                new
                {
                    VehicleId = 18,
                    Vin = "2HNYD18821H068749",
                    CurrentLicensePlate = "1-ABC-018",
                    FuelTypeId = 3,
                    VehicleType = VehicleType.Sedan,
                    LastMileageRecorded = 455,
                    IsActive = true,
                    ProductionYear = "2022",
                    VehicleModelId = 8
                },
                new
                {
                    VehicleId = 19,
                    Vin = "2HNYD18821H068748",
                    CurrentLicensePlate = "1-ABC-019",
                    FuelTypeId = 3,
                    VehicleType = VehicleType.Sedan,
                    LastMileageRecorded = 3300,
                    IsActive = true,
                    ProductionYear = "2022",
                    VehicleModelId = 8
                },
```

Figuur 15: VehicleConfiguration

Voor minder dubbele code maak ik gebruik van een generic repository. De generic repository is ontwikkeld om de standaard CRUD-operaties één keer te schrijven en deze dan te implementeren in andere (specifiekere) repository's. Mijn generic repository is een abstract class die de interface IAsyncRepository implementeert.

```
public abstract class BaseRepository<T> : IAsyncRepository<T> where T : class
{
    protected readonly FleetManagementContext _context;

    7 references
    protected BaseRepository(FleetManagementContext context)
    {
        _context = context;
    }

    4 references
    public async Task<T> AddAsync(T entity)
    {
        await _context.Set<T>().AddAsync(entity);
        await _context.SaveChangesAsync();
        return entity;
    }

    1 reference
    public async Task DeleteAsync(T entity)
    {
        _context.Set<T>().Remove(entity);
        await _context.SaveChangesAsync();
    }

    37 references
    public async Task<T> GetByIdAsync(int id)
    {
        return await _context.Set<T>().FindAsync(id);
    }

    4 references
    public async Task<IEnumerable<T>> ListAllAsync()
    {
        return await _context.Set<T>().ToListAsync();
    }

    11 references
    public async Task UpdateAsync(T entity)
    {
        _context.Entry(entity).State = EntityState.Modified;
        await _context.SaveChangesAsync();
    }
}
```

Figuur 16: BaseRepository

Door middel van dependency injection worden deze repository's geregistreerd en beschikbaar gesteld voor de write- en admin-API (meer informatie over de API's komt in het hoofdstuk van Presentation Layer). Om deze methode op te roepen en zo de dependency's binnen te halen is een instantie van IConfiguration nodig, omdat hierin de connectionstring van mijn MsSql-database staat.

```
namespace DAL.Write
{
    0 references
    public static class DependencyInjection
    {
        2 references
        public static IServiceCollection AddInfrastructure(this IServiceCollection services, IConfiguration configuration)
        {
            services.AddDbContext<FleetManagementContext>(options =>
                options.UseSqlServer(
                    configuration.GetConnectionString("DefaultConnection"),
                    options => options.MigrationsAssembly(typeof(FleetManagementContext).Assembly.FullName)));

            services.AddScoped<IRequestRepository, RequestRepository>();
            services.AddScoped<IDriverRepository, DriverRepository>();
            services.AddScoped<IVehicleRepository, VehicleRepository>();
            services.AddScoped<IFuelCardRepository, FuelCardRepository>();
            services.AddScoped<IMaintenanceRepository, MaintenanceRepository>();
            services.AddScoped<IRepairRepository, RepairRepository>();
            services.AddScoped<IVehicleModelRepository, VehicleModelRepository>();

            return services;
        }
    }
}
```

Figuur 17: DependencyInjection van de Write Data Access Layer

3.2.3 Read Data Access Layer

De gegevens die worden opgehaald voor de werknemer front-end zijn read-only en worden door middel van Dapper direct gemapt naar een gelijkaardig readmodel. Een entiteit mag nooit naar buiten gestuurd worden zoals het in de databank wordt bijgehouden, omdat dit informatie vrijgeeft over hoe de applicatie werkt. Hieronder ziet u een repository die werkt met behulp van Dapper.

```
public class VehicleRepository : IVehicleRepository
{
    private readonly IConfiguration configuration;

    0 references
    public VehicleRepository(IConfiguration configuration)
    {
        this.configuration = configuration;
    }

    3 references
    public async Task<IReadOnlyList<VehicleReadModel>> GetAllAsync()
    {
        var sql = "SELECT * FROM Vehicles";
        using (var connection = new SqlConnection(configuration.GetConnectionString("DefaultConnection")))
        {
            connection.Open();
            var result = await connection.QueryAsync<VehicleReadModel>(sql);
            return result.ToList();
        }
    }

    3 references
    public async Task<VehicleReadModel> GetAsync(int id)
    {
        var sql = @"SELECT * FROM Vehicles WHERE VehicleId = @Id";
        using (var connection = new SqlConnection(configuration.GetConnectionString("DefaultConnection")))
        {
            connection.Open();
            var result = await connection.QuerySingleOrDefaultAsync<VehicleReadModel>(sql, new { Id = id });
            return result;
        }
    }

    2 references
    public async Task<VehicleReadModel> GetVehicleReadModelByDriverId(int id)
    {
        var sql = @"SELECT v.VehicleId, v.vin, v.CurrentLicensePlate, v.LastMileageRecorded,
            v.ProductionYear, vm.Name VehicleModelName, ft.FuelName, vm.ImgName
            FROM Vehicles v
            inner join VehicleModels vm on v.VehicleModelId = vm.VehicleModelId
            inner join FuelTypes ft on v.FuelTypeId = ft.FuelTypeId
            inner join Drivers d on d.VehicleId = v.VehicleId
            WHERE d.DriverId = @Id";
        using (var connection = new SqlConnection(configuration.GetConnectionString("DefaultConnection")))
        {
            connection.Open();
            var result = await connection.QueryAsync<VehicleReadModel>(sql, new { id = id });
            return result.Single();
        }
    }
```

Figuur 15: Dapper VehicleRepository

Ook voor de Read Data Access Layer is er een dependency injection klasse die alle repository's registreerd. Deze klasse wordt door de read-API aangeroepen zodat de API weet hoe gegevens kunnen opgehaald worden.

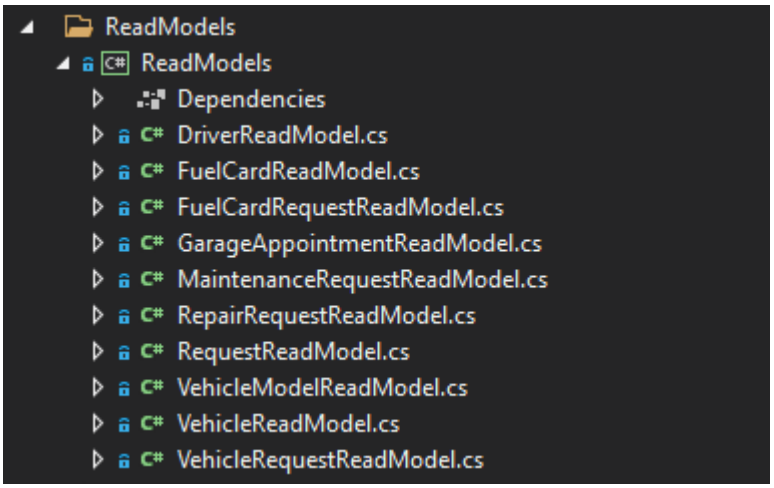
```
namespace DAL.Read
{
    0 references
    public static class ServiceRegistration
    {
        1 reference
        public static void AddInfrastructure(this IServiceCollection services)
        {
            services.AddScoped<IVehicleRepository, VehicleRepository>();
            services.AddScoped<IDriverRepository, DriverRepository>();
            services.AddScoped<IFuelCardRepository, FuelCardRepository>();
            services.AddScoped<IRequestRepository, RequestRepository>();
            services.AddScoped<IVehicleModelRepository, VehicleModelRepository>();
        }
    }
}
```

Figuur 17: ServiceRegistration Read Data Access Layer

3.3 Readmodels

In het vorige deel heb ik vermeld dat het niet slim is om de entiteiten die gebruikt worden om te persisteren in databank ook naar buiten te sturen (de gegevenswissel tussen back-end en front-end). Daarom heb ik read-only modellen gedefinieerd in een aparte class library. Het voordeel van deze in een aparte library te schrijven is dat later de Xaramin front-end voor de werknemer naar deze klassen kan verwijzen en moet ik ze niet nog eens schrijven voor de front-end. Omdat Xamarin cross-platform is moet de Readmodels library net standaard 2 zijn in plaats van .net 5.

Hieronder ziet u welke read-modellen benut worden door de werknemer front-end.



Figuur 18: Klassen binnen ReadModels

Belangrijk read-only betekent dat de instantie niet aangepast mag worden, daarom moet de setter oftewel private zijn, oftewel init (property kan enkel bij initiatie aangepast worden)!

```
public class DriverReadModel
{
    12 references
    public int DriverId { get; private set; }
    1 reference
    public string FirstName { get; private set; }
    1 reference
    public string LastName { get; private set; }
    1 reference
    public DateTime BirthDate { get; private set; }
    1 reference
    public string NationalRegistryNumber { get; private set; }
    1 reference
    public string DriverLicense { get; private set; }
    1 reference
    public int VehicleId { get; private set; }
```

Figuur 19: DriverReadModel

3.4 Business Logic Layer

De Business Logic Layer dient als intermediair voor gegevensuitwisseling tussen de Presentation Layer en de Data Access Layer. Deze laag behandelt de bedrijfsregels, berekeningen en logica binnen een applicatie die dicteren hoe deze zich gedraagt. In mijn Business Logic Layer wordt voor de uitwisseling van gegevens gebruik gemaakt van mapping (manueel & automapper), commands (m.b.v. MediatR) die als doel hebben gegevens toe te voegen of aan te passen en validators (m.b.v. FluentValidation) die controleren of binnenkomende gegevens correct zijn. De uitvoerende componenten in deze laag eindigen met -Bloc, deze afkorting heb ik tijdens mijn onderzoek gevonden en betekent Business Logic Component.

3.4.1 Mappings

Mapping gebeurt wanneer informatie tussen twee lagen wordt uitgewisseld. Gegevens die van de front-end komen en in de databank moeten worden opgeslagen worden door de package AutoMapper omgezet naar het domeinmodel dat de Data Access Layer kent. Wanneer de fleetmanager informatie opvraagt wordt deze eerst gemapt naar een DTO (data transfer object). Voor de werknemer front-end gebeurt deze mapping al in de Read Data Access Layer, omdat Dapper hier al mapt naar een readmodel (zie hoofdstuk Read Data Acces Layer -> Dapper).

Hieronder ziet u de MappingProfile die AutoMapper gebruikt om binnenkomende aanvragen van de werknemer front-end te mappen naar het model dat de databank gebruikt om aanvragen te persisteren.

```
public class MappingProfile : Profile
{
    0 references
    public MappingProfile()
    {
        CreateMap<CreateMaintenanceRequestCommand, MaintenanceRequest>()
            .ForMember(dest => dest.DateA, opt => opt.MapFrom(src => src.PreferredDate))
            .ForMember(dest => dest.DateB, opt => opt.MapFrom(src => src.OtherPreferredDate));

        CreateMap<CreateRepairRequestCommand, RepairRequest>()
            .ForMember(dest => dest.DateA, opt => opt.MapFrom(src => src.PreferredDate))
            .ForMember(dest => dest.DateB, opt => opt.MapFrom(src => src.OtherPreferredDate))
            .ForMember(dest => dest.Description, opt => opt.MapFrom(src => src.DescriptionOfDamage));
    }
}
```

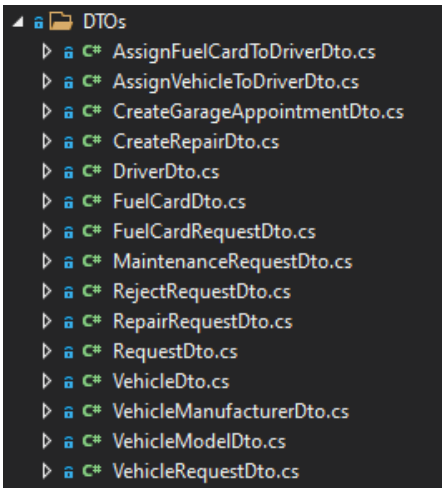
Figuur 20: MappingProfile

Wanneer informatie wordt opgevraagd voor de fleetmanager front-end gebeurt deze mapping manueel. Hiervoor heb ik vier static hulp-klassen die aangeroepen kunnen worden door de BLOCs (Business Logic Component). Op de volgende pagina ziet u een voorbeeld van zo'n zelf geschreven mapper.

```
public static class FuelCardDtoMapper
{
    5 references
    public static FuelCardDto MapToFuelCardDto(FuelCard fuelCard)
    {
        List<string> fuelTypes = new List<string>();
        foreach (FuelCardFuelType fuelCardFuelType in fuelCard.FuelCardFuelTypes)
        {
            fuelTypes.Add(fuelCardFuelType.FuelType.FuelName);
        }

        List<string> serviceTypes = new List<string>();
        foreach (FuelCardServiceType fuelCardServiceType in fuelCard.FuelCardServiceTypes)
        {
            serviceTypes.Add(fuelCardServiceType.ServiceType.ServiceName);
        }

        if (fuelCard.DriverId == null)
        {
            fuelCard.DriverId = 0;
        }

        FuelCardDto fuelCardDto = new FuelCardDto
        {
            FuelCardId = fuelCard.FuelCardId,
            Name = fuelCard.Name,
            CardNumber = fuelCard.CardNumber,
            ExpirationDate = fuelCard.ExpirationDate,
            AuthenticationType = fuelCard.AuthenticationType.ToString(),
            IsBlocked = fuelCard.IsBlocked,
            DriverId = (int)fuelCard.DriverId,
            FuelTypes = fuelTypes,
            ServiceTypes = serviceTypes
        };

        return fuelCardDto;
    }
}
```

Figuur 21: FuelCardDtoMapper

De modellen voor de DTO's bevinden zich ook in de Business Logic Layer. Enkel de DTO's die eindigen op -Request worden automatisch gemapt door AutoMapper. Voor de andere gebeurt een manuele mapping.



Figuur 22: DTO's binnen de Business Logic Layer

3.4.2 Commands

In de Business Logic Layer bevinden zich ook commands die ontworpen zijn om CUD-operaties (Create, Update, Delete) aan te duiden. Dit is volgens het CQRS-principe dat ook in de technische vereisten wordt vermeld. Het nut hiervan is dat de databank vele performanter gegevens kan toevoegen/aanpassen/verwijderen. Terwijl kan de read kant geoptimaliseerd worden door een projectie van de databank te gebruiken en hierop andere indexen te leggen.

Een command komt binnen in de Write-API en wordt door middel van MediatR doorgestuurd naar de Business Logic Layer waar deze command wordt gevalideerd door de bijhorende validator en wordt uitgevoerd door de handler als de validatie in orde is.

Hieronder ziet u een voorbeeld van een command. Net zoals read-modellen kunnen commands niet meer aangepast worden nadat ze zijn gecreëerd. Hiervoor gebruik ik als setter init.

```
public class CreateMaintenanceRequestCommand : IRequest<bool>
{
    3 references
    public DateTime PreferredDate { get; init; }
    3 references
    public DateTime OtherPreferredDate { get; init; }
    2 references
    public int VehicleId { get; init; }
    2 references
    public int DriverId { get; init; }
```

Figuur 23: CreateMaintenanceRequestCommand

Commands worden gevalideerd door een specifieke validator, deze wordt opgeroepen door een Business Logic Component vooraleer deze wijzigingen gaat doorvoeren naar de databank. Een CommandValidator erft over van AbstractValidator, die wordt aangeboden in de FluentValidation NuGet package. Een voorbeeld van validatie die moet gebeuren is wanneer een aanvraag wordt ingediend en het systeem controleert of de gebruiker wel bestaat. Hieronder ziet u de validator voor de aanvraag van een onderhoudsbeurt.

```
public class CreateMaintenanceRequestCommandValidator : AbstractValidator<CreateMaintenanceRequestCommand>
{
    private readonly IVehicleRepository vehicleRepository;
    private readonly IDriverRepository driverRepository;
    1 reference
    public CreateMaintenanceRequestCommandValidator(IVehicleRepository vehicleRepository, IDriverRepository driverRepository)
    {
        this.vehicleRepository = vehicleRepository;
        this.driverRepository = driverRepository;

        PreferredDateIsAfterOtherPreferredDate();
        OtherPreferredDateIsBeforePreferredDate();
        DriverMustExist();
        VehicleMustExist();
    }

    1 reference
    private void VehicleMustExist()
    {
        RuleFor(model => model.VehicleId)
            .NotEmpty()
            .MustAsync(VehicleExists);
    }
```

Figuur 24: CreateMaintenanceRequestCommandValidator

Vervolgens is er dus de handler die dankzij MediatR wordt opgeroepen wanneer in de Write-API een command binnen komt en wordt doorgestuurd. Het nut hiervan is dat er zo weinig mogelijk logica gebeurt in een controller van de Write-API. Ook in de handler is er geen logica aanwezig, de handler stuurt enkel de command door naar de Business Logic Component en zal uiteindelijk het resultaat van de actie terugkoppelen naar de controller.

Hieronder ziet u de handler voor het aanmaken van een aanvraag voor een ondehoudsbeurt.

```
public class CreateMaintenanceRequestHandler : IRequestHandler<CreateMaintenanceRequestCommand, bool>
{
    private readonly ICreateMaintenanceRequestBloc createMaintenanceRequestBloc;

    0 references
    public CreateMaintenanceRequestHandler(ICreateMaintenanceRequestBloc createMaintenanceRequestBloc)
    {
        this.createMaintenanceRequestBloc = createMaintenanceRequestBloc;
    }

    0 references
    public async Task<bool> Handle(CreateMaintenanceRequestCommand request, CancellationToken cancellationToken)
    {
        var entity = await createMaintenanceRequestBloc.CreateMaintenanceRequest(request);

        if (entity != null)
        {
            return true;
        } else
        {
            return false;
        }
    }
}
```

Figuur 25: CreateMaintenanceRequestHandler

3.4.3 BLOCs (Business Logic Components)

De BLOCs dienen als uitvoerende componenten die eerst gaan controleren of de verandering mag uitgevoerd worden a.d.h.v. een validator. Indien de validator aangeeft dat de informatie correct is, zal de BLOC een repository uit de Write Data Access Layer aanspreken om de wijzigingen door te voeren naar de databank.

De BLOCs dienen ook als doorgeefluik van gegevens voor de fleetmanager front-end. Vooraleer de gegevens naar de Presentation Layer gaan worden ze eerst gemapt, zoals beschreven in het hoofdstuk mapping.

Op de volgende pagina ziet u de BLOC die een aanvraag tot onderhoud van status zal veranderen en naargelang welke status (goedgekeurd of afgekeurd) een onderhoudsbeurt zal aanmaken en opslagen, oftewel bij de aanvraag een reden zal toevoegen waarom deze is afgewezen.

```
public class MaintenanceBloc : IMaintenanceBloc
{
    private readonly IMaintenanceRepository maintenanceRepository;
    private readonly IVehicleRepository vehicleRepository;
    private readonly IRequestRepository requestRepository;
    private readonly CreateGarageAppointmentValidator validationRules;

    0 references
    public MaintenanceBloc(
        IMaintenanceRepository maintenanceRepository,
        IVehicleRepository vehicleRepository,
        IRequestRepository requestRepository)
    {
        this.maintenanceRepository = maintenanceRepository;
        this.vehicleRepository = vehicleRepository;
        this.requestRepository = requestRepository;
        validationRules = new CreateGarageAppointmentValidator(requestRepository, vehicleRepository);
    }

    2 references
    public async Task<Maintenance> CreateMaintenance(CreateGarageAppointmentDto createGarageAppointmentDto)
    {
        ValidationResult validationResult = validationRules.Validate(createGarageAppointmentDto);
        if (!validationResult.IsValid) return null;

        Maintenance maintenance = await InstantiateMaintenance(createGarageAppointmentDto);
        Maintenance result = await PersistMaintenance(createGarageAppointmentDto, maintenance);
        return result;
    }

    1 reference
    private async Task<Maintenance> PersistMaintenance(CreateGarageAppointmentDto createGarageAppointmentDto, Maintenance maintenance)
    {
        var result = await maintenanceRepository.AddAsync(maintenance);
        if (result != null)
        {
            var request = await requestRepository.GetByIdAsync(createGarageAppointmentDto.RequestId);
            request.RequestStatus = RequestStatus.Accepted;
            await requestRepository.UpdateAsync(request);
        }

        return result;
    }

    2 references
    public async Task<MaintenanceRequest> RejectMaintenanceRequest(RejectRequestDto rejectRequestDto)
    {
        MaintenanceRequest maintenanceRequest = (MaintenanceRequest)await requestRepository.GetByIdAsync(rejectRequestDto.RequestId);
        if (maintenanceRequest != null)
        {
            maintenanceRequest.Feedback = rejectRequestDto.Feedback;
            maintenanceRequest.RequestStatus = RequestStatus.Rejected;
            await requestRepository.UpdateAsync(maintenanceRequest);
        }
        return maintenanceRequest;
    }
```

Figuur 26: MaintenanceBloc

3.5 Presentation Layer

De Presentation Layer van de back-end bestaat uit 3 API's. De Read-API die informatie ophaalt voor de werknemer front-end. De Write-API waar commands binnen komen die ook vanaf deze front-end verstuurd worden. Tot slot is er de Admin-API die zowel informatie ophaalt als wijzigingen doorgeeft aan de Business Logic Layer. Deze API dient enkel voor de fleetmanager front-end.

3.5.1 Read-API

De Read-API is ontworpen zodat werknemers informatie kunnen opvragen over: hun auto, tankkaart, recente aanvragen en de autocatalogus. In een controller komt een query binnen. Deze query wordt door MediatR verzonden en vervolgens uitgevoerd door een handler die zich ook in deze laag bevindt. Hieronder ziet u de controller die informatie over aanvragen zal doorsturen.

```
[Route("api/[controller]")]
[ApiController]
1 reference
public class RequestController : ControllerBase
{
    private readonly IMediator mediator;

    0 references
    public RequestController(IMediator mediator)
    {
        this.mediator = mediator;
    }

    [HttpGet("RequestReadModels")]
    0 references
    public async Task<IActionResult> GetRequestReadModels([FromQuery]GetRequestReadModelsQuery getRequestReadModelsQuery)
    {
        var response = await mediator.Send(getRequestReadModelsQuery);
        return response == null ? NotFound() : Ok(response);
    }

    [HttpGet("GarageAppointmentReadModel")]
    0 references
    public async Task<IActionResult> GetGarageAppointmentReadModel([FromQuery] GetGarageAppointmentReadModelQuery getGarageAppointmentReadModelQuery)
    {
        var response = await mediator.Send(getGarageAppointmentReadModelQuery);
        return response == null ? NotFound() : Ok(response);
    }
```

Figuur 27: RequestController

De parameters die binnenkomen zijn voor het systeem gekend als query's. Deze worden gedefinieerd als onwijzigbare instanties net zoals de commands en read-modellen. Op de volgende afbeelding ziet u de query voor recente aanvragen van de werknemer op te halen.

```
namespace API.Read.Features.Request.Queries
{
    3 references
    public record GetRequestReadModelsQuery(int id) : IRequest<List<RequestReadModel>>;
}
```

Figuur 28: GetRequestReadModelsQuery

3.5.2 Write-API

De Write-API wordt door de werknemer front-end aangesproken om specifieke aanvragen te creëren. Zo zijn er endpoints voor het aanmaken van een aanvraag voor een onderhoud, voor een herstelling, voor een tankkaart en voor een voertuig. De informatie die binnenkomt is binnen het systeem gekend als een command. De werknemer front-end heeft ook kennis van deze klassen om een command te kunnen versturen. Hieronder ziet u de vier endpoints voor aanvragen te creëren.

```
[Route("api/[controller]")]
[ApiController]
1 reference
public class RequestController : ControllerBase
{
    private readonly IMediator mediator;

    0 references
    public RequestController(IMediator mediator)
    {
        this.mediator = mediator;
    }

    [Route("CreateMaintenanceRequest")]
    [HttpPost]
    0 references
    public async Task<IActionResult> CreateMaintenanceRequest([FromBody] CreateMaintenanceRequestCommand createMaintenanceRequestCommand)
    {
        var result = await mediator.Send(createMaintenanceRequestCommand);
        return Ok(result);
    }

    [Route("CreateRepairRequest")]
    [HttpPost]
    0 references
    public async Task<IActionResult> CreateRepairRequest([FromBody] CreateRepairRequestCommand createRepairRequestCommand)
    {
        var result = await mediator.Send(createRepairRequestCommand);
        return Ok(result);
    }

    [Route("CreateFuelCardRequest")]
    [HttpPost]
    0 references
    public async Task<IActionResult> CreateFuelCardRequest([FromBody] CreateFuelCardRequestCommand createFuelCardRequestCommand)
    {
        var result = await mediator.Send(createFuelCardRequestCommand);
        return Ok(result);
    }

    [Route("CreateVehicleRequest")]
    [HttpPost]
    0 references
    public async Task<IActionResult> CreateVehicleRequest([FromBody] CreateVehicleRequestCommand createVehicleRequestCommand)
    {
        var result = await mediator.Send(createVehicleRequestCommand);
        return Ok(result);
    }
}
```

Figuur 29: RequestController

3.5.3 Admin-API

De Admin-API beschikt over controllers om voertuigen, tankkaarten en aanvragen op te halen, maar ook om de aanvragen af te handelen. De Admin-API maakt geen gebruik van MediatR en dus zal de controller zelf de juiste BLOC uit de Business Logic Layer aanspreken. Op de volgende afbeelding ziet u hoe een controller zowel informatie ophaalt uit de Business Logic Layer, als het afhandelen van aanvragen doorgeeft.

```
[Route("api/[controller]")]
[ApiController]
1 reference
public class VehicleController : ControllerBase
{
    private readonly IVehicleBloc vehicleBloc;

    0 references
    public VehicleController(IVehicleBloc vehicleBloc)
    {
        this.vehicleBloc = vehicleBloc;
    }

    // GET: api/<VehicleController>
    [HttpGet]
    0 references
    public async Task<IEnumerable<VehicleDto>> GetAsync()
    {
        return await vehicleBloc.ListAllAsync();
    }

    // GET api/<VehicleController>/5
    [HttpGet("{id}")]
    0 references
    public async Task<VehicleDto> GetAsync(int id)
    {
        return await vehicleBloc.GetByIdAsync(id);
    }

    [HttpPost("Accept")]
    0 references
    public async Task<string> AssignVehicleToDriver([FromBody] AssignVehicleToDriverDto assignVehicleToDriverDto)
    {..
        var result = await vehicleBloc.AssignVehicleToDriver(assignVehicleToDriverDto);
        if (result != null) return "Voertuig succesvol toegewezen.";
        return "Er is iets fout gegaan.";
    }

    [HttpPost("Reject")]
    0 references
    public async Task<string> RejectVehicleRequest([FromBody] RejectRequestDto rejectRequestDto)
    {..
        var result = await vehicleBloc.RejectVehicleRequest(rejectRequestDto);
        if (result != null) return "Feedback toegevoegd.";
        return "Er is iets fout gegaan.";
    }
}
```

Figuur 30: VehicleController

4 FRONT-END

De technologie voor de front-ends mocht ik zelf kiezen. Ik heb hierbij voor de werknemer gekozen voor een mobiele app in Xamarin. Een mobiele app omdat een werknemer niet veel informatie zal opvragen en heel gemakkelijk een aanvraag moet kunnen doen. Xamarin omdat AllPhi gespecialiseerd is in Microsoft .NET. Voor de fleetmanager was ik nog aan het twijfelen over welk javascript-framework ik ging gebruiken. Uiteindelijk heb ik gekozen voor Reactjs, omdat AllPhi mij wist te vertellen dat Reactjs een tijdje populairder was als Angular.

4.1 Werknemer front-end

De Xamarin-app bestaat voornamelijk uit services die data ophalen en versturen. Dan zijn er natuurlijk ook de views en bijhorend de viewmodellen.

4.1.1 Services

Binnen de services zijn er twee soorten: services die data ophalen in de vorm van read-modellen en services die veranderingen aan data versturen als commands (zie hoofdstuk Write-API).

Hieronder ziet u een ReadModelService en een RequestService respectievelijk.

```
public class GarageAppointmentReadModelService : IGarageAppointmentReadModelService
{
    HttpClient client;
    JsonSerializerOptions options;

    1 reference
    public GarageAppointmentReadModelService()
    {
        client = new HttpClient();
        client.BaseAddress = new Uri("http://10.0.2.2:60280/api/");
        client.DefaultRequestHeaders.Add("Accept", "application/json");
        options = new JsonSerializerOptions { PropertyNamingPolicy = JsonNamingPolicy.CamelCase };
    }

    2 references
    public async Task<GarageAppointmentReadModel> GetItem(int requestId)
    {
        var query = HttpUtility.ParseQueryString(string.Empty);
        query["id"] = requestId.ToString();
        var response = await client.GetAsync($"Request/GarageAppointmentReadModel?{query}");
        if (response.IsSuccessStatusCode)
        {
            var responseAsString = await response.Content.ReadAsStringAsync();
            var result = JsonSerializer.Deserialize<GarageAppointmentReadModel>(responseAsString, options);
            return result;
        }
        return null;
    }
}
```

Figuur 31: GarageAppointmentReadModelService

```
public class FuelCardRequestService : IFuelCardRequestService
{
    HttpClient client;
    JsonSerializerOptions options;

    0 references
    public FuelCardRequestService()
    {
        client = new HttpClient();
        client.BaseAddress = new Uri("http://10.0.2.2:64144/api/");
        client.DefaultRequestHeaders.Add("Accept", "application/json");
        options = new JsonSerializerOptions { PropertyNamingPolicy = JsonNamingPolicy.CamelCase };
    }

    2 references
    public async Task<string> CreateFuelCardRequest(CreateFuelCardRequestCommand createFuelCardRequestCommand)
    {
        string json = JsonSerializer.Serialize(createFuelCardRequestCommand, options);
        var content = new StringContent(json, System.Text.Encoding.UTF8, "application/json");
        var uri = "Request/CreateFuelCardRequest";
        var response = await client.PostAsync(uri, content);
        if (!response.IsSuccessStatusCode)
        {
            throw new ApplicationException(response.ToString());
        }
        string stringResult = await response.Content.ReadAsStringAsync();
        return stringResult;
    }
}
```

Figuur 32: FuelCardRequestService

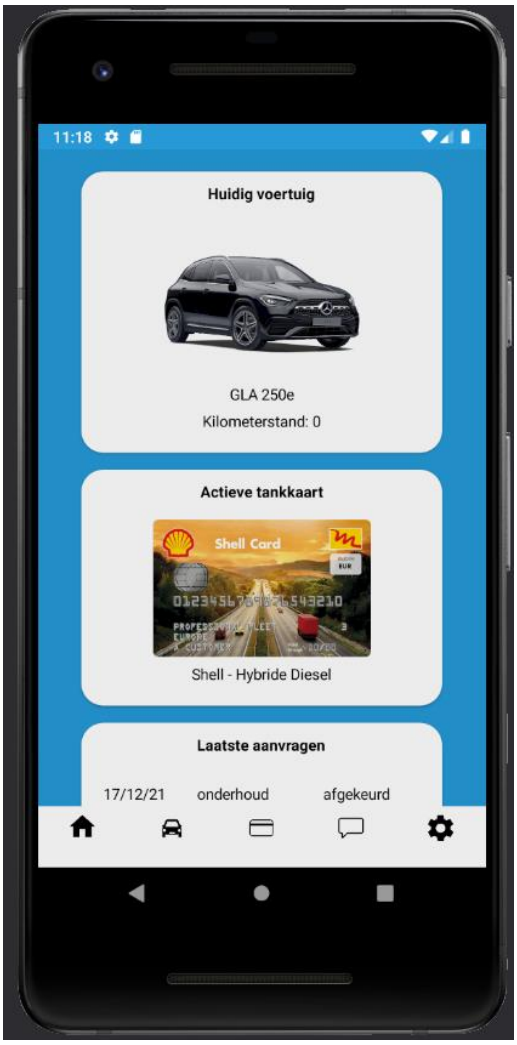
De services registreer ik als een dependency in de App klasse zodat ik ze met de DependencyService kan ophalen in mijn viewmodels.

```
6 references
public partial class App : Application
{

    2 references
    public App()
    {
        InitializeComponent();

        DependencyService.RegisterSingleton<INavigationService>(new NavigationService());

        DependencyService.RegisterSingleton<IFuelCardReadModelService>(new FuelCardReadModelService());
        DependencyService.RegisterSingleton<IVehicleReadModelService>(new VehicleReadModelService());
        DependencyService.RegisterSingleton<IRequestReadModelService>(new RequestReadModelService());
        DependencyService.RegisterSingleton<IDriverReadModelService>(new DriverReadModelService());
        DependencyService.RegisterSingleton<IVehicleModelReadModelService>(new VehicleModelReadModelService());
        DependencyService.RegisterSingleton<IGarageAppointmentReadModelService>(new GarageAppointmentReadModelService());

        DependencyService.Register<IMaintenanceRequestService, MaintenanceRequestService>();
        DependencyService.Register<IRepairRequestService, RepairRequestService>();
        DependencyService.Register<IFuelCardRequestService, FuelCardRequestService>();
        DependencyService.Register<IVehicleRequestService, VehicleRequestService>();

        MainPage = new AppShell();
    }
```

Figuur 33: App

4.1.2 Views

De views die ik ontworpen heb zijn: een inlogpagina, een landingspagina, een view waar een werknemer zijn/haar voertuig/tankkaart/aanvragen kan bekijken en een view om af te melden. Tenslotte is er nog een view voor wanneer een werknemer een onderhoud of herstelling aanvraagt en een view met de autocatalogus. Een view wordt beschreven in XAML. Op de volgende pagina ziet u de landingspagina en een deel van de XAML.



Figuur 34: LandingsPage in actie

```
<StackLayout Padding="40,0,40,10">
    <Frame HasShadow="True" CornerRadius="20">
        <StackLayout>
            <StackLayout IsVisible="{Binding HasFuelCard}">
                <Label
                    Text="Actieve tankkaart" FontAttributes="bold"
                    Style="{StaticResource CenterLabelColorBlack}" Margin="0,-10,0,10"
                    />
                <Image Source="{Binding FuelCardReadModel.ImgName}" WidthRequest="200"/>
                <Label
                    Text="{Binding FuelCardReadModel.Name}"
                    Style="{StaticResource CenterLabelColorBlack}"
                    />
            </StackLayout>
            <StackLayout IsVisible="{Binding HasFuelCard, Converter={converters:BooleanConverter}}">
                <Label
                    Text="Geen tankkaart om te tonen"
                    Style="{StaticResource CenterLabelColorBlack}"
                    />
            </StackLayout>
        </StackLayout>
    </Frame>
</StackLayout>
```

Figuur 35: een deel van LandingsPage.xaml

4.1.3 Viewmodels

Bij elke view in mijn Xamarin-applicatie hoort een viewmodel, enkel voor views met weinig informatie & acties is dit overbodig (in- en uitloggen bv.). Het model-view-viewmodel patroon heeft als voordeel dat de user interface (view) en de logica erachter (viewmodel) al ontworpen kunnen worden zonder afhankelijk te zijn van één van de twee. Verder nog moet het model niet aangepast worden bij het uitbreiden van de applicatie en kunnen model & viewmodel getest worden zonder view.

Hieronder ziet u het viewmodel voor de autocatalogus. DependencyService haalt de veschillende services binnen die op hun beurt alle voertuig modellen ophaalt en een aanvraag verstuurd wanneer een werknemer een voertuig kiest.

```
public class VehicleCatalogPageViewModel : BaseViewModel
{
    private readonly IVehicleModelReadModelService vehicleModelReadModelService = DependencyService.Resolve<IVehicleModelReadModelService>();
    private readonly IVehicleRequestService vehicleRequestService = DependencyService.Resolve<IVehicleRequestService>();
    private readonly IDriverReadModelService driverReadModelService = DependencyService.Resolve<IDriverReadModelService>();

    private List<VehicleModelReadModel> vehicleModelReadModels;

    1 reference
    public List<VehicleModelReadModel> VehicleModelReadModels
    {
        get { return vehicleModelReadModels; }
        set { SetProperty(ref vehicleModelReadModels, value); }
    }

    1 reference
    public ICommand GetVehicleModelReadModelsCommand { get; set; }
    1 reference
    public ICommand RequestVehicleModelCommand { get; set; }

    0 references
    public VehicleCatalogPageViewModel()
    {
        GetVehicleModelReadModelsCommand = new Command(() => GetVehicleModelReadModels());
        RequestVehicleModelCommand = new Command<int>((vehicleModelId) => RequestVehicleModel(vehicleModelId));
    }

    1 reference
    private async void RequestVehicleModel(int vehicleModelId)
    {
        var driver = driverReadModelService.GetCurrentDriver();
        CreateVehicleRequestCommand createVehicleRequestCommand = new CreateVehicleRequestCommand { DriverId = driver.DriverId, VehicleModelId = vehicleModelId };
        var result = await vehicleRequestService.CreateVehicleRequest(createVehicleRequestCommand);
        if (result == "true")
        {
            await Shell.Current.CurrentPage.DisplayAlert("Done!", "Aanvraag succesvol ingediend.", "ok");
        }else
        {
            await Shell.Current.CurrentPage.DisplayAlert("Mislukt!", "Er ging iets mis, probeer nog eens.", "ok");
        }
    }

    1 reference
    private async void GetVehicleModelReadModels()
    {
        var models = await vehicleModelReadModelService.GetItems();
        VehicleModelReadModels = models.OrderBy(x => x.VehicleManufacturerName).ToList();
    }
}
```

Figuur 36: VehicleCatalogPageViewModel

4.2 Fleetmanager front-end

Voor de fleetmanager front-end heb ik een React-applicatie geschreven (Reactjs). De app is gestructureerd in container components, presentation components en controlled components.

4.2.1 Container components

Ik heb een container components voor tankkaarten, voertuigen en aanvragen. Een container component is verantwoordelijk voor data op te halen en acties door te geven aan presentation components.

Hieronder ziet u de container component voor aanvragen op te halen en deze in een tabel te tonen (RequestList).

```
class RequestListContainer extends Component {
    constructor() {
        super()

        this.state = {
            requests: []
        };
    }

    componentDidMount() {
        axios.get(baseUrl).then(res => {
            res.data.forEach(element => {
                element.creationDate = this.sliceDate(element.creationDate);
                element.requestStatus = this.translateStatus(element.requestStatus);
                element.requestType = this.translateType(element.requestType);
            });
            console.log(res.data);
            this.setState({ requests: res.data })
        })
    }

    sliceDate(date) {···
    }

    translateStatus(status) {···
    }

    translateType(type) {···
    }

    render() {
        let style = {
            backgroundColor: "#2799D7",
        }
        return (
            <div style={style}>
                <div style={{margin: '30px 5% 30px 5%'}}>
                    <RequestList data={this.state.requests}/>
                </div>
            </div>
        );
    }
}
```

Figuur 37: RequestListContainer

4.2.2 Presentation components

Presentation components hebben enkel als functie data tonen en voor terugkoppeling zorgen wanneer data in deze component is gewijzigd.

Voorbeelden hiervan zijn de componenten die eindigen op -Card. Deze componenten hebben enkel als functie data te tonen in een card vorm zoals een card van Bootstrap.

```
const VehicleModelCard = (props) => {
    let imgStyle = {
        alignSelf: 'center',
        width: '200px'
    }

    console.log(props);

    return (
        <div className="card" style={{width: "18rem", height: "17rem", borderRadius: '15px', margin: '30px 30px 0px 0px'}}>

            <div className="card-body">
                <h5 className="card-title">Aangevraagd voertuig</h5>
                <img src={vehicleModels[props.item.vehicleModelDto.imgName]} alt="" style={imgStyle}/>
                <p className="card-text">{props.item.vehicleManufacturerDto.name}</p>
                <p className="card-text">{props.item.vehicleModelDto.name}</p>
            </div>
        </div>
    );
}
```

Figuur 38: VehicleModelCard

4.2.3 Controlled components

Controlled components beschikken zelf over een state om veranderingen aan een invoerveld in een form bij te houden. Door de invoervelden in een state te bewaren hoeft een controlled component enkel zijn state mee te geven aan een container component wanneer een form verstuurd mag worden.

Voor mijn applicatie heb ik dit toegepast bij elke form die nodig is, enkel bij het afwijzen van een aanvraag maak ik gebruik van een modal door Ant Design. Op de volgende pagina ziet u de controlled component voor een herstelling te plannen waarbij een planningsdatum en garage invoervelden zijn.

```
class RepairForm extends Component {
    constructor(props) {
        super(props);
        this.state = {
            preferredDate: this.props.item.preferredDate.split(' ')[0],
            otherPreferredDate: this.props.item.otherPreferredDate.split(' ')[0],
            date : this.props.item.preferredDate.split(' ')[0],
            time : '08:00',
            visible: false,
            garage: ''
        };

        this.handleDateChange = this.handleDateChange.bind(this);
        this.handleTimeChange = this.handleTimeChange.bind(this);
        this.handleGarageChange = this.handleGarageChange.bind(this);
        this.onCreateFeedback = this.onCreateFeedback.bind(this);
    }

    handleDateChange(event) {···
    }

    handleTimeChange(value) {···
    }

    handleGarageChange(e) {···
    }

    onCreateFeedback(values) {···
    }

    render() {
        let Button = styled.button`···
        `;

        let acceptButton;
        let rejectButton;
        if (this.props.item.requestStatus === 'In afwachting') {
            acceptButton = <Button onClick={() => this.props.handleConfirmRepair(this.state)}>Goedkeuren</Button>;
            rejectButton = <Button onClick={() => this.setState({visible:true})}>Afkeuren</Button>;
        }

        return (
            <div className="card" style={{width: "18rem", height: "17rem", borderRadius: '15px', margin: '30px 30px 0px 0px'}}>
                <div className="card-body">
                    <h5 className="card-title">Inspectie vastleggen</h5>
                    <Input placeholder="Garage" onChange={(e) => this.handleGarageChange(e)} style={{margin: '1em 0'}}/>
                    <select value={this.state.date} onChange={this.handleDateChange}>
                        <option value={this.state.preferredDate}>{this.state.preferredDate}</option>
                        <option value={this.state.otherPreferredDate}>{this.state.otherPreferredDate}</option>
                    </select>
                    <TimePicker format={'HH:mm'} value={moment(this.state.time, 'HH:mm')} onSelect={(value) => this.handleTimeChange(value)} style={{marginTop:'1
                    {acceptButton}
                    {rejectButton}
                    <RejectRequestForm
                        visible={this.state.visible}
                        onCreate={this.onCreateFeedback}
                        onCancel={() => {
                            this.setState({visible:false});
                        }}/>
                </div>
            </div>
        );
    }
```

Figuur 39: RepairForm