# The ReenWise Backend

TODO: A description of what the backend will do

* Collection sensor data through a separate process
  + data store
  + connecting to the ReenWise Backend through the Web API
* Collection/synching ABAX data
  + connecting to the ReenWise Backend through the Web API
* Offering a Web API
* The ReenWise database
* …

Hente oversikt fra Terjes dokument

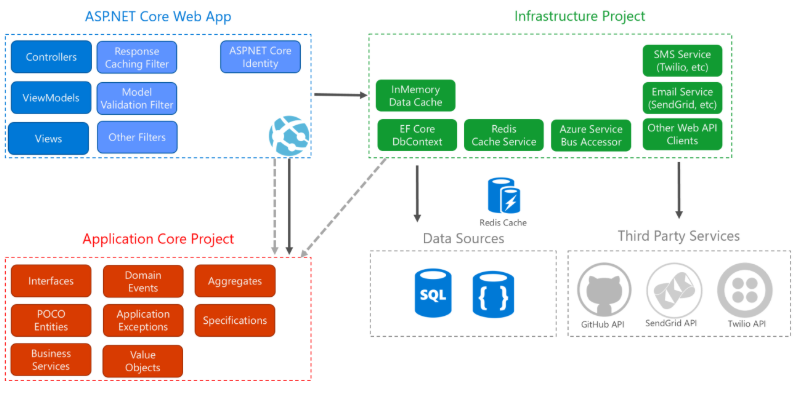
# ReenWise Backend Architecture

## Clean Architecture Template

The backend solution is build as a ASP.NET Core Web App offering a Web API where the following elements are included

* .NET Core Template Package
* ASP.NET Core 3.0
* Entity Framework Core 3.0
* (ASP.NET Core Identity 3.0)

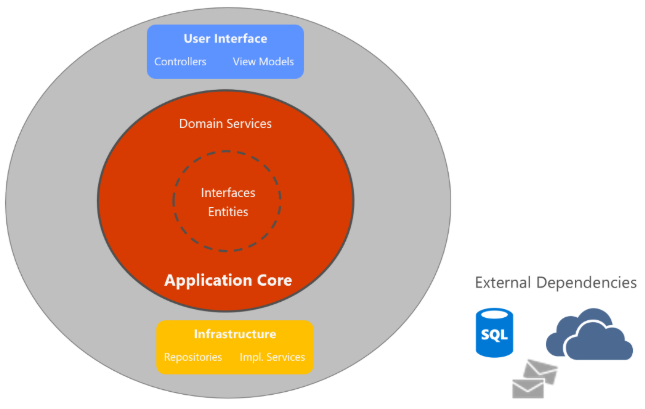
The figure below shows a more detailed view of an ASP.NET Core application's architecture when built following the Clean Architecture recommendations.



## Clean Architecture

The backend solution is biild following the principles of Clean Architecture - also referred to as Onion Architecture - as outlined in this article by Microsoft <https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures>. A Clean Architecture design pattern gives several benefits; to mention some

* Independence of frameworks
* Independent of UI
* Independent of database
* Independent of anything external
* A code that is easier to maintain and extend.



In this diagram, dependencies flow toward the innermost circle and illustrates that the Application Core has no dependencies on other application layers. The application's entities and interfaces are at the very centre. Just outside, but still in the Application Core, are domain services, which typically implement interfaces defined in the inner circle. Outside of the Application Core, both the UI and the Infrastructure layers depend on the Application Core, but not on one another (necessarily). Since the UI layer doesn't have any direct dependency on types defined in the Infrastructure layer, it's very easy to swap out implementations, either to facilitate testing or in response to changing application requirements. The same applies for all elements in the Infrastructure layer as e.g. database of external CRM or ERP systems. Clean Architecture is about organizing our code into layers with a very explicit rule governing how those layers may interact.

*”The overriding rule that makes this architecture work is The Dependency Rule. This rule says that source code dependencies can only point inwards. Nothing in an inner circle can know anything at all about something in an outer circle.”*

* Uncle Bob

## Application Core layer

The Application Core holds the business model and are typically divided into the innermost Domain entities layer and and the Application layer.

### Domain

This will contain all entities (enterprise model classes that are persisted), enums, exceptions, interfaces, types and logic specific to the domain layer.

### Application

This layer contains all application logic and Use Cases. It is dependent on the domain layer, but has no dependencies on any other layer or project. This layer defines interfaces that are implemented by outside layers. For example, if the application need to access a notification service, a new interface is added to Application and an implementation is created within Infrastructure. The Application layer also contains non-entity types that have no dependencies on UI/Web API or Infrastructure. These can be defined as simple Data Transfer Objects (DTOs).

## Infrastructure layer

This layer contains classes for accessing external resources such as data access, file systems, web services, mail services, and so on. These classes should be based on interfaces defined within the application layer. The Infrastructure layer also include the Entity Framework (EF) DbContext, any EF Core Migration objects that have been defined, and data access implementation classes. The data access abstraction implementation code is through the use of the Repository design pattern.

### Repository Pattern

Repository pattern provides an abstraction of data, so that the application can work with a simple abstraction that has an interface approximating that of a collection. Adding, removing, updating, and selecting items from this collection is done through a series of straightforward methods, without the need to deal with database concerns like connections, commands, cursors, or readers.

The code below defines a generic C# repository interface that ensures we have a common interface for working with any of our objects.

public interface IRepository<T> where T : EntityBase

{

T GetById(int id);

IEnumerable<T> List();

IEnumerable<T> List(Expression<Func<T, bool>> predicate);

void Add(T entity);

void Delete(T entity);

void Edit(T entity);

}

public abstract class EntityBase

{

public int Id { get; protected set; }

}

### Web API layer

This layer holds the *Web*, *UI* and *Presenter* concerns. In the context of our API, this means it accepts input in the form of http requests over the network (e.g., GET/POST/etc.) and returns its output as content formatted as JSON or XML.

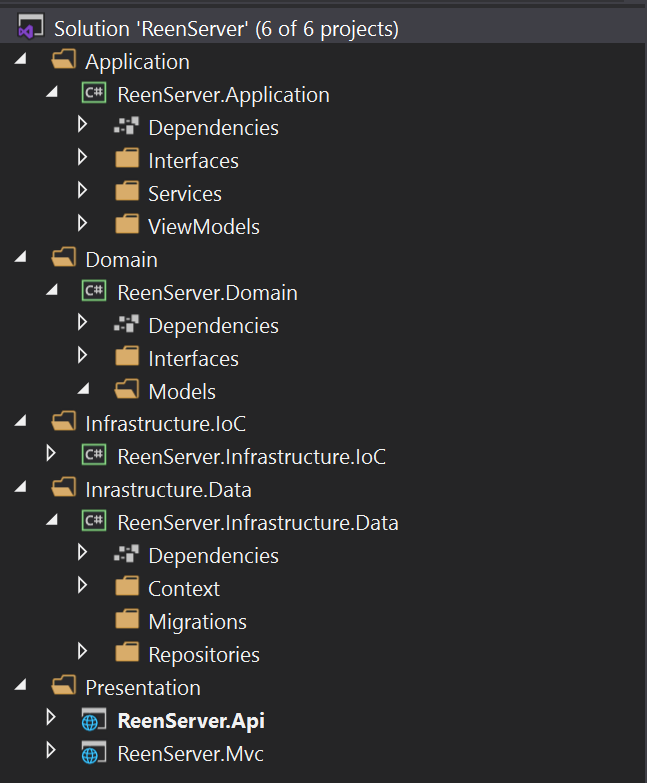
This layer will only reference the Application Core layer, and its types will interact with infrastructure strictly through interfaces defined in Application Core. No direct instantiation of or static calls to the Infrastructure layer types is allowed in this layer.

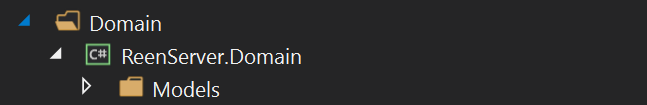
# Test Driven Development

Developing the backend solution will be based on a TDD approach by building out Use Cases "from the outside in" by defining them first and then using tests to iteratively implement only the bits of functionality required to pass them.

# The Project Organization

The backend solution shall be built as a ASP.NET Core Web Application offering a Web API written in C#. The project is recommended to be organized in a hierarchy, as shown below, that reflects the clean architecture as outlined above.

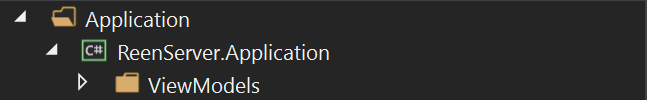




The Domain *Models* folder will hold the domain entities as e.g. the container entity which we name *Equipment*.

using ReenServer.Domain.Interfaces;  
using System;  
using System.Collections.Generic;  
using System.Text;

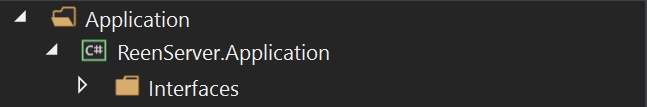
namespace ReenServer.Domain.Models  
{  
 public class Equipment  
 {  
 public Guid id { get; set; }  
 public string alias { get; set; }  
 public string model { get; set; }  
 public Guid locationId { get => locationId; set => locationId = value; }  
 public Guid companyId { get => companyId; set => companyId = value; }  
 }  
}



The Application *ViewModels* folder holds the viewmodels the presentation layer uses. A viewmodel maps the domains entities to the presentation layer; e.g. a viewmodel that returns a list of the *Domain* entity *Equipment* that can be presented in a view.

using ReenServer.Domain.Models;  
using System;  
using System.Collections.Generic;  
using System.Text;

namespace ReenServer.Application.ViewModels  
{  
 public class EquipmentViewModel  
 {  
 public IEnumerable<Equipment> Equipment { get; set; }  
 }  
}



The Application *Interfaces* folder holds all interfaces that is implemented by the Presentation layer. The interfaces (or services) defined here offers a ‘contract’ that allows the Presentation layer to perform CRUD operations; e.g. to create, read, update and delete. What the Presentation layer sees is the *ViewModels* - the Domain entities are hidden from the outer layers. In the code snippet below the interface defines the method *GetEquipement* that returns the Domain entities *Equipment* (i.e. all containers) mapped through the viewmodel *EquipmentViewModel*.

using ReenServer.Application.ViewModels;

using System;

using System.Collections.Generic;

using System.Text;

namespace ReenServer.Application.Interfaces

{

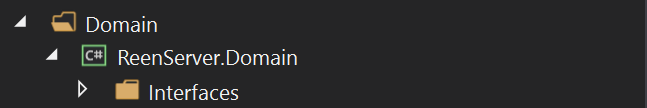
public interface IEquipmentService

{

EquipmentViewModel GetEquipment();

}

}



For the Application services to provide data to the ViewModels, data - the actual domain entities - has to be retrieved from the data store and that is obtained through a *Repository* interface. The *Repository* interfaces are implemented in the outer *Infrastructure* layer to hide the physical data store device from the inner *Application Core*. The code snippet below defines the interface method *GetEquipment* that retrieves all *Equipment* (i.e. containers) from the data store.

using ReenServer.Domain.Models;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Text;

namespace ReenServer.Domain.Interfaces

{

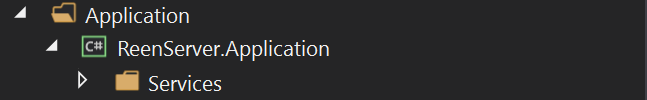
public interface IEquipmentRepository

{

public IEnumerable<Equipment> GetEquipment();

}

}



The Application Services folder holds the implementation of the Application service interfaces. The code snippet below implements the method *GetEquipment* that gets all containers (i.e. the Domain entities *Equipment*) from the data store. The reference to the repository (data store) from which the containers are retrieved, is inserted through *Dependency Injection* which is done at startup of the application.

using ReenServer.Application.Interfaces;

using ReenServer.Application.ViewModels;

using ReenServer.Domain.Interfaces;

using System;

using System.Collections.Generic;

using System.Text;

namespace ReenServer.Application.Services

{

class EquipmentService : IEquipmentService

{

private IEquipmentRepository \_equipmentRepository;

public EquipmentService(IEquipmentRepository equipmentRepository)

{

\_equipmentRepository = equipmentRepository;

}

public EquipmentViewModel GetEquipment()

{

return new EquipmentViewModel()

{

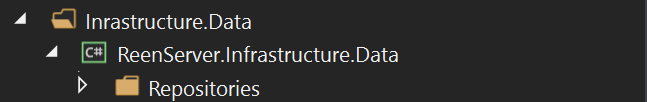
Equipment = \_equipmentRepository.GetEquipment()

};

}

}

}



This folder holds the implementation of the Domain Repository interfaces. The implementation of a repository interface will use *Dependency Injection* to access the data context (i.e. *ReenDbContext*). The code snippet below implements the *IEquipmentRepository* interface and gets all containers from the data store through the repository data context (*ReenDbContext*).

using ReenServer.Data.Context;

using ReenServer.Domain.Interfaces;

using ReenServer.Domain.Models;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Text;

namespace ReenServer.Data.Repositories

{

public class EquipmentRepository : IEquipmentRepository

{

private ReenDbContext \_context;

public EquipmentRepository(ReenDbContext context)

{

\_context = context;

}

public IEnumerable<Equipment> GetEquipment ()

{

return \_context.Equipment;

}

}

}

## IRepository interface

As mentioned earlier, the backend solution will implement a *generic* repository interface to avoid entity specific repositories – one implementation for every entity.

IRepository.cs – Interface defined in the Domain layer

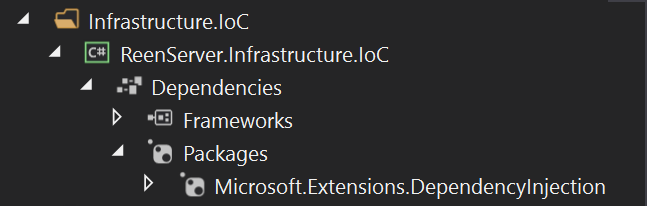
|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq.Expressions;  using System.Text;  namespace ReenWise.Domain.Interfaces  {  public interface IRepository<T> where T : EntityBase  {  T GetById(Guid id);  Ienumerable<T> List();  Ienumerable<T> List(Expression<Func<T, bool>> predicate);  void Add(T entity);  void Delete(T entity);  void Update(T entity);  }  } |

Repository.cs – Implementation in the Infrastructure.Data layer

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Linq.Expressions;  using System.Text;  using Microsoft.EntityFrameworkCore;  using ReenWise.Domain.Interfaces;  using ReenWise.Infrastructure.Data.Context;  namespace ReenWise.Infrastructure.Data.Repositories  {  public class Repository<T> : IRepository<T> where T : EntityBase  {  private readonly ReenWiseDbContext \_dbContext;  public Repository(ReenWiseDbContext dbContext)  {  \_dbContext = dbContext;  }  public T GetById(Guid id)  {  . . .  }  public IEnumerable<T> List()  {  . . .  }  public IEnumerable<T> List(Expression<Func<T, bool>> predicate)  {  . . .  }  public void Add(T entity)  {  . . .  }  public void Update(T entity)  {  . . .  }  public void Delete(T entity)  {  . . .  }  }  } |

### Entity Framework Core

The backend solution will use Entity Framework Core v6 (EF) and the *Code First* design pattern to build and access the actual data store (database). EF is a Microsoft framework that allow us to write C# classes that correspond to database tables, and EF creates the database. This approach is known as *Code First* design pattern.



This folder holds the *Dependency Injection*s that binds the abstractions to the concrete implementation; e.g. connect the application service interfaces to its implementation, and the repository interfaces to its implementation. The code snippet below register the mapping for the service interface *IEquipmentService* and the repository interface *IEquipmentRepository* to their respective implementations.

using System;

using System.Collections.Generic;

using System.Text;

using Microsoft.Extensions.DependencyInjection;

using ReenServer.Application.Interfaces;

using ReenServer.Application.Services;

using ReenServer.Domain.Interfaces;

using ReenServer.Data.Repositories;

namespace ReenServer.Infrastructure.IoC

{

public class DependencyContainer

{

public static void RegisterServices(IServiceCollection services)

{

// Application layer

services.AddScoped<IEquipmentService, EquipmentService>();

// Infrastructure.Data layer

services.AddScoped<IEquipmentRepository, EquipmentRepository>();

}

}

}



The *Independency Injection* registration is done in the Presentation layer - in the ASP.NET application’s *Startup* file.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Builder;

using Microsoft.AspNetCore.Identity;

using Microsoft.AspNetCore.Identity.UI;

using Microsoft.AspNetCore.Hosting;

using Microsoft.AspNetCore.HttpsPolicy;

using Microsoft.EntityFrameworkCore;

using ReenServer.Mvc.Data;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using ReenServer.Data.Context;

using ReenServer.Infrastructure.IoC;

namespace ReenServer.Mvc

{

public class Startup

{

...

RegisterServices(services);

}

private static void RegisterServices(IServiceCollection services)

{

DependencyContainer.RegisterServices(services);

}

}

}



The *Controllers Api* folder holds the *Web*, *UI* and *Presenter* concerns. This means that the controller code in this layer accepts input in the form of http requests over the network (e.g., GET/POST/PUT/DELETE) and returns the output as content formatted as JSON mainly. In the code snippet below a GET *api/equipment* request is executed and a list of all containers returned.

|  |
| --- |
| using System.Collections.Generic;  using Microsoft.AspNetCore.Mvc;  using ReenServer.Application.Interfaces;  using ReenServer.Application.ViewModels;  using ReenServer.Domain.Models;  namespace ReenServer.UI.Controllers.Api  {  [Route("api/[controller]")]  [ApiController]  //[Authorize]  public class EquipmentController : ControllerBase  {  private IEquipmentService \_equipmentService;  public EquipmentController(IEquipmentService equipmentService)  {  \_equipmentService = equipmentService;  }  // GET: api/equipment  [HttpGet]  public IEnumerable<Equipment> Get()  {  EquipmentViewModel model = \_equipmentService.GetEquipment();  return model.Equipment;  }  }  } |

## Data Transfer Objects (DTOs) - AutoMapper

The backend solution will expose data to the Presentation layer as Data Transfer Objects (DTOs). This to hide the Domain Entity data to the outside world. To present data as DTOs we achieve a number of benefits

* Hide particular properties that clients are not supposed to view
* Omit some properties in order to reduce payload size
* Flatten object graphs that contain nested objects, to make them more convenient for clients
* Decouple the service layer from the database layer
* Improve performance
* Flatten object hierarchy
* Exclude properties

Internally the backend server will use *AutoMapper* to map from Domain Entities to DTOs and back. AutoMapper is a simple and free library that solves the mapping operation – converting from one type of object to another.

DTO (Data Transfer objects) is a data container for moving data between layers. DTO is only used to pass data and does not contain any business logic. They only have simple setters and getter.

**Data Transfer Objects** are used to transfer data between the **Application Layer** and the **Presentation Layer**.

The Presentation Layer calls an [Application Service](https://aspnetboilerplate.com/Pages/Documents/Application-Services) method with a Data Transfer Object (**DTO**). The application service then uses these domain objects to perform some specific business logic, and then finally returns a DTO back to the presentation layer. Thus, the Presentation layer is completely isolated from the Domain layer. In an ideally layered application, the presentation layer never works with domain objects, ([Repositories](https://aspnetboilerplate.com/Pages/Documents/Repositories), or [Entities](https://aspnetboilerplate.com/Pages/Documents/Entities)...).

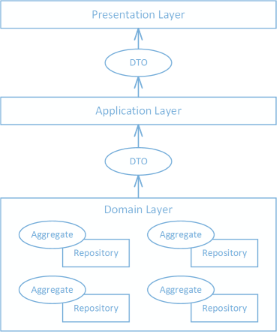
DTOs provide an efficient way of abstracting domain objects from the presentation layer. In effect, your layers are correctly separated. If you want to change the presentation layer completely, you can continue with the existing application and domain layers. Alternatively, you can re-write your domain layer, completely change the database schema, entities and O/RM framework, all without changing the presentation layer. This, of course, is as long as the contracts (method signatures and DTOs) of your application services remain unchanged.

When used to move data from the Domain Layer to the Presentation Layer, a **DTO** is:

“designed to hold the entire number of attributes that need to be **displayed in a view**.” - [Vernon](https://www.safaribooksonline.com/library/view/implementing-domain-driven-design/9780133039900/ch14lev2sec2.html)

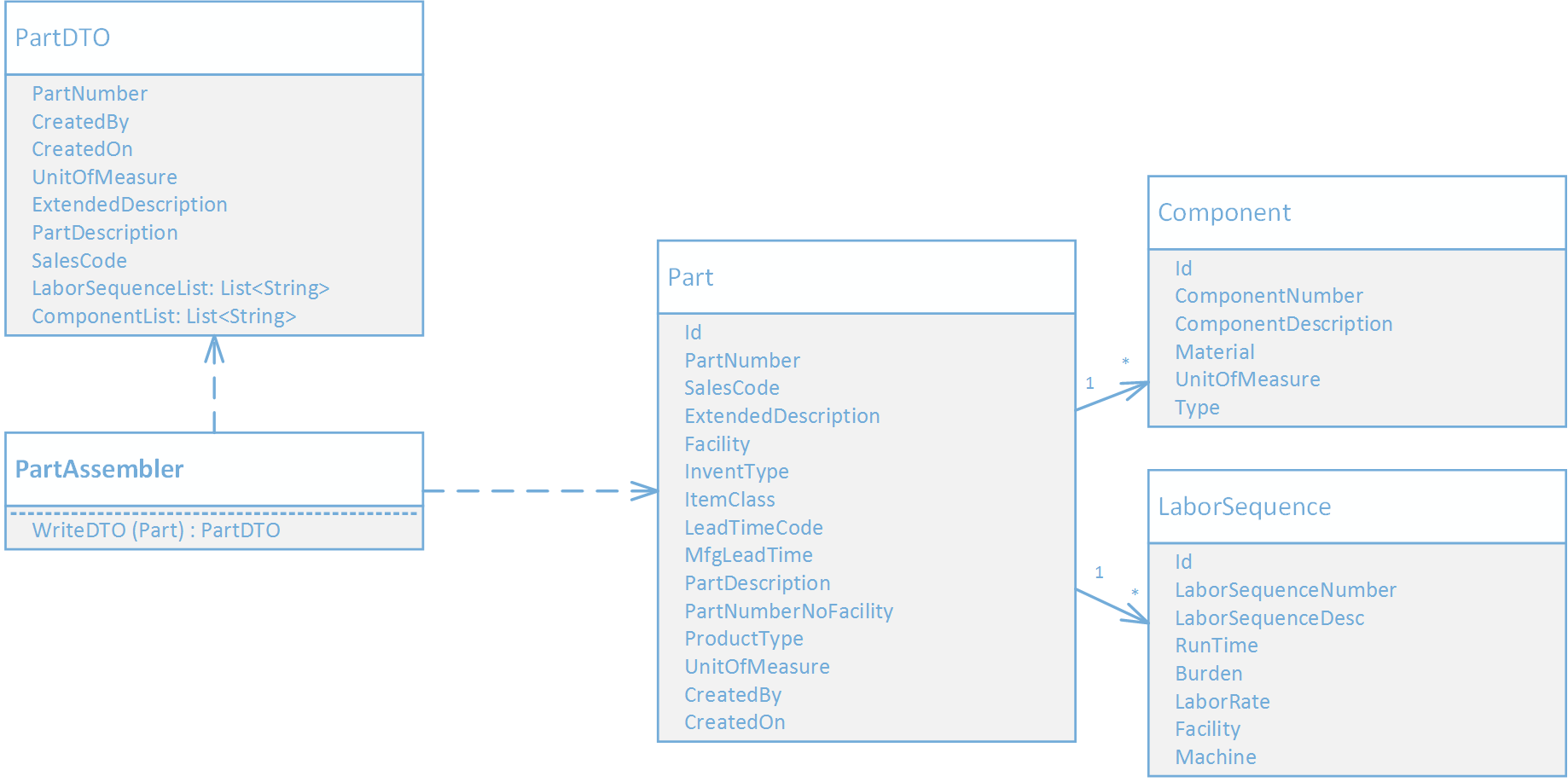
It is in the DTO’s job description to be a carrier of Aggregate state to the Presentation Layer. Still, the DTO needs to be **assembled** somewhere…

### The DTO Assembler

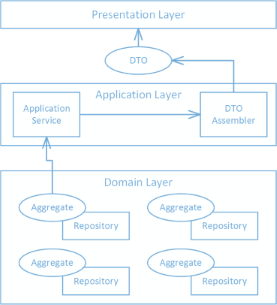
[](https://buildplease.com/img/dto.png)

If not in the Repository, then where do you shape your Aggregate(s) into objects suitable for your presentation layer?

A dedicated [**DTO Assembler**](https://www.safaribooksonline.com/library/view/implementing-domain-driven-design/9780133039900/ch14lev2sec2.html) has the single responsibility of mapping (as in [Mapper](http://www.martinfowler.com/eaaCatalog/mapper.html)) the attributes from the Aggregate(s) to the DTO.

[](https://buildplease.com/img/dto_assembler.png)

A DTO Assembler can live in an [**Application Service**](https://www.safaribooksonline.com/library/view/implementing-domain-driven-design/9780133039900/ch04lev1sec2.html) that is a **client** of your Repository. The Application Service “will use Repositories to read the necessary Aggregate instances and then delegate to a **DTO Assembler** [Fowler, P of EAA] to map the attributes of the DTO.” ([Vernon](https://www.safaribooksonline.com/library/view/implementing-domain-driven-design/9780133039900/ch14lev2sec2.html))

[](https://buildplease.com/img/application_service.png)

The Application Service:

public class PartCatalogService {

private readonly IRepository<Part> \_partRepository;

private readonly PartAssembler \_partAssembler;

public PartCatalogService(IRepository<Part> partRepository, PartAssembler partAssembler) {

\_partRepository = partRepository;

\_partAssembler = partAssembler;

}

public PartDTO GetPart(string partNumber) {

var part = \_partRepository.Get(x => x.PartNumber == partNumber);

return \_partAssembler.WriteDto(part);

}

## CQRS



# ReenWise Web API

TODO: A detailed description of the functions offered through the API.

# Sensor Data Collection

TODO: A description of the sensor data collection process

# ABAX Data Collection

TODO: A description of the ABAX data collection process