Sprint 4 – Data-Access Layer, Repositories & Entity Framework

1. Legen Sie für den Data-Access Layer (DAL) wiederum neue Projekte an, z.B.:

   Abc.Qwe.DataAccess.Sql (für die SQL Implementierungen)
   Abc.Qwe.DataAccess.Interfaces (für die Interfaces)
   Abc.Qwe.DataAccess.Entities (für die neuen Entities)
   Abc.Qwe.DataAccess.Tests (für die Unit Tests)

2. Implementieren Sie das REPOSITORY Pattern, um den Data Access zu kapseln.
   Pro sinnvollem Aggregate (z.B. Parcel + Receipient) soll es ein **C# Interface** für ein Repository geben, das jeweils CRUD Funktionalität für den Aggregate anbietet.
   Dieses Interface kann für mehrere Datenquellen seperat implementiert werden und erlaubt lose Koppelung des Data-Access Layers an den Business Layer.

   Beispiel für ein Repository Interface(nicht vollständig, XX/YY sind Platzhalter):

```
public interface IParcelRepository
{
        int Create(Parcel p);
        void Update(Parcel p);
        void Delete(int id);

        // Mehrere GETS möglich…
        IEnumerable<Parcel> GetByXX(string xx);
        Parcel GetByYY(int yy);

        Parcel GetByTrackingId(string trackingid);
}
```

   Im Falle eines Aggregates werden alle Objektes des Aggregates zurückgegeben. Z.B. enthält das Parcel immer auch den Sender & Receiver.

3. Es gibt seperate **Entities** für den Data-Access Layer, die am Business Layer über AutoMapper gemapped und dann übergeben werden. Diese befinden sich im oben angelegten zugehörigen Projekt.
   Die Entities können / sollen sich, wo sinnvoll von den Entities der anderen Schichten unterscheiden. D.h. auf die Datenbank optimiert sein!

4. Implementieren Sie jeweils die Interfaces für SQL Repositories mit Entity Framework Core 5 für SQL SERVER - um die CRUD Funktionionen für die Datenbank zu implementieren.
   (z.B. SqlParcelRepository : IParcelRepository).

5. Nutzen Sie lokal einen SQL Server (zB. am Besten in Docker, Files siehe Moodle, oder aber auch SQL Express)
   Verwenden Sie im RELEASE / Production Environment eine Azure SQL DB auf die die Azure WebApp zugreift. Probieren Sie aus, ob dies auch wirklich funktioniert 😊

6. Rufen Sie die Repositories unter Verwendung der **Interfaces** aus der **Business Logic** auf.
   (lose gekoppelt, so wie bereits schon die Business Logic aus den Services aufgerufen wurde!)
   Wichtig: Die Repositories werden dabei als Interface über Constructor Parameter in die

Business Logik Klassen übergeben, welche sie dann aufrufen! („Inversion of Control“ – http://youtu.be/GvHWg-89pHc)

Die Business Logic nicht weiß, WELCHE Repositories konkret verwendet werden. Diese können über den Constructor aus dem Controller oder UnitTests per IoC übergeben werden..
z.B.:

```
public class ParcelLogic : IParcelLogic
{
  public ParcelLogic(IParcelRepository repo) { ... }

  public SubmitNewParcel(..)
  {
      //...
      repo.Create(p);
  }
}

IParcelLogic logic = new ParcelLogic(new SqlParcelRepository(...));
```

7. Mocken Sie für die **Unit Tests des Business Layers** die Repositories mit MOQ.

8. Mocken Sie für die **Unit Tests des Data Access Layers** das Entity Framework Core (DbContext) ähnlich der hier beschriebenen Anleitung.
   https://medium.com/@briangoncalves/dbcontext-dbset-mock-for-unit-test-in-c-with-moq-db5c270e68f3

9. **Code Coverage (>= 70%)**, folgende Typen können Sie ignorieren
   - Alle Entities / Models / DTOs / ...
   - Validators
   - Startup.cs, Program.cs
   - Automapper Profiles
   - Custom Attributes, Custom Filters

10. Abgabe:
    Der MAIN Branch ihres Azure DevOps Git Repositories sollte immer den Abgabestand des Source Codes enthalten. Arbeiten Sie auf einem anderen Branch und mergen Sie auf MAIN, sobald Sie abgeben / releasen.
    Beide GruppenteilnehmerInnen müssen die Abgabe (=Angabe der Daten) auf Moodle machen!