Produktrapport

Benjamin Elif Larsen

17. marts 2023

Titleblad

TECH COLLEGE

Tech College Aalborg,
Struervej 70,
9220 Aalborg

Elev:
Benjamin Elif Larsen

Firma:
Skolepraktik

Projekt:
Livsform Observeringsplatform

Uddannelse:
Datateknikker med speciale i programmering

Projektperiode:
23/01/2023 – 21/03/2023

Afleveringsdato:
17/03/2023

Fremlæggelsesdato:
21/03/2023

Vejledere:

Indhold

Figurer

Tabeller

Kapitel 1

Læsevejledning

Denne rapport er en af to rapporter for det tværfaglige projekt. Denne rapport går igennem arkitekturen af produktet, kravspecifikationerne og test sager, de valgte teknologier og mønstre, opsætningen af produktet, samt en brugervejledning. Rapporten kan læses uden at læse processrapporten. Hver kapitel kan læses uden behov for de andre for den fulde forståelse af rapporten, men visse emner vil give mere mening, hvis rapporten læses fra starten.

Kapitel 2

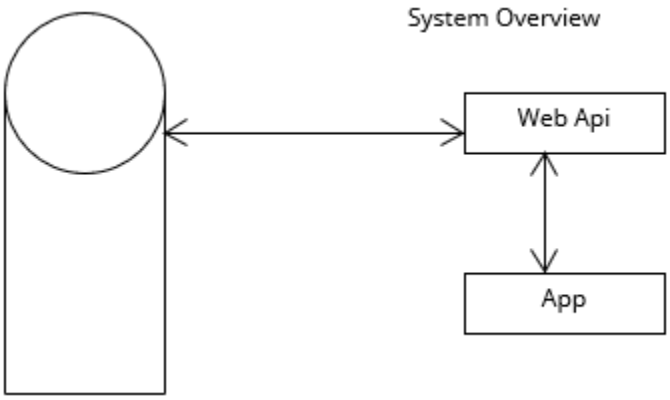
Introduktion

På hovedforløb 5, af uddannelsen Datateknikker med Speciale i Programmering, skal der udvikles et projekt, som består af tre dele (Crossplatform applikation, database og API) under et tværfagligt projekt.
De fag der er indedraget i projektet er 'Systemudvikling og Projektstyring', 'Serverside Programmering II' og 'App II og III'. Grunden til disse fag var for at sikre det tværfaglige projekt dækkede en stor del af undervisningsmaterielet. Formålet med at udviklet sådan et projekt, er at hjælpe med forberedelserne til svendeprøven på hovedforløb 6.

Kapitel 3

Produkt Arkitektur

Systemmet består af tre dele, se figur 3.1. Et Web API, en cross-platform application og en database. Api'et er skevet i C♯10, hvilket er .Net 6.0. Databasen er en SQL-database[1] og for at kommunikere med databasen fra Web API'et, så er en ORM[2] benyttet. Cross-platform application er skrevet i udviklingsplatformen Flutter med sproget Dart 2.9. Applikationen benyttes til at påvirke data over i databasen, samt at kunne hente data fra databasen. Alt kommunikation mellem applikationen og databasen sker igennem Web API'et. Alt kommunikation mellem API'et og applikationen sker over nettet via HTTPS. Figur 3.2 viser en basisk onion diagram over Web API'et. Som det kan ses, så



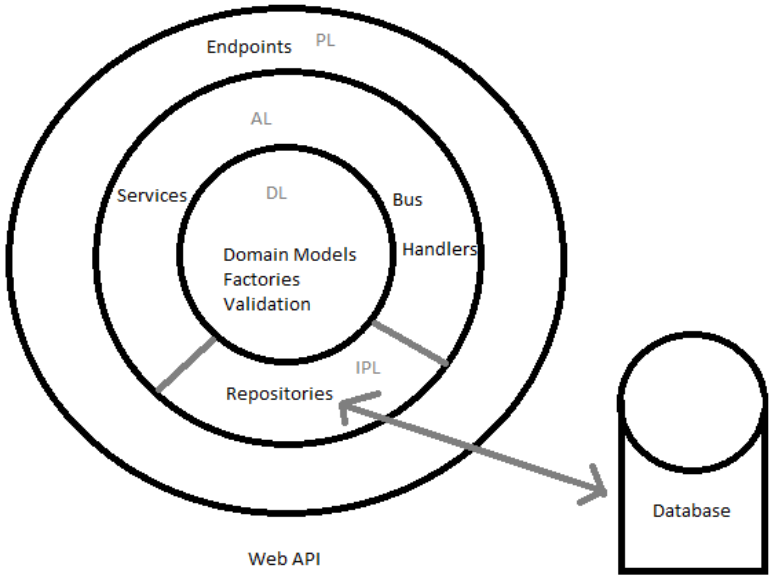
Figur 3.1: System Oversigt

er der tre lag til systemet. Et lag har kun muligheden at kommunikere med laget direkte nederunder den selv og aldrig med laget over den. Det yderste lag består af Presentation Layer (PL), det er her alle endpoints ligger og er dermed det lag andre systemer, som f.eks. crossplatform applikationen, vil kontakte. Det middeste lag inderholder Application Layer (AL) og Infrastructure Persistance Layer (IPL). IPL indeholder alt der har med datalagering at gøre, Unit of Work, repositories og datakontekst. Denne del af softwaren kan kommunikere med databasen, hvilket er vist med <-> pilen. AL inderholder de services endpoints kan kontakte og udførelsen af commands via bussen og handlers. Det skal peges ud af PL ikke har kendskab til bussen og handlers i AL'et, se kapitel 3.2 for en forklaring på hvordan bussen virker, samt IPL'et. Det inderste lag består af Domain Layer (DL). DL'et indeholder alt, der skal bruges af domæne modellerne, samt modellerne dem selv. Dette er factories, validering, commands og fejlbeskeder.

3.1 Filstruktur

Crossplatform applikationens filestruktur er sat op sådan, at alle modeller ligger i mappen 'models', alle services ligger i mappen 'services' og alle sider ligger i mappen 'features'.

[1] SQL står for Structured Query Language.
[2] ORM står for Object Relational Mapping og tillader at mappe et objekt til tabel og tilbage.

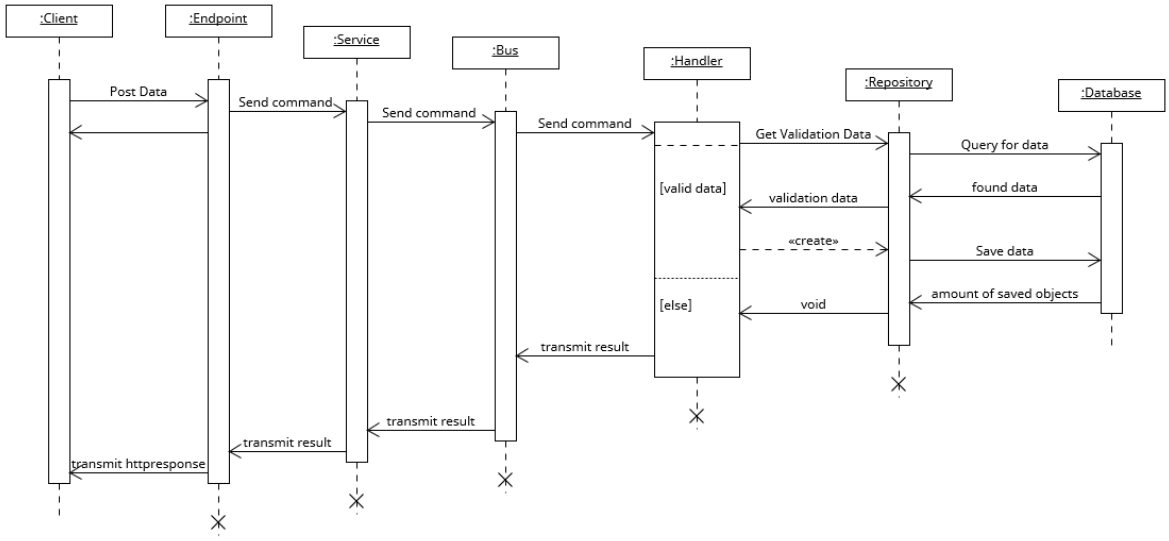


Figur 3.2: Software Onion Diagram

Når det kommer til API'et, så er det opdelt i flere projekter. API indeholder endpoints og alt kode nødvendigt for API'et. Domain inderholder alle services, bussser, domæne modeller, repositories og så videre. Der er tre hovedmapper, AL, DL og IPL. De to projekter Shared og SharedImplementations hænger sammen. Shared indeholder primært generiske metoder og klasser og har formålet at kunne benyttes i forskellige projekter. SharedImplementation inderholder implementeringer af Shareds generiske metoder og klasser. Det sidste projekt er et Test-projekt der inderholder test metoder og test udgaver af unit of work, repository og en test datakontekst.

3.2 Kommunikation mellem moduller

API'et har en lidt mere komplekst kommunikation mellem modulerne end hvad der kunne forventes, se figur 3.3. Som det kan ses, så sender klienten en request hen til en bestemt endpoint på serveren, som fanger requesten og sender den til en service, som videresender den som en command på en bus. I de rigtige endpoints for POST, vil sekvensen se lidt anderledes ud, f.eks. et kald til en factory der udføre validering og objektskabelse. For en Get request, så kan servicen hente data fra databasen igennem en Unit Of Work. Kort sagt, så foregår alt kommunikation mellem en endpoint og resten af systemet igennem en service. Hvis det er en GET endpoint kontaktes et repository igennem Unit of Work direkte ellers sendes en command til command bussen, som sender den til en registret command handler.



Figur 3.3: Sekvens diagram for et 'generisk' http post

Kapitel 4

Kravspecifikationer og Test Sager

For projektet blev det opsat kravspecifikationer og test sager, disse er givet i de to næste afsnit.

4.1 Kravspecifkationer

Kravkspecifikationerne er angivet i billag A. Det skal dog peges ud at en kolumne mangler, Notes, denne blev fjernet for at gøre tabellen mere let læslig, samt at den ikke bragte meget til forståelsen. Note inderholde en note, nået ikke alle havde. Den fulde tabel kan findes i mappen 'Documentation' i projektets rod.
I forhold til de kravspecifikationer med en prioritering på 9999, så indikere 9999 at de krav er blevet fjernet. Disse kravspecifikationer var f.eks. ikke testbare og dermed udskiftet med andre kravspecifikationer.

4.2 Test Sager

Test sager blev udviklet for nogle af kravspecifkationerne og disse kan ses i billag B. Grunden til der ikke er en test sag for hver kravspecifikation var tidsbegrænsninger.

Kapitel 5

Teknologier og Mønstre Forklaring

5.1 API

Denne sektion vil forklare de valgte teknologier benyttet i Web API'et.

5.1.1 EntityFramework Core

API'et benytter EntityFramework Core som ORM, som kan kommunikere med en del forskellige SQL-databaser. Denne ORM gør opsætningen af database og database modeller let at udføre, da en database kontekst fil benyttes. Entityframework Core vil selv finde frem til kolonner, primære- og fremmedenøgler, relationer og ekstra modeller den skal skaber tabeller for, men ikke fået fortalt. Den automatiserer processen med at spore nye entities, entities der er opdateret og entities der skal slettes. På sammen tid tillader den at manuel påvirke entities, da den viser tilstand på alle entities den spore, f.eks. vil den normal kun opdatere de værdier, i databasen, der er blevet ændret i softwaren, men det er muligt at få den til at opdatere alle værdier i databasen. Hvis soft delete er implementeret[1], så kan entities med state Deleted sættes til Unchanged og derefter ændre soft delete feltet til den nødvendige værdi.

5.1.2 ASP.Net Core

ASP.Net er det mest almenlige framework for at skrive web applikationer i C♯. Dette framework er udviklet af Microsoft. Det tillader let at indsætte middleware, der håndterer requests og responses, ind i pipelinen, der aktiveres, når nogen/noget kontakter applikationen. På sammen tid, er det let at sætte endpoints, sikkerhed og ligende op. ASP.net kan automatisk mappe HTTP request til parameterne på endpoints [6] og dermed gør web udvikling letter.

5.1.3 SignalR

SignalR er en teknologi udviklet af Microsoft og benytter websocket til at sende og modtage data mellem en eller flere klienter og en server real-time. Dette er uden at en klient kontakter serveren hele tiden, da serveren ved hvilken klienter der lytter på et bestemt område og kan sende data til dem automatisk [8].

5.1.4 Domain Driven Design

Domain driven design gives her med en meget kort forklaring. Domain driven design en arkitektur, som lægger vægt på domæne eksperter for at hjælpe med at udvikle kerne domæner og domæne logik. Det er selve modellerne der er vigtige og indeholder normalt alt den forretningslogik de skal bruge [5]. Der er normalt givet 4 lag i et system, der benytter domain driven design og disse er givet i tabel 5.1.
I domain driven design er det der to vigtige begraber, domain og bounded context. Et domæne er normalt et meget dækkende område inde for en forretning, f.eks. inde for en bank, uddannelse eller skaldopsamling. Domæner inde for en netbutik kunne være lager, shipping, kundebestilling, butikbestilling, betaling osv.. En bounded context benyttes til at opdele modeller, f.eks. vil en ordre være

[1] Soft delete er at 'gemme' data væk frem for at slette dem i databasen, normalt med en boolean eller et tidspunkt.

Presentation Layer
Application Layer
Domain Layer
Infrastructure Layer

Tabel 5.1: Layers i Domain Driven Design

forskellige for en kunde, for butikken og for shipping. At oprette en model der kunne håndtere alle ville bliver problematisk at benytte. I stedet for vil der oprettes en dedikeret ordre model for hver domæne der skal benytte en ordre [7]. En kunde behøves ikke at vide hvilken dele af deres ordre mangler at blive pakket ned. Shipping behøves ikke at vide om ordren er betalt, kun om hvorvidt den er klar til at blive sendt, hvor den skal sendes hen og hvorvidt den er blevet sendt af sted. Betalings behøves kun at vide hvilken ordre der skal betales for, hvor meget og om ordren er betalt. Den eneste værdi der skal være det samme, mellem de forskellige ordre-modeller, er id'et, da det bruges til at hente data fra de forskellige bounded contexts for en bestemt ordre.
Domain driven design benytter det der kaldes Ubiquitous Language med det formål, at lette kommunikationen mellem folk der arbejder på projektet og brugerne [1]. Et objekt kan have forskellige navne alt efter dens bounded context eller det samme navn kan have forskellige betydninger i forskellige bounded context [2]. F.eks. en person kan være en Ansat (ansat i virksomheden.) i et bounded context og i et andet bounded context være en Operatør (kan bemande tungt maskineri.). En meter kan være en afstandsenhed i en bounded context og i et anden være et stykke udstyr.
Til sidst skal aggregat nævnes. Dette er en samling af modeller der er tilknyttet hinanden. Der er altid en rod model, som er den eneste model, andre aggregater kan have en reference for. Roden står for at styre alle objekter i samlingen og er det eneste objekt med kendskab til data i de andre objekter, f.eks. opdatering, skabning og sletning af objekter inde i aggregatten og alt påvirkninger sker igennem metoder på roden [4]. Et eksempel kunne være en aggregatrod Bygning model med samling af Rum modeller. Hvis et rum bliver malet sker det igennem bygning.PaintRoom(id). En anden aggregat kunne være Adresse, som har en reference til Bygning, men den kan ikke have reference til rummene i bygningen.

Value Object

Inde for domain driven design eksisterer der et koncept kaldt 'value object'. I et value objekt er det kun værdien der er vigtig og hvis flere af dem deler sammen værdi, anses de for at være ens. Et value objekt vil aldrig blive opdateret, kun overskrevet af en ny [4], det vil sige at de er immutable. Et value objekt kan også indeholde forretningslogik [4].

5.1.5 Command Query Responsibility Segregation

Command Query Responsibility Segregation (CQRS) er en mønstre, som opdeler en model i to dele. En Write- og en Readmodel. Writemodellen benyttes af commands og readmodellen benyttes af forespørgsler. Dette øger kompleksiteten af softwaren, da antallet af modeller øges, da der er minimum to modeller for hver model (en writemodel og minimum en readmodel). Dog tillader CQRS bedre kontrol over data-manipulering.

Command

Command-delen, af CQRS, har det formål, at få systemet til at gøre noget, f.eks. oprette en bog, ændre titlen, fyre en ansat osv.. Denne del sender, normalt, ikke noget data tilbage, så lang tid noget ikke går galt. For dette software vil den dog sende en Result, se 5.1.6 for Result Pattern, tilbage pga. det mere simple software opbygning, der benyttes i dette projekt.
En command påvirker kun en enkel aggregaterod, f.eks. en enkel command kan ikke ændre på en bogs titel og en forfatters navn.

Query

Forespørgsel-delen står for at læse data og sende det ud af systemet. En forespørgsel vil aldrig ændre på data i konteksten. Som nævnt i 5.1.5, så benytter der dedikeret readmodels i forespørgsel-delen, dermed foregår en mapping af data fra en model til en anden. Dette hjælper med at styre hvad der sendes ud af systemet, data kan ændres til at passe bedre til et behov, samt at navne på felterne ændres til noget mere passende[2].

5.1.6 Result Pattern

Result Pattern er et mønstre, der hjælper med at sende forskellige slags data tilbage fra et metode kald, uden behov for keywords som `out`, `ref` eller med exception [3]. Dette kan hjælpe udvikleren, siden det gør metode kaldet letter, parameter listen er kortere og der skal ikke håndters exceptions. For softwaren i dette projekt benyttes result pattern i forhold til forespørgsler af data og for resultatet af commandhandlers. De forskellige service metoder, for forespørgsler, modtager data fra repository'et, hvis data bliver fundet puttes det i en `SucceededResult<T>` og sender tilbage. Hvis data ikke bliver fundet, benyttes en `NotFoundResult<T>`, som kan modtage en samling af fejlbeskeder. Det kontaktede endpoint er ligeglad med hvad for et type result den får tilbage fra service kaldet, den kalder en extension metoder, der omdanner result objektet til et http response, se 5.1 for selve udvidelsemetoden.

Listing 5.1: Extension metode

```
1  public static class ControllerExtensions
2  {
3      public static ActionResult FromResult<T>(this ControllerBase controller, Result
           <T> result)
4      {
5          return result.ResultType switch
6          {
7              ResultType.Success => controller.Ok(result.Data),
8              ResultType.SuccessNotData => controller.NoContent(),
9              ResultType.Invalid => controller.BadRequest(result.Errors),
10             ResultType.NotFound => controller.NotFound(result.Errors),
11             ResultType.Unexpected => controller.BadRequest(result.Errors),
12             _ => throw new Exception("Not well tested code if this point has been
                   hit."),
13         };
14     }
15 }
```

5.1.7 Specification Pattern

Specification Pattern er et mønstre med det formål, at hjælpe med at udvælge data og validere data via fastlagte regler. Mønstret kan benyttes til at sætte op boolean-statements, både enlige og sammesatte kæder, og tjekke om et givet objekt overholder reglerne [9]. F.eks. i stedet for at sende flere prædikater over til et repository, kan man have en enkel variable med sammesat validering, hvilket er letter at håndtere i et repository, se 5.2 for forskellen. Line 1-3 benytter en samling af Expression<Func<TEntity, bool>>, hvorimod line 4 benytter en ISpecification<TEntity>. Line 4 tillade både enlige og sammesatte ISpecificiation<TEntity>'er som argument.

Listing 5.2: Specification vs Expression Predicate

```
1      var query = _entities.AsQueryable();
2          query = predicates.Aggregate(query, (current, where) => current.Where(where
               ));
3      return await query.SingleOrDefaultAsync();
4      return (await _entities.ToArrayAsync()).Where(x => predicate.IsSatisfiedBy(x)).
           SingleOrDefault();
```

I 5.3 fremvises et eksempel på en metode der implementerer ISpecification<RecogniseAnimal>, RecogniseAnimal er en command, for data-validering.

[2] F.eks. 'EukaryoteId' kan ændres til 'Id', hvilket gør det lette at benytte feltet andre steder, som i frontend

Listing 5.3: Implementeret ISpecification Eksempel

```
1 internal sealed class IsAnimalSpeciesSat : ISpecification<RecogniseAnimal>
2 {
3     public bool IsSatisfiedBy(RecogniseAnimal candidate)
4     {
5         return !string.IsNullOrWhiteSpace(candidate.Species);
6     }
7 }
```

Hvordan implementeret ISpecification kan blive kaldt er vist i 5.4. Line 2 er en sammensat ISpecification kæde og line 3 er en enkel ISpecification.

Listing 5.4: Kald af både enkel og sammesat ISpecification

```
1 flag += new IsAnimalSpeciesSat().And(new IsAnimalSpeciesNotInUse(_validationData.
      Species)).IsSatisfiedBy(_animal) ? 0 : SpeciesInvalid;
2 flag += new IsAnimalMaxOffspringSat().IsSatisfiedBy(_animal) ? 0 :
      MaximumOffspringInvalid;
```

5.1.8 Kommunikationsbus

Software benytter en kommunikationsbus, herfra bare kaldt en bus, til at sende commands, se 5.1.5, fra en del af systemet til en anden del. Bussen indeholder et 'kort' over hvor en command skal sendes hen ud fra command'ens type. På et tidspunkt under opstart, har en commandhandler registret dens handlers for forskellige commands over i bussen, 5.5. Når en commands skal sendes, så benyttes Dispatch.

Listing 5.5: Command bus og command handler kontrakter

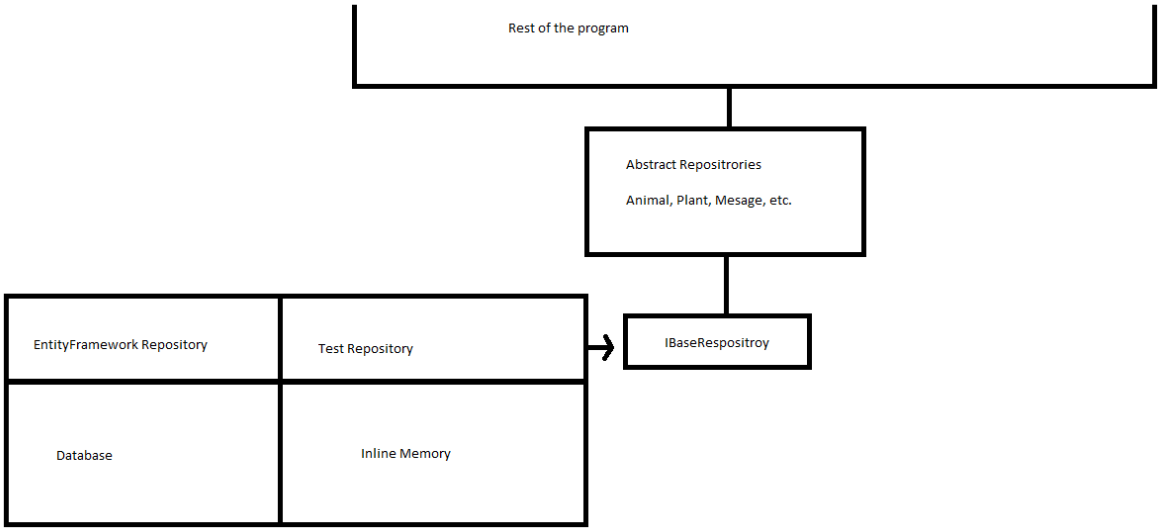
```
1 public interface ICommandBus
2 {
3     public void RegisterHandler<T>(Action<T> handler) where T : ICommand;
4     public void Dispatch<T>(T command) where T : ICommand;
5 }
6
7 public interface ICommandHandler<TCommand> where TCommand : ICommand
8 {
9     public void Handle(TCommand command);
10 }
```

Det skal bemærkes, at en command kan kun sendes til en enkel modtager, da en command er noget der skal gøres. Det modsatte til dette er events[3], som kan have mange modtager og er hvad der er udført.

5.1.9 Repository Pattern

Repository Pattern benyttes til at data-overførelse mellem API'et og datakonteksten, i dette tilfælde en SQL database. Siden softwaren benytter EntityFramework Core som ORM betyder det, at der automatisk er en repository pattern til stede, da EntityFramework Core indeholder dette. Det er dog blevet valgt at implementere et eget repository pattern. Grunden til dette valg, er for at sætte et lag mellem EntityFramework Core og resten af softwaren, så i det tilfælde at ORM'en skal udskifte, så skal kun en mindre del af programmet udskiftes.
Det eget implementeret repository pattern starter med en kontrakt, som så kan implementeres af forskellige konkrete klasser, hver er dedikeret til deres egen ORM. Derefter er der abstrakte repository klasser, som indeholder en variable af repository kontrakten og kan, via dens metoder, henter data gennem kontraktens implementation de har fået via dependency injection. Disse abstrakte repository klasser bliver kaldt af resten af systemet. Kontraktens metoder er så generiske som muligt, hvorimod de abstrakte repository klassers metoder er præcise og lette at benytte.
Et eksemple på det abstrakte repository lag er givet i 5.6. IBaseRepository er kontrakten og det abstrakte lag ved ikke hvor data kommer fra eller hvordan data hentes, kun at den kan hente data via kald til _baseRepository.

[3]Events er ikke en del af dette software.



Figur 5.1: Opbygningen af Repository Pattern i API'et.

Listing 5.6: Abstrakt Repository Design

```
1  public class BookRepository : IBookRepository{
2      private readonly IBaseRepository<Book> _baseRepository;
3
4      public BookRepository(IBaseRepository<Book> baseRepository)
5      {
6          _baseRepository = baseRepository;
7      }
8
9      public async Task<IEnumerable<TMapping>> AllAsync<TMapping>(BaseQuery<Book,
           TMapping> query) where TMapping : BaseReadModel
10     {
11         return await _baseRepository.AllAsync(query);
12     }
13 }
```

Et diagram der viser forholdene mellem de forskellige dele er angivet i figur 5.1. Som det kan ses, så er designet delt i to 'dele'. Selve implementeringerne på den ene side af kontrakten IBaseRepository og det abstrakte lag, der benytter metoderne i IBaseRepository, på den anden side. På sammen tid er der to implementeringer af IBaseRepository, en ORM repository (i form af EntityFramework Core) og en test repository (som ikke har vedholdenhed). Det abstrakte lag ved ikke hvad for en implementering der benyttes.

Specification Pattern

Der skal gøres opmærksom på, at i det implementeret Repository Pattern benyttes Specification Pattern, se kapitel 5.1.7 for dette mønstre, til at udvælge data. Siden dette gøres via metode kald, så kan EntityFramework ikke oversætte disse til SQL, hvilket betyder at det er nødvendigt at hente alt data over til API'et for at udføre operationer på det modtaget data. Dette problem kan løses med koden på line 2 i 5.7.

Listing 5.7: Specification vs Expression

```
1      public Task<IEnumerable<TEntity>> AllByPredicateForOperationAsync(
           ISpecification<TEntity> predicate, params Expression<Func<TEntity, object
           >>[] includes); // Specification
2      public Task<IEnumerable<TEntity>> AllByPredicateForOperationAsync(Expression<
           Func<TEntity, bool>> predicate, params Expression<Func<TEntity, object>>[]
           includes); // Non-specification
```

I forhold til at udføre udregninger i API'et eller databasen, så er der gode sider til at udføre udregningerne på begge sider. Hvis det gøres i databasen, så skal der sendes mindre trafik tilbage til kalderen, men databasen skal bruge tid på udregningerne. Hvis API'et gør det, så skal der sendes mere trafik fra databasen, men API'et har normalt flere ressourcer til at udføre udregningerne med.

5.2 Database

Databasen der benyttes for dette projekt er en SQL-database, som bliver kontaktet af API'et via en ORM. Selve SQL-databasen er en Micrsoft SQL-database (MSSQL). Den første udgave af MSSQL er fra 1989, den udgave der benyttes for dette projekt er en Server 2022 Udvikler udgave.

5.2.1 Relationer

På grund af softwaren benytter Domain Driven Design, kapitel 5.1.4, så burde der helst ikke være relationer mellem modellerne, der ikke tilhøre den samme aggregate rod. Dette, som nævnt i kapitel 5.1.4, er for at undgå en rod ændre på data der tilhøre en anden rod. Dog pga. projektkrav er der relationer mellem de forskellige aggregate rødder i boundered context Message.

5.3 Application

Som nævnt i Process Rapporten, så er crossplatform applikation udviklet i Flutter med Dart som sprog. Flutter blev udgivet i år 2015 og Dart i år 2013. Flutter er et framework, som tillader at udviklet crossplatform applikationer for Windows, Linux, macOS, Web, iOS og Google Androids og kan deploy til alle disse platforme ud fra en enkel codebase [12]. Frameworket er udviklet af Google og er open source. Sproget der benyttes i Flutter er Dart. Flutter tilbyder stateful hot reload, adaptive og responsive design, automatiseret tests og debugging af både kode og design.
I selve projektet blev Dart 2.19.2 benyttet. Dart er en klient-optimeret sprog, udviklet for at kunne gøre applikationer hurtigt ligegyldig platformen og har fokus på UI skabelse. Dart bliver kompilled til enten ARM eller Intel maskinkode eller til JavaScript [10] og sproget ligner TypeScript på mange punkter. Dart inderholder både en Dart VM med just-in-time kompilering og en ahead-of-time kompiler [10].
Flutter benyttes af f.eks. ebay, BMW, Google, Philips hue, Toyota og mange flere [11].
Flutter tillader udvidelse via pakker, som normalt ligger på pub.dev, hvor de skal skrives ind i en pubspec.yaml og Flutter vil foretage resten af arbejdet med at hente og klargøre pakken. Der er dermed let at hente andres pakker, f.eks. pakker for netværk og GPS. Dette betyder også at Flutter er meget let, da den ikke automatiske indeholder alt muligt kode, der muligvis ikke skal benyttes.

Kapitel 6

Opsætning

Her vil der blive forklaret opsætningen af softwaren.

6.1 API

Det er vigtigt at API'et køre på localhost med port 7107, da det er dette punkt applikationen vil kontakte. Denne værdi er sat i lauchSettings.json:profiles:API:applicationUrl i API projektet. Det burde ikke være nødvendig at ændre denne værdi. Opstartsprojektet skal være 'API'. Dermed ikke ISS express eller WSL.

Seed Data

Hvis man vil ændre på seed-data'en, skal dette gøres via Domain/IPL/Context/Seeder.cs filen. Det skal peges ud at det er muligt, at oprette data via kald direkte til Unit of Work eller via kommandbussen. Kommandbussen burde være det fortrukne valg, da det udføre det nødvendige forretningslogik, hvilket Unit of Work ikke gør, f.eks. når en bruger oprettes i User bounded context, at en Author udgave er oprettet i Message bounded context.
Der er kun seed-data for bruger og og livsformer. Grunden til dette, er fordi id'erne på livsformer og brugerne ikke er kendte på dette tidspunkt. Dog kan man udvide koden, således at den hente bruger og livsform id'er og bruge dem i seed-data'en.
Det skal peges ud, at det ikke er muligt at gemme hver kontekst for sig selv, det implementeret Unit of Work vil gemme alle tre på en gang.
Det skal peges ud at parameteren routingRegistry benyttes til at sætte de router op, som commands vil benytte.
Seed-data vil kun blive overført til databasen, hvis databasen er tom.

6.2 Crossplatform Applikation

For at benytte crossplatform applikationen, så kræves det at API'et er oppe og køre først, da alle sider enten sender eller henter data fra API'et. Programmet kan køre på Linux, Windows, macOS, iOS og Google Androids. Hvis en Windows Platform benyttes kræves det, at Dev mode er aktiveret og at apps har tilladelse til at få fat i Geolokation.
Hvis man vil køre programmet kræves det, at Android Studio er installeret, selv hvis det køres igennem en anden IDE, da Android Studio kræves af Flutter for at virke korrekt.

6.3 Database

Databasen kræver at der er opsat en MSSQL server på computeren og at serveren er aktiv. For at oprette tabbellerne, så skal der benyttes nogle commands der ligger i roden af selve projektet. Der er en fil kaldt ToDo.txt, som indeholder de commands der skal køres for at oprette den krævet database, se 6.1 for disse commands. De skal køres fra roden af projektet for at virke. De nødvendige migration filer

eksisteres allerede, men hvis de ikke gør skal commands i 6.2 benyttes. Disse ligger også i ToDo.txt. Hvis databasen skal placeres et andet sted, så ligger der to appsettings filer i API-projektet. Begge indeholder den værdi, som vil bruges til at oprette databasen. Hvis database navn skal være et andet eller databasen skal placeres på en anden server, så er det der disse ændringer skal udføres.

Listing 6.1: Database update commands

```
  cd .\Domain\;
dotnet ef --startup-project ..\API\ database update --context LifeformContext;
dotnet ef --startup-project ..\API\ database update --context UserContext;
dotnet ef --startup-project ..\API\ database update --context MessageContext; cd
    ..;
```

Listing 6.2: Database migration commands

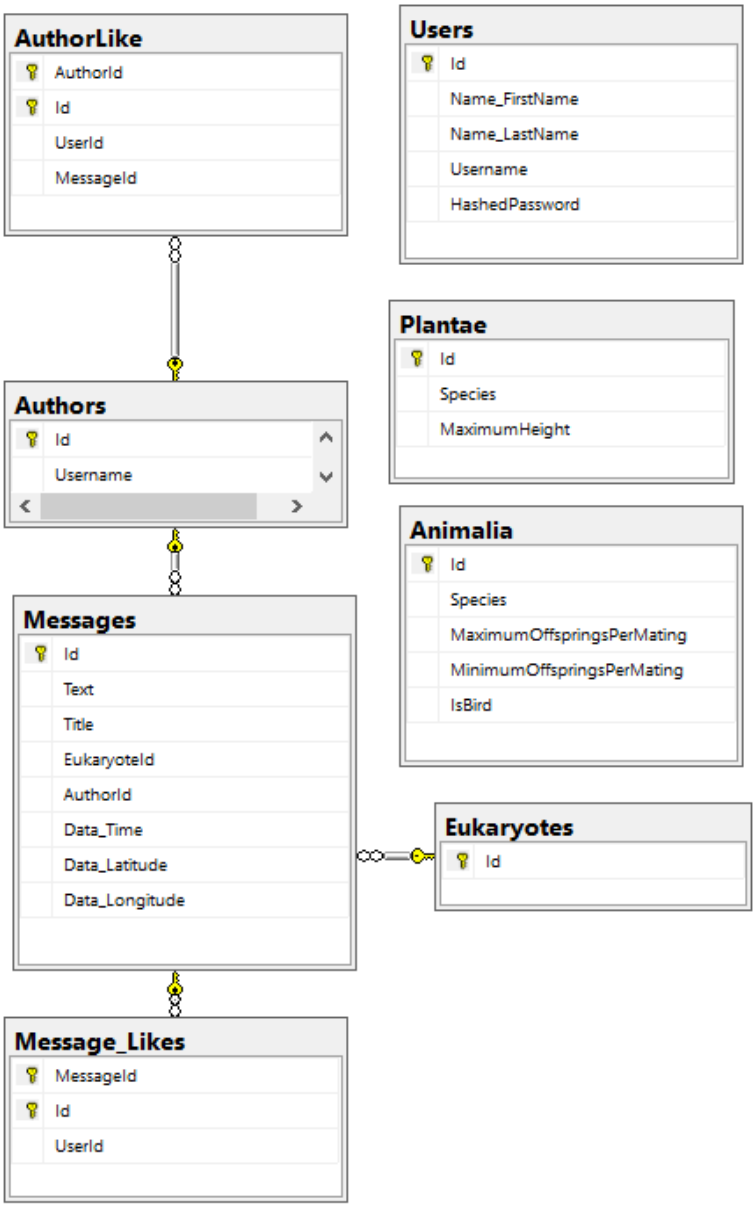
```
cd .\Domain\;
  dotnet ef --startup-project ..\API\ migrations add "init" --context
      LifeformContext  --output-dir IPL/Migrations/Lifeform;
  dotnet ef --startup-project ..\API\ migrations add "init" --context UserContext
      --output-dir IPL/Migrations/User;
  dotnet ef --startup-project ..\API\ migrations add "init" --context
      MessageContext  --output-dir IPL/Migrations/Message; cd ..;
```

De forskellige tabeller, der bliver oprettet i databasen, samt deres relationer, er angivet i figure 6.1. Som det kan blive observeret, så er der ikke nogle relationer mellem de forskellige bounded contexts. Dette kan ses via at Users, Plantae og Animalia ikke har nogle relationer med andre tabeller, hvorimod AuthorLike, Authors, Messages, Eukaryotes og Message_Likes har relationer med andre tabeller. User tilhøre sit eget bounded context, Plantae og Animalia tilhøre deres eget og resten tilhøre Message bounded context.
Der skal lægges mærke til at Username feltet i Authors og Users indeholder duplikeret data[1], men dette er tilladt inde for domain driven design. På sammen tid, så er Plantae og Animalia samlet som en enkel tabel i Message bounded context i form af Eukaryotes. Hvis der kigges på Message tabellen, så er der tre felter der started med Data_, disse stammer fra et value objekt, ObservationTimeAndLocation, men Entiyframework Core er sat op til at overføre felter fra ejet objekter til modellen der ejer de andre objekter (Det sammens kan ses i Users tabellen). Message_Like er opsat til at være sin egen tabel, da Entityframework Core normalt vil overføre felterne til ejer modellen og siden MessageLike er i et 1-M forhold til Message, så ville der har været danne en ny entry i Message for hver MessageLike, dermed skabe en del redundant data.

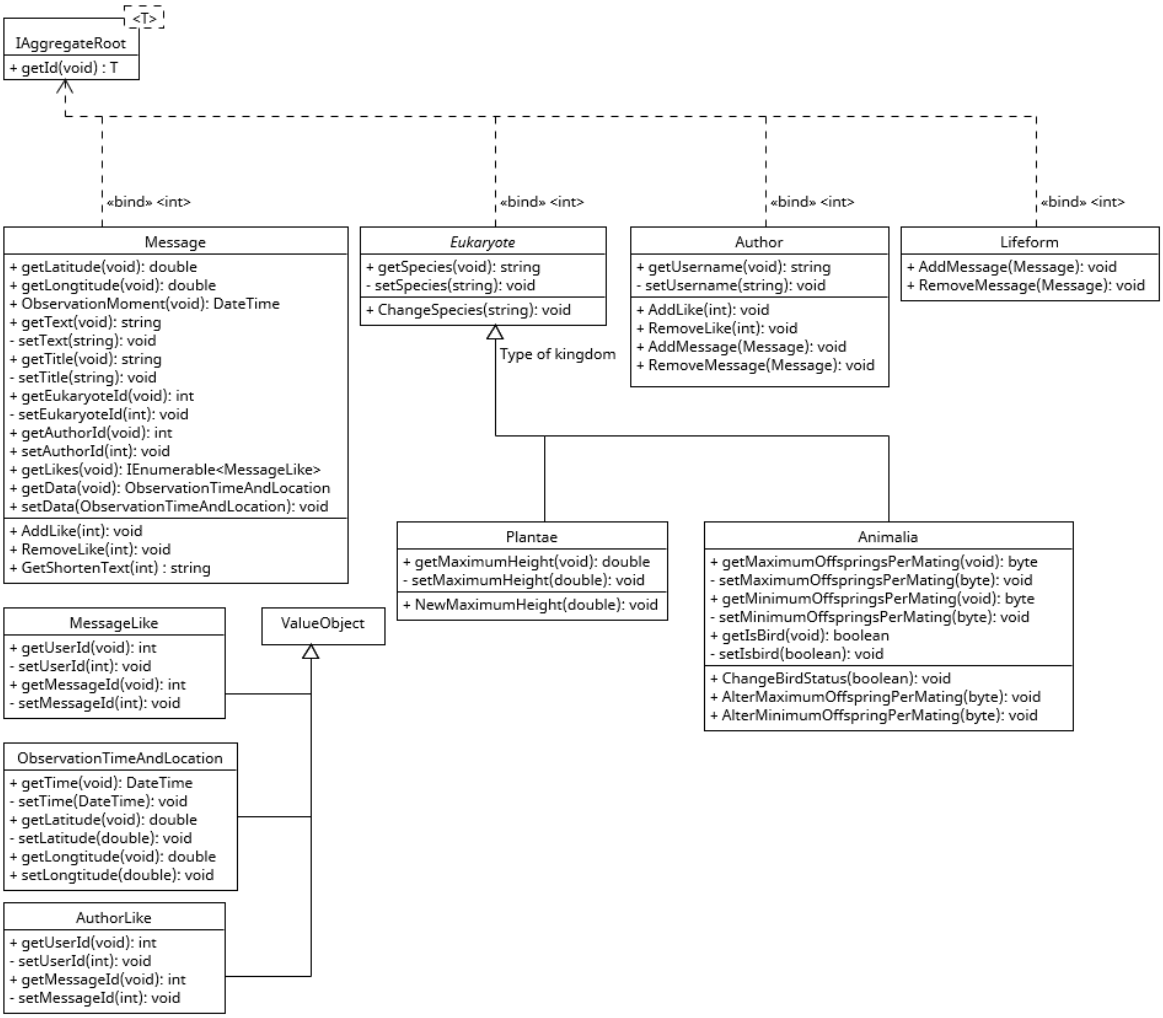
6.4 Klassediagrammer

I figur 6.2 er der angivet et klassediagram af nogle af de implementeret domæne modeller. Der er modeller fra de to bounded context 'Message' og 'Lifeform'. Der er ikke angivet en klassediagram for hver klasse i API'et, da der er 206 cs-filer i API'et, hvor nogle få af dem ikke er i brug og 15 af dem er relateret til test. Der er heller ikke givet diagrammer for klasserne over i applikationen. Fields er ikke vist, samt de properties EntityFramework Core benytter til navigations. I forhold til fields, er de alle backing fields for properties. Det skal peges ud at properties i digrammet er opdel i deres get og set metoder. F.eks. double Latitude {get; set;} er opdelt i getLatitude(void): double og setLatitude(double): void. Grunden til dette, er fordi dette er hvad de er kaldt, når Reflection benyttes.

[1] Det samme gælder for id'erne for de forskellige udgaver af en model.



Figur 6.1: Entity Relationship



Figur 6.2: Klasse diagram

Kapitel 7

Brugervejledning

I dette kapitel gives der brugervejledning for opstart og benyttelse af både Web API'et og crossplatform applikationen.

7.1 API

Hvis man vil benytte Swagger til at interagere med API'et, så vil den starte automatisk op, når API'et starter. Alle endpoints vil være ligger under enten Message eller Lifeform, hvilket er de to kontrollers der er i softwaren. Alle POST, PUT og DELETE endpoints vil fremvise, hvilken modeller de modtager, hvorimod GET ikke fremvise hvad for noget data der bliver sendt tilbage. Det skal peges ud at delete endpoints ikke benytte HTTPDelete, grunden til dette er fordi HTTPDelete ikke tillade en body i request og i nogle tilfælde er det brugbart at havde det, f.eks. en begrundelse eller sletningdato.
Hvis der opstår en fejl med det data man sender til API'et, så vil fejlbeskederne bliver fremvist i Swagger. Fejlbeskeder bliver ikke vist over i applikationen.
Når det kommer til SignalR, så kan det ikke ses i Swagger. Dog findes der en fil i roden af projektet 'PostmanSignalRTesting.txt', som forklare hvordan hubben kan kontaktes via Postman. Teksten fremviser hvordan en message kan blive posted.

7.2 Crossplatform Applikation

Applikationen består af seks sider, som er angivet med deres formål i tabel 7.1.

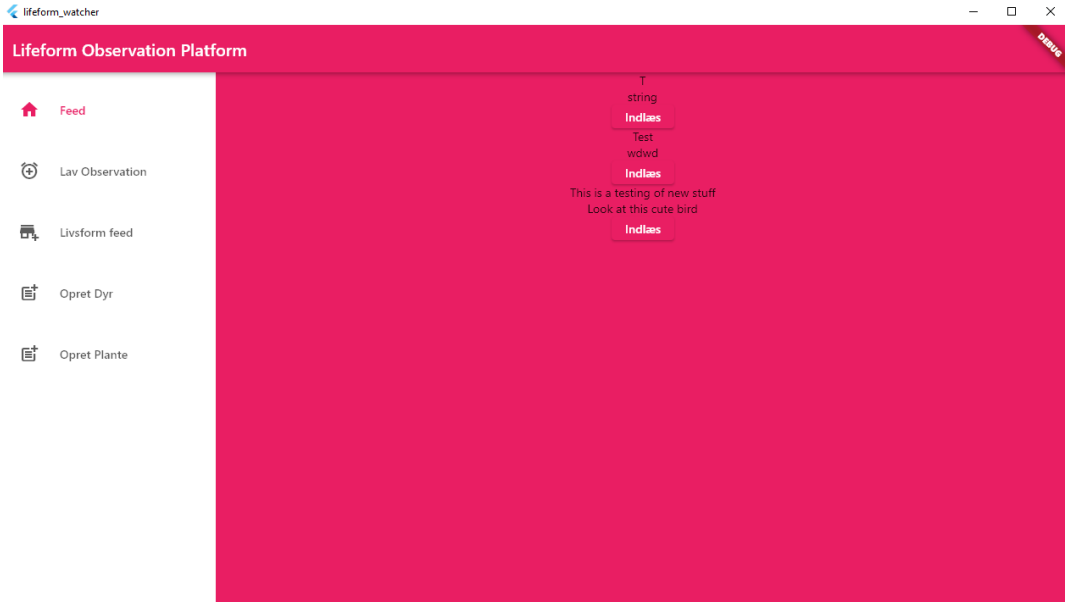
Side	Formål
Feed	Liste over alle beskeder (observationer)
Lav observation	Oprette en ny observation
Livsform feed	Liste over alle dyr og planter
Opret dyr	Oprette en nyt dyr
Opret plante	Oprette en ny plante
Besked detaljer	Se en besked i sin fulde og kunne 'like' den

Tabel 7.1: Crossplatform Application Sider

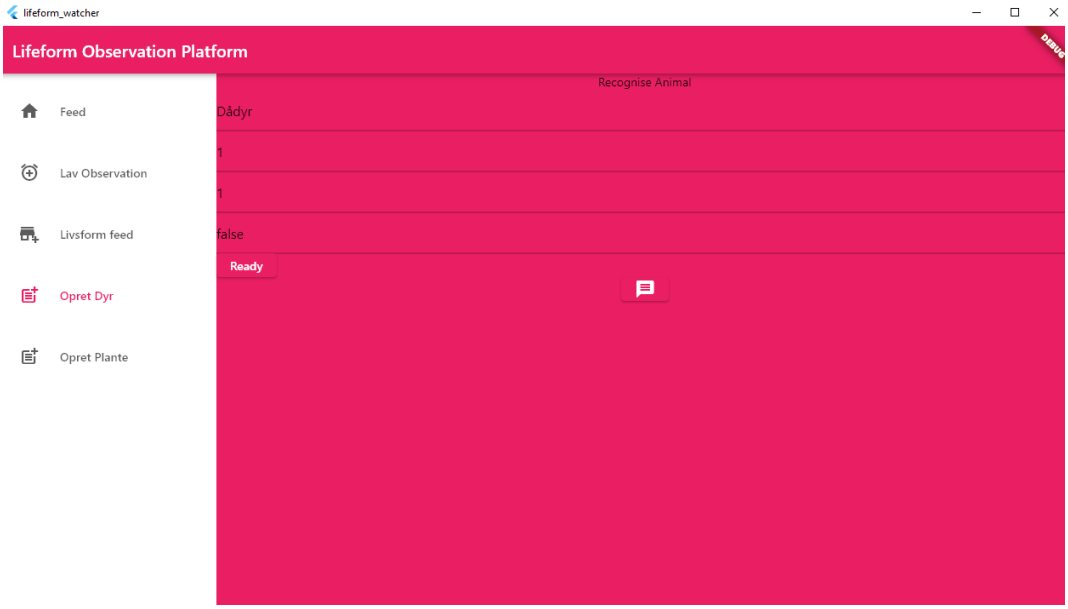
Applikationen starter op på Feed siden, se figur 7.1. Alle sider, med undtagen af Besked Detaljer, har en 'safe zoon' på den højre side, denne safe zoon benyttes til at fremvise navigation til de forskellige sider (som ikke er Besked Detaljer).

Hvis man vil oprette et livsform, skal der entens benytte 'Opet Dyr' eller 'Opret Plante' og derefter udfylde den fremviste form og trykke på knappen 'Ready' for at validere formen. Når dette er udført, skal der trykkes på knappen med et Besked ikon for at sende data'en til API'et. See figur 7.2 og 7.3 for oprettelsen af et dyr. Oprettelse af en plante er ligene. Hvis man vil oprette endnu en livsform, er det nødvendigt at genindlæse siden.

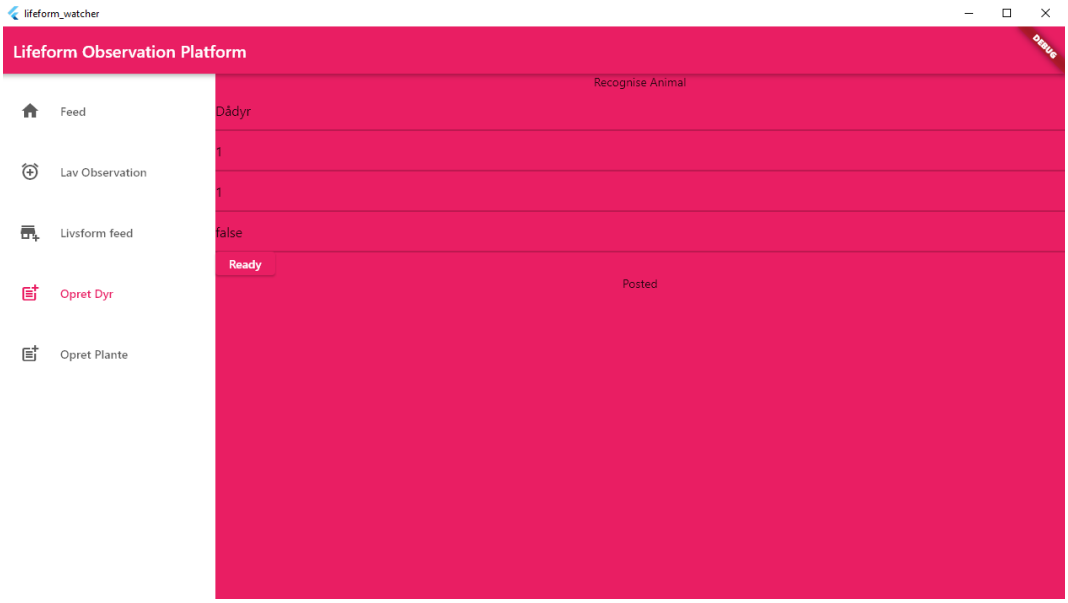
For at se hvilken beskeder der er i API'et, så skal der benyttes selve startsiden af programmet, Feed, da det er den side, hvor dette data vises.



Figur 7.1: Feed side.



Figur 7.2: Livsform klar til indsendelse.

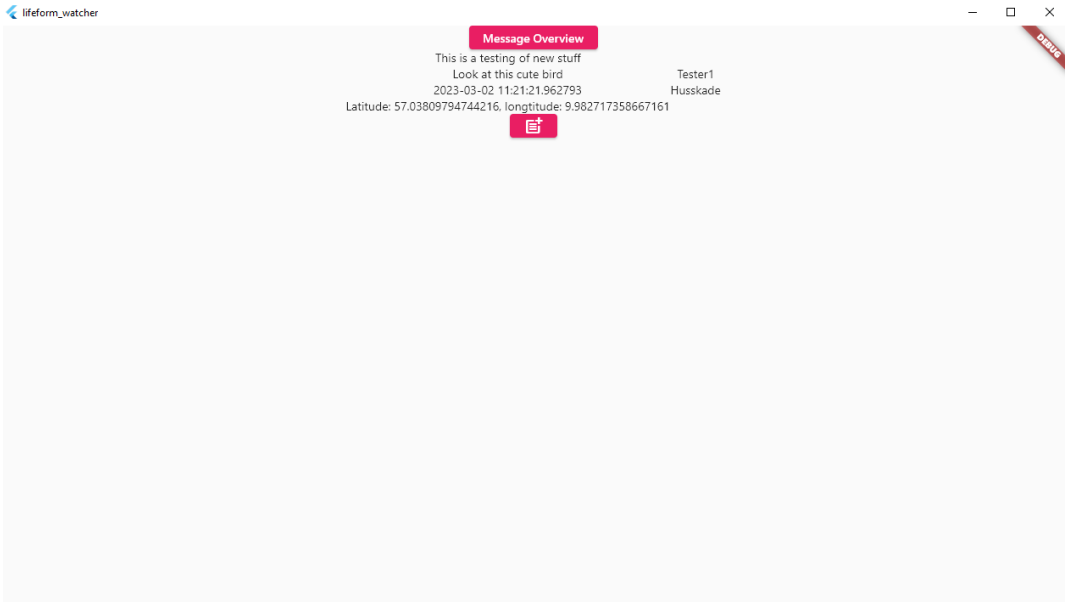


Figur 7.3: Livsform indsendt.

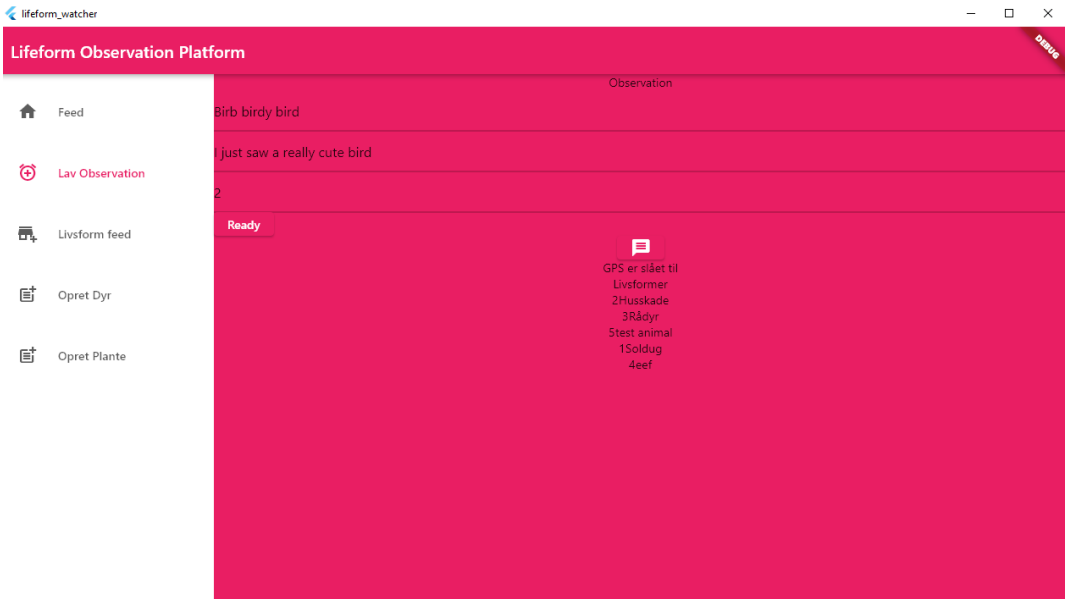
Hvis en besked skal fremvises i fulde, så skal man vælge en besked på feed listen via 'Indlæs'-knappen. På denne side er det muligt, at lave en 'like' på beskeden, figur 7.4. For at gå tilbage til feed'et, så trykkes på knappen 'Message Overview'.

Når der skal indesendes en besked, så gøres dette via 'Lav observation', se figur 7.5 og figur 7.6. Det mulige id'er der kan benyttes står nedunder 'Besked'-ikonnet.

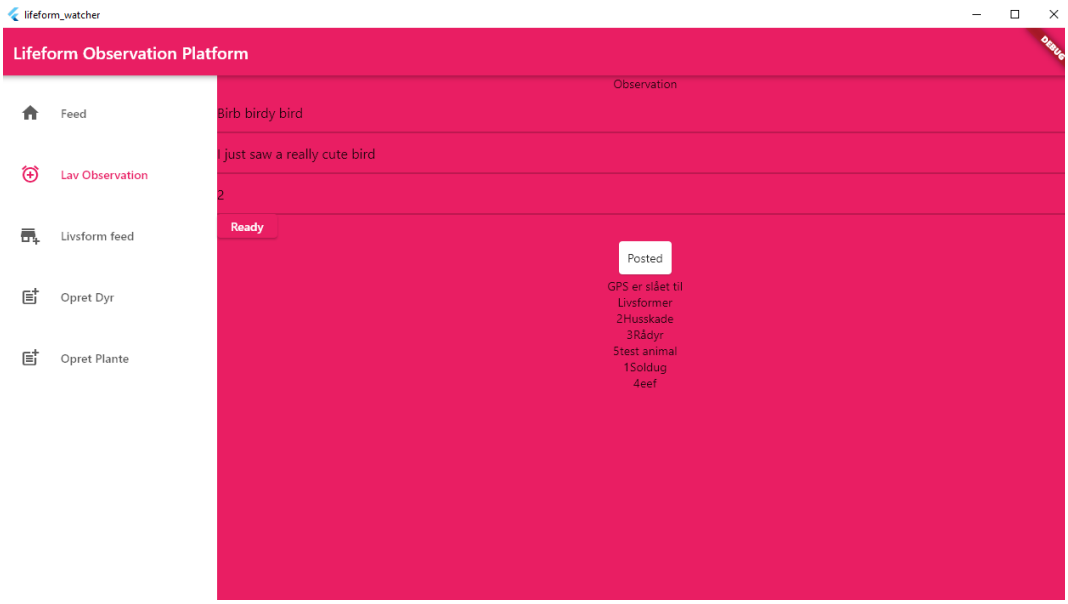
I forhold til brugerinformation, så er dette på nuværende tidspunkt hårdt kodet. Når en besked oprettes er det altid brugeren med id 1, der opretter den. Når der laves en 'like', så er det brugeren med id 2, der liker beskeden. Grunden til dette valg, var fordi bruger-login ikke var implementeret i første udkast af softwaren.



Figur 7.4: Besked Detaljer.



Figur 7.5: Observation klar til indsendelse.



Figur 7.6: Observation indsendt.

Referencer

[1] M. Fowler. "UbiquitousLanguage." (2006), webadr.: https://martinfowler.com/bliki/UbiquitousLanguage.html (hentet 26.01.2023).

[2] M. Fowler. "BoundedContext." (2014), webadr.: https://martinfowler.com/bliki/BoundedContext.html (hentet 20.02.2023).

[3] A. Dunn. "Clean Up Your Client to Business Logic Relationship With a Result Pattern (C Sharp)." (2019), webadr.: https://alexdunn.org/2019/02/25/clean-up-your-client-to-business-logic-relationship-with-a-result-pattern-c/ (hentet 02.03.2023).

[4] P. Holmstöm. "DDD Part 2: Tactical Domain-Driven Design." (2020), webadr.: https://vaadin.com/blog/ddd-part-2-tactical-domain-driven-design (hentet 01.02.2023).

[5] Abp. "Domain Driven Design." (2021), webadr.: https://docs.abp.io/en/abp/latest/Domain-Driven-Design (hentet 26.01.2023).

[6] Microsoft. "Overview of ASP.NET Core." (2022), (hentet 16.03.2023).

[7] Microsoft. "Using domain analysis to model microservices." (2022), webadr.: https://learn.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis (hentet 20.02.2023).

[8] Microsoft. "Real-time ASP.NET with SignalR." (2023), webadr.: https://dotnet.microsoft.com/en-us/apps/aspnet/signalr (hentet 08.03.2023).

[9] E. Evans og M. Fowler. "Specifications." (), webadr.: https://www.martinfowler.com/apsupp/spec.pdf (hentet 02.02.2023).

[10] Google. "Dart Overview." (), webadr.: https://dart.dev/overview (hentet 28.02.2023).

[11] Google. "Flutter." (), webadr.: https://flutter.dev (hentet 28.02.2023).

[12] Google. "Flutter Multi-Platform." (), webadr.: https://flutter.dev/multi-platform (hentet 28.02.2023).

Bilag A

Kravspecifikationer

Id	Category	Requirement	Priority (1 highest)	Source
App-Com-1	Application/Communication	Can retrieve list of lifeforms	1	Brainstorm
App-Com-2	Application/Communication	Can retrieve list of messages	1	Brainstorm
App-Tech-1	Application/Tech	When posting a message the app should auto add time	1	Brainstorm
App-Tech-2	Application/Tech	When posting a message the app should auto add location	1	Brainstorm
App-Tech-3	Application/Tech	When posing a message the app should auto add user id	1	Test Cases
App-UI-1	Application/UI	Feed page displaying all mesages	1	Brainstorm
App-UI-2	Application/UI	Page for posting a message	1	Brainstorm
App-UI-5	Application/UI	All pages are responsive	1	Brainstorm
Context-1	Data Storage	Can read and insert data to and from context via backend	1	Brainstorm
CQRS-1-1	CQRS	Backend can handle commands	1	Brainstorm
End-2	Endpoint	Endpoint for getting all messages	1	Brainstorm
End-3	Endpoint	Endpoint for getting single message	1	Brainstorm
End-4	Endpoint	Endpoint for getting all lifeforms	1	Brainstorm
End-6	Endpoint	Endpoint for posting a message	1	Brainstorm
End-7	Endpoint	Endpoint for getting single lifeform	1	Brainstorm
Error-1	Error Handling	Binary flag from validation can be converted to error messages	1	Brainstorm
Factory-1	Factory	Factories to create entities, one for each aggregate root	1	Brainstorm
Factory-1a	Factory	Factory for User	1	Brainstorm
Factory-1b	Factory	Factory for Message	1	Brainstorm
Factory-1c	Factory	Factory for Lifeform	1	Brainstorm
Factory-5	Factory	Factories uses Result Pattern	1	Brainstorm
Repo-4	Repository	Repository Layer uses transformation mapping	1	Brainstorm
Repo-5	Repository	Repositories can be queried	1	Brainstorm
Repo-6	Repository	Repository saving should be done via a unit of work	1	Brainstorm
Repo-7	Repository	Repository for each aggregate root	1	Brainstorm
Repo-8	Repository	Base repository used by the abstraction repositories	1	Brainstorm
Result-1	ResultPattern	Result Pattern implemented	1	Brainstorm
Result-2	ResultPattern	A result can be converted to a http type (GET, DELETE, POST etc.)	1	Brainstorm
Security-2	Security	Hash and salt user password before storing it	1	Brainstorm

Tabel A.1: Kravspecifikationer Del 1

Id	Category	Requirement	Priority (1 highest)	Source
Service-2	Service	Service for Message	1	Brainstorm
Service-3	Service	Service for Lifeform	1	Brainstorm
Service-3a	Service	Can add animal	1	Brainstorm
Service-3b	Service	Can delete animal	1	Brainstorm
Service-3c	Service	Can update animal	1	Programming
Service-3d	Service	Can add plant	1	Brainstorm
Service-3e	Service	Can delete plant	1	Brainstorm
Service-3f	Service	Can update plant	1	Programming
Service-3g	Service	Can pull all plants	1	Brainstorm
Service-3h	Service	Can pull all animals	1	Brainstorm
Service-3i	Service	Can pull details about one plant	1	Brainstorm
Service-3j	Service	Can pull details about one animal	1	Brainstorm
Service-4	Service	Service method for each endpoint	1	Brainstorm
SpecPtn-1	CQRS	Specification Pattern for data validation	1	Brainstorm
SpecPtn-1a	SpecificationPattern	Can validate animal creation request	1	Brainstorm
SpecPtn-1b	SpecificationPattern	Can validate plant creation request	1	Brainstorm
SpecPtn-1c	SpecificationPattern	Can validate animal update request	1	Brainstorm
SpecPtn-1d	SpecificationPattern	Can validate plant update request	1	Brainstorm
SpecPtn-1e	SpecificationPattern	Can validate user creation request	1	Brainstorm
SpecPtn-1f	SpecificationPattern	Can validate message creation request	1	Brainstorm
SpecPtn-2	SpecificationPattern	Specification Pattern data retrievel from context	1	Brainstorm
App-Com-3	Application/Communication	Can login	2	Brainstorm
App-Com-4	Application/Communication	Can logout	2	Brainstorm
App-Com-5	Application/Communication	It should be possible to like a message on the feed page	2	Brainstorm
Security-1	Security	Middleware should validate if the user is logged in	2	Brainstorm
End-1	Endpoint	Endpoint for login	3	Brainstorm
App-UI-3	Application/UI	Login page with username and password	4	Brainstorm
App-UI-4	Application/UI	Logout page	4	Brainstorm
End-5	Endpoint	Endpoint for registrating user	4	Brainstorm
Security-5	Security	Can create login tokens and refresh tokens	5	Brainstorm
Security-4	Security	Remove expired refresh tokens	7	Brainstorm
File-1	File Handling	Can read image files	8	Brainstorm
File-2	File Handling	Can validate file signatures	8	Brainstorm
Security-3	Security	Rename file before uploading it	8	Brainstorm
File-3	File Handling	Can upload BLOBs to storage	9	Brainstorm
File-4	File Handling	Can remove BLOBs from storage	9	Brainstorm
File-5	File Handling	Can handle file with invalid signature	9	Brainstorm
CQRS-1-2	CQRS	Backend can use commandbus	9999	Brainstorm
CQRS-1-3	CQRS	Handlers for each command	9999	Brainstorm
Error-2	Error Handling	Validation can generate a binary flag of errors	9999	Brainstorm
Repo-1	Repository	Repository for User	9999	Brainstorm
Repo-2	Repository	Repository for Message	9999	Brainstorm
Repo-3	Repository	Repository for Lifeform	9999	Brainstorm
Service-1	Service	Service for User	9999	Brainstorm

Tabel A.2: Kravspecifikationer Del 2

Bilag B

Test Cases

Id	Description	Criteria	Notes
App-Com-1	Get list of lifeforms	It should should either retrieve a 204 or a 200	200 if there is data, else 204
App-Com-2	Get list of messages	It should should either retrieve a 204 or a 200	200 if there is data, else 204
App-Tech-1	Add time to message	The system should pull the correct time + date from the platform	
App-Tech-2	Add location to message	The system should pull the correct location from the platform	Not all platforms may support getting location
App-Tech-3	Add user id to message	The system should pull id from the logged in user	first version it will be hardcoded
App-UI-1	Display messages	All messages should be displayed, such i t is possible to see what is what without taking to much space	
App-UI-2	Post message	it is required to be able to set species out from a list of species	
App-UI-5	Responsive design	Pages should display their information on different screen sizes and rotations	
Context-1	read and write from context	the backend can contact the context and conduct operations in the context	basic CRUD (something like Context-1a, Context-1b, etc for the different test tasks
Context-1a	Create message	using data in a command	PostMessage
Context-1b	Read message	can get a specific entity and map to a specific model	MessageDetails
Context-1c	Read all message	can map all entities to the a specific model	MessageListItem
CQRS-1-1	backend can handle commands	backend can receive a command, transmit it to the correct handler and catch any errors while handling it	
End-2	Getting all messages	It should return 204 if nothing is found else 200	
End-3	Getting a single message	it should return 200 if found else 404 with error	
End-4	Getting all lifeforms	It should return 204 if nothing is found else 200	
End-6	posting a message	it should return 200 if successful else 400 with errors	
End-7	Getting a single lifeform	it should return 200 if found else 404 with error	
Error-1	validation binary flag conversion	binary flags get converted into the correct error messages	
Factory-5	factories uses Result Pattern	should return SuccessResult<T> if success else InvalidResult<T> with errors	
Repo-4	repository transformation mapping	can use mappers to map entities to different models	
Repo-5	Can query using CQRS	Entities are correctly mapped	
Repo-6	Unit of Work successfully saves all context	All contexts are saved at the same time	
Result-2	Correctly result to status conversion	The Result enum is mapped to the correct http status code	
Security-2	Hashing and salting password	Same passwords always generate a different result	
Service-2	Service method for each endpoint	all methods are working correctly.	
Service-3	Service method for each endpoint	all methods are working correctly	
Service-4	Service method for each endpoint	all methods are working correctly	
SpecPur-1	All validations for creation are working	If err they return error messages	
SpecPur-2	All validations for querying work	the querying understand them and the correct data is found	
Factory-1	factory for each aggregate root	each aggregate root model got a factory that validates the creation data and returns correctly	If validation err, return errors else return the model

Tabel B.1: Test Cases