Process Rapport
Livsform Observeringsplatform

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

Kapitel 1

Læsevejledning

Denne rapport er en af to rapporter for det tværfaglige projekt. Denne rapport går igennem den valgte case, tidsplanen, hvorfor de valgte teknologier og mønstre blev valgt, diskussionen af projektet og konklusionen af projektet. Rapporten kan læses uden et behov for at læse produktrapporten.

Kapitel 2

Introduktion

På hovedforløb 5 af 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 undervisnings-materielet. Grunden til at sådan et projekt skal udvikles, er at hjælpe med forberedelserne til svendeprøven på hovedforløb 6.

2.1 Case Beskrivelse

Det liv der er i naturen, kan være meget interessant at kigge på, men det er ikke altid let, hvis overhoved muligt, for en person, at dele hvad de har observeret og hvornår med andre personer. Det er heller ikke altid muligt at vide hvad der er blevet observeret, hvis personen ikke er veltrænet inde for bestemte område, f.eks. fugle eller træer, da visse arter kan ligne andre med meget små forskelle. På samme tid er der et hav af folk, som ville elske at kunne finde frem til bestemte livsformer eller hjælpe andre med at artsbestemme livsformer, men har ikke muligheden for disse. Et andet problem folk har, er at på nuværende tidspunkt, er det kun er muligt at slå livsformer op i opslagsværk, hvilket ikke altid er muligt at have med sig.
Derfor vil et proof-of-concept bliver udviklet, der vil tillade en person at indsende informationer om en observation via en app på deres smartphone og andre kan se informationen med det samme, så de kan selv finde frem til livsformen eller hjælpe med artsbestemme den. Dermed vil folk slippe for at have opslagsværk med sig på deres naturture og have muligheden for både artsbestemmelse-hjælp og hjælper andre med at finde livsformen.
Løsningsforslaget vil tillade en bruger, at indsende en observation som enten 'Plante', 'Dyr' eller 'Ukendt', hvor applikationen automatisk hente information som lokation og tidspunkt. Brugeren vil, under 'Plante' og 'Dyr' kunne indsætte en art, hvis de har kendskab til den. På sammen tid vil andre brugere have en feed over nyeste observationer, som automatisk opdatere, samt kunne sætte arten på ukendte livsform observationer.

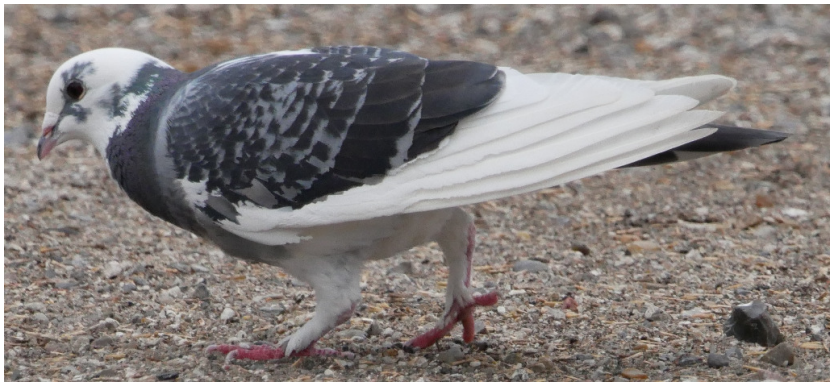
2.2 Problemformulering

Hvordan kan en bruger indsende en livsform-observation, med livsform-kategori og art, tid og sted, og andre kan automatisk modtage det på deres feed, samt give andre bruger muligheden for at give et 'like' på observationen?

2.3 Projekt Afgrænsning

På grund af tidsbegrænsninger, var de følgende afgrænsninger givet.

- Det vil ikke være muligt, at overføre billedere, ej fremvisning af billeder.
- Det vil ikke være muligt, at oprette brugere eller logge ind/ud.



Figur 2.1: En tamdue, en fugl hvis fjerdragt kan ændres meget mellem individer og dermed svær at genkende for det utrænede øje

- Det grafiske brugerflade vil være simpel.
- Det vil ikke være muligt, at slette data via applikationen.
- Det vil ikke være muligt, at oprette en observation uden en art.

Kapitel 3

Projekt Planlægning

Tidsplanen er skrevet som en Gantt diagram, hvor hver kolumne er en arbejdsdag. Tid er kun blevet fastlagt for de dage, hvor tværfaglige fag var fastlagt. Dog starter tidsplanen nogle få dage før den første skemalagt tværfaglige dag, da der var ingen grund til at spilde tid, dette kan ses at f.eks. tidsplan opsætningen starter på dag 2, mens case beskrivelsen skete dag 1 sammen med noget API.
I forhold til aktiviteterne, så er de fleste samlingerne[1] af flere kravspecifikationer. Aktiviteterne for crossplatform application'en er navngivet efter deres primært id i kravspecifikationen, f.eks. alle App-Tech-n ligger under App-Tech i diagrammet. Det skal bemærkes, at API bliver kaldt Backend i tidsplanen. Aktiveterne 'Endpoints', 'CQRS', 'Factories', 'Repositories', 'Result Patterns', 'Services', 'Context' ,Implementing SignalR, og 'Specification Pattern' ligger indenunder API'et. 'Security' aktiviteten dækker over både API'et og applikationen.
Når det kommer til placeringen af de forskellige aktiviteter, er de fleste placeret i starten. Grunden til dette valg, er fordi udvikleren fortrækker at have travlt i starten, så der er bedre tid til at håndtere problemer, nye opgaver, forsinkelser og lignende længere henne i projektet.
Tidsplanen kan ses i billag A. Som det kan observeres, så indeholder tidsplanen også det realiseret tidsplan. Det er dog 'Startdato for plan' og 'Varighed for plan' der er det vigtige punkter for nu, da de fremviser hvornår det var forventet en aktivitet startede og hvor lang tid den ville tage.
Dette projekt var en en-personsprojekt, hvilket betyder der ikke er nogen fordeling af arbejde.

[1] Dem der tilhøre produktudvikling

Kapitel 4

Valgte Teknologier og Mønstre

Dette kapitel vil forklare, hvorfor de forskellige mønstre og teknologier blev valgt for projektet. Det vil også blive nævn andre valgmuligheder og hvorfor disse ikke blev valgt.
Hvis der er nysgerrighed omkring hvad disse teknologier og mønstre er, så henvises der til produktrapporten.

4.1 ORM

Entityframework Core blev valgt som ORM, fordi udvikleren allerede havde god kendskab til Entityframework Core, samt at udvikleren er i ide-fasen af sin egen ORM. Entityframework Core er normalt den ORM der bruges inde for C♯. En anden mulighed var at skrive SQL direkte inde i softwaren, dermed skippe behovet for en ORM.
Grunden til at benytte en ORM er f.eks. for at formindske chancen for SQL-injection, den bedre kontrol over at data-overførelse mellem database og kalderen, ikke nødvendig at skrive egen sql-kode, samt letter arbejde med databasen, det vil sige opretning af tabeller, mappe objekt til tabeller og tilbage, samt styring af både primære- og fremmede nøgler.

4.2 Database

For at sikre at data'en var vedholden blev en SQL-database benytte. Grunden til at benytte en relationel database, var det tillader let at operette, hente data og mere på en overskuelig måde. Det tillader også formindskning af duplikeret data, da relationer mellem de forskellige tabeller kan opsættes. Den SQL-database der blev valgt var en MSSQL-database, udviklet af Microsoft. Andre muligheder var SQLite, MySql eller at gemme i en tekst fil.
Grunden til at en MSSQL-database blev valgt var simpelt fordi en MSSQL-server allerede var installeret på udviklerens computer og da API'et ville benytte Entityframework Core som ORM, så var der ikke nogen rigtig udvikler-forskel, når det kom til sql-databaserne. SQLIT er en meget let, gratist, sql-database, men har meget få funktioner end i forhold til MySQL og MSSQL. MySQL er en gratist database, hvorimod MSSQL koster penge for ikke-udviklere.

4.3 API Framework

Det valgte framework for API'et var Asp.Net Core, udvikleren havde god kendskab til det og det tillader god kontrol, f.eks. at kunne styre pipeline for requests og response via middlewares og sikkerhed, tjekke cookies, mappe fra json-til-objekt og objekt-til-json. Dette framework er hvad der normalt benyttes til udviklet af web applikationer i C♯, hvilket var en anden grund til at Asp.Net Core blev valgt som API framework.
Der er selvfølgelig andre framework, som kunne har været brugt. F.eks. Ruby on Rails, som er baseret på Ruby, og Laravel, som er baseret PHP. Disse blev ikke valgt, da udvikleren havde næsten ikke noget kendskab til Ruby og ikke glad for PHP. Det ville ikke give mening at skulle lære syntaksen på Ruby og dens web framework, når der kun var nogle få uger til at udvikle projektet.

4.4 Real-Time Kommunikation

En del af problemformuleringen var at automatisk at sende data fra serveren til brugerne, når en observation blev indsendt. Her blev SignalR valgt, da det er en teknologi udvikleren ikke har arbejdet med før, det lød interessant og kunne være brugbart at kende, samt at SignalR er en del af ASP.Net Core. En anden teknologi der kunne benyttes for dette var Ajax, men denne teknologi kan ikke bruges med det valgte crossplatform-applikation framework, se 4.7 for den valgte framework, ud fra hvad der blev læst på nettet, hvorimod der fandtes en SignalR klient-implementering pakke for det valgte framework. Ellers kunne der har været spænende, at udvikle egen real-time kommunikations teknologi, men dette ville havde taget for lang tid.

4.5 Domain Driven Design

Grunden til domain driven design blev taget ind i projektet, var fordi det er en interessant arkitektur, som har fokus på domænet, f.eks. modellerne og forretningslogikken. På sammen tid tillader det bedre fin-kontrol over data, da en aggregate rod er den eneste der kan påvirke sit eget data og den eneste der har kendskab til sin data. Udvikleren havde også god kendskab til domain driven design fra andre projektet og viste hvordan det kunne bliver implementeret.
Domain driven design er et meget 'tungt' mønstre og vil tage lang tid at implementere i sit fulde, samt at det normalt bruges for store komplekse software frem for et lille tværfaglig projekt som dette, hvor det vil tage længere tid at implementere korrekt end hvad man får ud af det i sidste ende og der ikke vil være tid til at finde rigtige domæne-eksperter for projektet. Disse kan være grunde til ikke at benytte domain driven design.
Når det kommer til andre valg-muligheder, så kan det være svært at give eksempel, da domain driven design er en arkitektur, som dækker over en hel del og en del mønstre lægger sig mere eller mindre op af domain driven design. Et eksempel kunne være Onion arkitektur, men domain driven design vil normalt ende med at være opbygget på den samme måde i forhold til lag og opdeling af moduler.

4.6 Kommunikationsbus

Grunden til dette blev valgt som en teknologi, var fordi det hjælper med at decouple moduler fra hinanden og streamliner hvordan modulerne kommunikere sammen. På sammen tid var det allerede kendskab til at benytte busser for modul-kommunikation, samt at denne teknologi kan hænge godt sammen med command i CQRS, se 4.6.2 for hvad en command er. På sammen tid hjælper det med testning af software, da det hjælper med at decouple moduler fra hinanden.
Det er muligt at opnå en ligende mængde decoupling via kontrakter, hvilket også formindsker mængden af kode. Dette mønstre øger kompleksiteten af softwaren, samt at det kan være mere svært at følge 'tråden' af data igennem systemet, da der ikke er direkte reference mellem klasser og/eller kontrakter.

4.6.1 Mønstre

Her vil der blevet givet en kort begrundelse for valgte mønstre. For at læse hvad de er, bedes der at kigge i produktrapporten. Ikke alle mønstre fremviser andre valgmuligheder, primært fordi ikke alle mønstre har noget der gør det samme.

4.6.2 Command Query Responsibility Segregation

Grunden til dette mønster benytte, er fordi der normalt læses mere fra end der skrives til en model. På samme tid er der ikke altid behov for alt data modellen har, mens felter kan navngives til noget mere brugervenligt og mindre data sendes igennem modulerne, f.eks. mellem database og API og applikation. Til sidst hjælper det med refactoring, da write- og readmodellerne er adskilte [2].
Dette mønstre øger kompleksiteten af softwaren, da alle database modeller deles op. En andet muligt valg var benyttelsen af en auto-mapper som Mapster til at mappe data fra en model til en anden, men fra erfaring kan Mapster let skabe fejl, tage en del manual opsætning at mapper mellem modeller og kan øge mængden af arbejde, når der skal refaktor.
Til sidst kan command query separation nævnes, dette er det mønstre som CQRS er baseret på, og

går ud på at en metode enten ændre data (command) eller henter data (query), aldrig begge to på sammen tid [1]. Dermed er der en model frem for 2+ modeller, men modellen indeholder metoder for at påvirke dens data og metoder for at hente det data, som kalderen har brug for.

4.6.3 Repository Pattern

Repository Pattern blev inddraget i dette projekt, da det et godt mønster at have liggende mellem en datakontekst og resten af systemet. Det kan hjælpe med skrivning og læsning af data til og fra databasen.
Givet projektet benytter Entityframework Core er det ikke nødvendigt at selv-implementere dette mønstre, da Entityframework Core indeholder mønstreret, men det blev valgt at inddrage det for at fjerne koblingen mellem Entityframework Core og resten af softwaren, da udvikleren godt kan lide at kunne skifte ORM uden at skulle ændre for meget i koden. Dette er brugbart, når der skal udføres automatiseret tests, da test repositories kan bringes ind uden at skulle ændre i resten af softwaren.

4.6.4 Specification Pattern

Grunden til dette mønstre blev inddraget i systemet, var fordi det kan bruges til data-validering, udvælgelse af data og tillader at let udføre komplekse validering og udvælgelser.
En anden mulighed var ikke at benytte dette mønstre, da det eneste formål er at streamline data-udvælgelse og -validering, men udvikleren fortrækker at streamline opgave-udførelser[1], når det er muligt.
Til sidst kan der nævnes, at der ikke var kendskab til andre mønstre som kan udføre data-validering og data-udvælgelse. Der er dog kendskab til et lidt anden design af mønstret [3], som udføre noget lignende, men kun for data-udvælgelse, og udvikleren har benyttet denne udgave før mens personen var i lære. Det specification pattern der dog blev brugt for dette projekt, var det design der normalt bruges inde for domain driven design [4].

4.6.5 Result Pattern

Result Pattern blev udvalgt på grund af det tillader, at sende flere salgs data tilbage fra en metode kald, uden at skulle benytte `ref`, `out`, exception handling eller klasser der er designet for alle muligheder (Gik godt felt, ikke-korrekt-data felt, exception felt, osv.). Det er selvfølgelig muligt ikke at benytte dette mønstre med det kost at skulle håndtere mere komplekse metode kald og fejl.

4.7 Application Framework

Applicationen'en blev udviklet i Dart med Flutter som framework. Begge var helt ukendte til udvikleren, hvilket var grunden til at de blev udvalgt. På sammen tid har de eksisteret i flere år, hvilket forhåbentligt betyder at bugs og sikkerhedbrister er blevet rettet. Valget af Dart og Flutter betød, at der ikke kendskab til bestemte teknologier indefor disse.
Andre teknologier, der kunne har blevet valgt, var Progressive Web APP (PWA) eller Maui. Grunden til at Flutter blev valgt frem for de andre, var fordi Flutter lød mere spændende end PWA og Maui er baseret på C♯og dermed tætter på hvad udvikleren har kendskab til. Flutter har en del fokus på GUI, hvorimod udvikleren normalt følger 'funktion over design'-filosoffen, og dette framework kunne potentielt være brugbart for udvikleren senere, i forhold til udviklingen af egne produkter.

[1] En opgave i dette tilfælde er nået der skal gøres, f.eks. validering af data, udregning af en værdi, henting af objekter, osv..

Kapitel 5

Realiseret Tidsplan

Den realiseret tidsplan kan ses i billag A. Her er det' Faktisk Startsdato' og 'Faktisk Varighed', som er de vigtige punkter. Den gule farve før den lilla farve indikerer en tidligere start end forventet og den gule farve efter den lilla farve indikerer, at aktiviteten tog længere tid end forventet. De steder hvor farven er gennemsigtig indikerer det, at aktiviteten blev udført hurtigere end forventet eller startet senere end forventet.
Næsten alle aktiviteter blev udført, kun dem der var relateret til SignalR og sikkerhed blev ikke udført
Som det kan ses, i forhold til udviklingen af applikationen, så tog det længere tid end forventet, hvilket kom fra manglende kendskab til Dart og Flutter. F.eks. omdannelse af objekt til json og json til objekt var mere besværligt end forventet, det meste ikke-GUI relateret arbejde tog længre tid end forventet.
For API'et, så er der en blanding i forhold til at en aktivitet blev udført hurtigere, langsommer eller til tiden. De API aktiviteter der tog længere tid var pga. nogle små-ændringer og udvidelser, der blev udført. F.eks. blev data-struktureret ændret en del på et tidspunkt, således at det passede bedre til domain driven design. 'Backend base struktur' tog en del kortere tid, da det blev valgt ikke at benytte Events og Process Managers, da disse ville havde krævet lang tid at implementere pga. deres kompleksiteter.
Der er nogle aktiviteter, som Security, som ikke blev arbejdet på, da disse ikke havde de højeste prioriterer og det blev valgt at fokusere på at skrive rapporterne. Disse skulle nok ikke har været med i tidsplanen for den første udkast af produktet.
Testene tog meget kortere tid end forventet. Grunden til dette var fordi koden hele tiden blev testet, mens det blev udviklet. For API'et blev nogle få automatiske tests sat op ellers blev resten udført i hånden.
I forhold til dokumentering, så tog kun Gantt opsætning længere tid end forventet, men dette kom fra manglende erfaring med Gantt. Det tog også længere tid end forventet at sætte kravspecifikationerne op, dette var fordi det blev valgt at dele dem op i mindre dele, der var letter at teste. Nået der manglede i tidsplanen i forhold til dokumentation, var aktiviteter for opstillingen af diagrammer som ER-, sekvens- og klassediagrammer. ER-diagram kan let opstilles af management-software for den valgte database, men sekvens- og klassediagrammer skulle skrives 'i hånden' og i form af UML, hvilket er en tidskrævende opgave, da der ikke var noget værktøj til at automatisere klassediagram skabelsen.
De fleste aktiviteter blev udført på sammen tid som andre. F.eks. alle App/* blev udført sammen, da de 'krævede' hinanden for at kunne blive færdiggjort og det kunne måske have givet mere mening at samle dem under en stor aktivitet og så have en beskrivelse af hvad der vil blive udført under den store aktivitet.
I forhold til SignalR, så er det til dels implementeret. Grunden til dette, er fordi det først blev startet på meget sent i udviklingsforløbet, samt at det var en teknologi, udvikleren ikke rigtigt have kendskab til i forvejen. SignalR er kun implementeret i API'et og der ligger en text fil 'PostmanSignalRTesting' i projektroden, der forklare hvordan Postman kan bruges til at kommunikere med denne del af API'et. Grunden til at SignalR i API'et ikke er 100 % udført, er fordi der blev løbt ind i nogle problemer, der ville havde krævet større ændringer i API'et for at løse.
Det burde nævnes at udvikleren har en vane med at overestimere en opgave frem for at underestimere en opgave, hvilket personen viste fra sin tid som lærling.

Kapitel 6

Diskussion

Slutproduktet bestod af tre dele, et crossplatform applikation, et Web API og en SQL-database. Det var muligt at kommunikere direkte med API'et via dens endpoints eller igennem crossplatform applikationen. Applikationen bestod af seks sider, som forbundene med forskellige endpoints i API'et og tillod en mere brugervenlig kommunikation med API'et. Alt kommunikation med databasen foregik via API'et.
API'et benyttede domain driven design, men det implementeret udgave overholdte ikke alle kravene. F.eks. var der navigation properties mellem aggregate rødder, hvilket tillader en rod at påvirke en anden rod. Grunden til at domain driven design ikke blev fuldt implementeret, var begrænsninger opstået af projektkrav og tid.
Når det kom til automatiseret testning af API'et, så var det muligt at teste factories og metoder på domæne modellerne, hvilket var de primære ting handlers vil kalde, samt testning af selve handlerne med en mock repository og mock kontekst, men der var ikke automatiseret test for alle dele af API'et.
I forhold til datalagering, så blev det overvejet at benytte domæne events[1] med en Event Store. Dette ville formindske mængde af tabeller, da der som minimum skal benyttes en tabel for at holde på events. Dog er det praktisk at havde en tabel, hvor der ligger information om entities, f.eks. type, id og versionsnummer[2]. Grunden til dette ikke blev implementeret var tidsbegrænsninger, samt kravene for databasen.
På nuværnede tidspunkt af denne rapports skrevning, så er brugerinformation hårdt-kodet i applikationen, men i virkeligheden ville Json Web Token (JWT) blive benyttet sammen med en refresh JWT. Disse ville blive sendt med requests i deres header og en middleware i API'ets request pipeline ville tjekke om JWT'en var korrekt. Dette ville også betydet, at bruger id'et ikke ville har været sendt med som en del af modellen i requesten, denne værdi ville har været hentet ud af JWT'en.
Kravspecifikationerne kunne har været opdelt bedre, da det var besværligt at komme op med lette/små test sager for flere af dem. Et eksempel kunne være krav 'Factory-1', som dækkede alle factories og burde har været opdelt i mindre dele, f.eks. 'Factory-1-a' Animalia, 'Factory-1-b' Message osv.. Dette er blevet gjort for nogle få krav, men burde har været gjort for alle komplekse krav.

[1] En event beskriver påvirkningen af en entity, f.eks. oprettelse, ændringer, sletning med de nødvendige data til at kunne udføre denne påvirkning igen.
[2] Hver event øger versionnummeret.

Kapitel 7

Konklusion

Formålet med dette projekt var udviklingen af en første-udkast prototype for et produkt, hvor en bruger vil kunne oprette observationen af en livsform og dele denne observation med andre og 'like' de observationer der var indsendt. Dermed blev næsten alle af de stillede krav opfyldt. Det eneste der manglede var det automatiske posting på feed'en hos andre brugere, når en observation blev oprettet. Produktet endte med at består af 3 dele, en database (Microsoft SQL-database), et web API (Asp.Net Core) og en crossplatform applikation (Flutter).
I applikationen er det muligt at oprette livsformer, se hvilken livsformer der findes, oprette observation på en livsform, samt at kunne se oprettede observation og 'like' dem. Selve den grafiske del, består af et meget simpel UI og UX. Applikationen vil automatisk indehente lokation og tidspunkt, når en observation oprettes.
I forhold til API'et, så er muligt at hente, oprette, ændre og slette data via endpoints. Hvis der opstår fejl, f.eks. ukorrekt data under oprettelse eller henting af ikke-eksisterende data, så ville API'et sende brugbare fejlbeskeder tilbage til kalderen.
Der var krav, at produktet indeholdte en database, et web API og en crossplatform applikation. Crossplatformen applikationen skulle kunne kommunikere med API'et og API'et skulle kunne kommunikere med både applikationen og databasen. Disse krav blev opfyldt. Applikationen kunne kommunikere med API'et via HTTPS endpoints. Database-kommunikationen skete via en ORM.

Referencer

[1] M. Fowler. "CommandQuerySeparation." (2005), webadr.: https://martinfowler.com/bliki/CommandQuerySeparation.html (hentet 16.03.2023).

[2] M. Fowler. "SQRS." (2011), webadr.: https://martinfowler.com/bliki/CQRS.html (hentet 16.03.2023).

[3] R. Santos. ".NET Core — Using the Specification pattern alongside a generic Repository." (2019), webadr.: https://medium.com/@rudyzio92/net-core-using-the-specification-pattern-alongside-a-generic-repository-318cd4eea4aa (hentet 16.03.2023).

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

Bilag A

Tidsplan