Processrapport
Cateringplatform

Benjamin Elif Larsen

2. maj 2024

Titelblad

TECHCOLLEGE

Techcollege Aalborg,
Struervej 70,
9220 Aalborg

Elev:
Benjamin Elif Larsen

Firma:
EnergyInvest ApS

Projekt:
Cateringplatform

Uddannelse:
Datateknikker med Speciale i Programmering

Projektperiode:
09/04/2024 – 15/05/2024

Afleveringsdato:
03/05/2024

Fremlæggelsesdato:
14/05/2024

Vejledere:
Frank Rosbak
Lars Thise Pedersen

Indhold

1 Læsevejledning **1**

2 Introduktion **2**
2.1 Case Beskrivelse . . . 2
2.2 Problemformulering . . . 2
2.3 Projekt Afgrænsning . . . 3

3 Projekt Planlægning **4**

4 Valgte Teknologier og Mønstre **5**
4.1 Message Broker . . . 5
4.1.1 Kafka . . . 5
4.1.2 RabbitMQ . . . 6
4.2 ORM . . . 6
4.3 Database . . . 6
4.4 Docker . . . 7
4.5 API Framework og REST . . . 7
4.6 Frontend Framework . . . 8
4.7 Test . . . 8
4.7.1 Automatiskeret Test Frameworks . . . 9
4.8 Domain Driven Design . . . 9
4.9 Logging . . . 10
4.10 Konfigurationsstyring . . . 10
4.11 Mønstre . . . 11
4.11.1 Command Query Responsibility Segregation . . . 11
4.11.2 Repository Pattern . . . 11
4.11.3 Unit Of Work . . . 11
4.11.4 Result Pattern . . . 11
4.12 Software Udvikling Framework . . . 12

5 Realiseret Tidsplan **13**

6 Diskussion **14**

7 Konklusion **16**

A Tidsplan **18**

B Dagbog **20**
B.1 9/4/24 . . . 20
B.2 10/4/24 . . . 20
B.3 11/4/24 . . . 20
B.4 12/4/24 . . . 20
B.5 15/4/24 . . . 21
B.6 16/4/24 . . . 21
B.7 17/4/24 . . . 21

B.8 18/4/24 21
B.9 19/4/24 21
B.10 22/4/24 21
B.11 23/4/24 22
B.12 24/4/24 22
B.13 25/4/24 22
B.14 26/4/24 22
B.15 29/4/24 22
B.16 30/4/24 22
B.17 1/5/24 22
B.18 2/5/24 22

Figurer

4.1 Seq dashboard 10
4.2 Seq fejl log 10

6.1 Løsningen 14

Kapitel 1

Læsevejledning

Denne rapport er en af to rapporter der blev skrevet for svendeprøven, den anden rapport er produktrapporten. Rapporten vil gennemgå den valgte case, tidsplanen, teknologi og mønstre valg, diskussion af den endelige produkt og konkludere på produktet. Denne rapport kan læses uden behov for at havde læst produktrapporten.

Kapitel 2

Introduktion

På hovedforløb 6 af Datateknikker med Speciale i Programmering skulle der udvikles et svendeprøveprojekt med de følgende krav:

- Konfigurationsstyring
- Sikkerhed
- Test
- Database
- Server

På sammen tid skulle en af de følgende krav også inddrages:

- Klient/Server
- APP Udvikling
- Desktop Program
- Embedded

Det blev valgt at inddrage en klient, via browser, som den ekstra del. Grunden til dette, var fordi udvikleren havde kun en smule kendskab til frontend kodning og udvidet viden inde for dette område blev anset for at være brugbart. På sammen tid gav en web-browser klient mere mening i den moderneverden frem f.eks. et desktop-program, når det kom til at bestille mad hos en catering-virksomhed.
Formålet med svendeprøven var at fremvise den viden der blev anskaffet under uddannelsen.
Produktet blev udført i form af en en-personsprojekt.

2.1 Case Beskrivelse

Ikke alle virksomheder har mulighed for at servere mad i deres kantiner fra deres egen køkken. Dermed er der økonomiske gevinster for en cateringvirksomhed ved at sælge færdiglavede retter til disse virksomheder. På sammen tid er det blevet mere udbredt blandt virksomheder at sælge deres produkter via internettet. Disse muligheder vil cateringvirksomheden FoodForAll A/S[1] gerne benytte sig af for at øge deres omsætning og øge antallet af kunder via en hjemmeside der er let at bruge, da de på nuværende tidspunkt kun tillader bestillinger via deres telefon besat af en enkelt ansat.

2.2 Problemformulering

Hvordan kan et system for catering blive opbygget, der tillader brugere at oprette sig, bestille bestemte retter med tidspunkt og lokation, samt tillade virksomheden at oprette valgmuligheder og se bestillinger?

[1] FoodForAll A/S i dette projekt er en fiktiv virksomhed og har intet med potentielle virkelige virksomheder at gøre.

2.3 Projekt Afgrænsning

På grund af tidsbegrænsninger og at produktet er en prototype, var de følgende begrænsninger sat:

- Intet betalingssystem.
- Det grafiske brugerflade vil være simpel, funktion over design.
- Sikkerheden vil minimal i forhold til et virkelig produkt.
- Der ville være mere fokus på User frontend/REST-API delen end Catering frontend/REST-API-delen, hvor Catering-delen ikke ville få en frontend.

Kapitel 3

Projekt Planlægning

Produktet blev planlagt den første svendeprøvedag og før der blev begyndt at arbejde på selve produktet. Det blev valgt at benytte et Gantt-diagram for tidsplanen, da der var kendskab til dette fra H5.
Hver tidsenhed i Gantt-diagrammet var en arbejdsdag og tiden blev fastlagt for de normalle arbejdsdage, der var fra opstart til aflevering (Det vil sige mandag til og med fredag), hvilket gav 19 arbejdsdage. Dog på grund af den sidste dag var afleveringsdagen blev det vedtaget, af udvikleren, at alt arbejde skulle være færdig dagen før, hvilken gav i alt 18 arbejdsdage. De forskellige aktiviteter er samlinger af flere kravspecifikationer[1], f.eks. indeholder Factories alle kravspecifikationer, som har kategorien 'Factory', men der blev prøvet at oprette en aktivitet for hver Id samling, f.eks. alle Cat-Service-n ligger under Catering Dataprocess projekt.
I forhold til placeringen af de forskellige aktiviteter, så var de primært i starten af projektet med den begrundelse at udvikleren fortrækker at havde travlt i starten og dermed havde tid til at bedre håndtere nye opgaver, forsinkelser, forbedringer og lignende længere henne i arbejdsprocessen.
Tidsplanen kan ses i billag A. Tidsplanen kan også findes under projektroden/Data og hedder Schedule.xlsx. Givet dette projekt var en en-personprojekt betød det, at der ikke var nogen fordeling af arbejdet.

[1] Kravspecifikationerne kan ses i Produktrapporten

Kapitel 4

Valgte Teknologier og Mønstre

Dette kapitel vil forklare hvad de valgte mønstre og teknologier, der blev benyttet, er og hvorfor de blev valgt. Andre teknologier vil også bliver nævnt, samt givet en forklaring på hvorfor de ikke blev valgt. Produktrapporten forklare hvordan de valgte teknologier blev implementeret.

4.1 Message Broker

Udvikleren valgte at benytte en message broker til at kommunikere mellem catering-delen og user-delen, da produktet var designet til at være opdelt i to afskilte dele, se produktrapport sektion 'Produkt Arkitektur'.
Det er mange grundene til at benytte en message broker, f.eks. at kunne opskaler de dele af opsætningen der er under pres, et eksempel kunne være hvis der er stor pres på catering-delen, så kan endnu en catering-processe startes op og begynde at håndtere data. På sammen tid kan data sendes til forskellige programmer, der har behov for data'en for forskellige grunden, uden at nogle af programmerne har kendskab til hinanden, hvilket betyder lav kobling mellem de forskellige dele. Selvfølgelig gør en message broker det mere besværligt at følge data igennem et system, da man ikke kan se hvad der modtager data eller sender data til systemet ved at kigge på koden.
Der blev kigget på to forskellige teknologier inde for dette, Kafka, se 4.1.1, og RabbitMQ, se 4.1.2. Det blev valgt at gå med RabbitMQ, da RabbitMQ var mere ukendt for udvikleren, samt at RabbitMQ virkede mere realistisk for en catering-virksomhed frem for Kafka. Udvikleren havde lidt kendskab til Kafka fra et tidligere forløb og lidt kendskab til RabbitMQ fra nuværende læreplads, men kun benyttet begge lidt.
Det var selvfølgelig ikke nødvendigt, at benytte en message broker til at kommunikere med andre dele. Hver del kunne havde deres egen REST-API, som kunne bruges til at kommunikere via. Delene kunne har været del af den samme kode-projekt og dermed kunne kalde nødvendige metoder direkte. Disse løsninger ville dog øge koblingen mellem de forskellige dele af produktet.

4.1.1 Kafka

Apache Kafka er en åben-kilde event streaming platform, hvilket betyder at den kan bruges til at publish (skrive) events og subscribe (læse) til events. Alle events bliver gemt i en eller flere cluster af server og har dermed mulighed for at stadigvæk fungere, hvis en server går ned, samt der er muligt at replikere data over flere server [6].
En lager service bliver kaldt en Broker i Kafka, men Kafka kan også have zookeepers[1], workers og mere alt efter behovet.
Kafka lager events under topics, events bliver aldrig slettet når de bliver læst, de slettes først når de bliver for gamle, hvilket en kafka-bruger kan indstille tiden på. På sammen tid læses events altid i den rækkefølge de blev indsat i. En subscriber kan vælge at kun læser events fra efter den har startet eller også de events før start-op.
Kafka tillader også schema for de oprettede topics, hvilket betyder at Kafka kan validere, at events, der sendes til den, passer ind i den valgte topic[6], hvilket betyder at Kafka kan anses for at benytte

[1] Efter [10], så er dette snart ikke længere en ting

Extract-Transform-Load, data er transformeret til et bestemt schema og derefter sendt til Kafka, som så gemmer det transformeret data.

4.1.2 RabbitMQ

RabbitMQ er en åben-kilde messaging og streaming broker, som benytter queues (kø) til at sende data mellem producers (skriver) og consumers (læser) og så snart en consumer har anerkendt, at de har modtaget en besked, bliver beskeden slettet fra dens kø. RabbitMQ benytter First-In-First-Out for beskederne i en kø [11].
RabbitMQ kan anses at benytte Extract-Load-Transform, da den ikke har noget schema for data'en i en kø, da data lægges som bytes, dermed kan alt skrives og alt kan læses til/fra en kø, hvilket betyder at consumer'en skal tjekke om data'en opfylder krav, f.eks. om den kan mappes til et bestemt objekt eller ej.
Med sin standard-opsætning benyttes der round-robin, hvilket betyder, at hvis der køre en consumer, så modtager den alt data, der sendes på den kø den lytter til, hvis der er to consumers, så modtager de halvdelen hver osv. Dette er fint, hvis alle consumer gør det samme, men hvis consumerne gør forskellige ting, så kan der opstå problemer. Det er dog muligt, at opsætte RabbitMQ til at sende en event til at alle consumers [11].
RabbitMQ tillader også opsætningen af Remote Procedure Call (RPC), hvilket er, at en producer kan modtaget et svar, for en sendt besked, fra den consumer som håndtere beskeden [12].

4.2 ORM

Den valgte ORM (Object-Relational Mapper) for dette projekt er EntityFramework Core 8.x. Denne ORM er udviklet af Microsoft og er den primære ORM inde for C♯.
Formålet med en ORM er, at tillade lettere benyttelse af forretnings-baseret modeller og en database ved at havde ORM'en som et mellemled, der står for at mappe mellem et objekt-model og en eller flere tabeller.
EntityFramework Core har en del sikkerhed bygget ind i sig, f.eks. automatisk sanitising af data og validering af parameter for at undgå SQL injection angrib. På sammen tid kommer EntityFramework Core med Repository Pattern, se 4.11.2, og Unit Of Work, se 4.11.3, men i dette projekt, er der egen implementationer af disse. Grunden til dette er mere fin-kontrol over database-adgang end hvad Entityframework Cores egen udgaver tilbyder.
EntityFramework Core tillader også opsætningen af hvordan data i en model burde mappes til tabel-data og tilbage.
Andre valgmuligheder kunne har været, at udvikle egen ORM eller oprette og sende SQL-kommandoer via C♯s SqlClient. Disse blev ikke valgt, da disse ville tage for lang tid at udvikle og ville ikke give en bedre løsning end i forhold til EntityFramework Core.

4.3 Database

For at sikre at alt oprettet data blev gemt for senere behov, blev det valgt at inddrage en relationel database i form af en SQL database. En relationel database tillader oprettelse, hentning, sletning og ændre af data på en overskuelig måde, samt at havde data-schema over de forskellige typer af data, der bliver gemt. En SQL database har en data-schema for hver tabel i databasen, hvilket betyder at hver kolumne svare til en bestemt datatype og hver rækker er et bestemt objekt. Som navnet angiver er der relationer mellem det gemte data i databasen.
På sammen tid tillader en SQL database formindskning af duplikereret data, da der kan oprettes relationer mellem de forskellige tabeller. SQL databaser tillader også at opsætte, begrænset, regler for hvordan data'en skal se ud, f.eks. om nullable data er tilladt eller om bestemt data må være duplikeret i flere rækker i en tabel.
Den valgte database er MSSQL, også kendt som Microsoft SQL, database. Grunden til dette valg, var fordi udvikleren havde godt kendskab til den, både fra egen udvikling og fra lærepladser. På sammen tid benyttes EntityFramework Core til at håndtere kommunikationen med databasen og dermed vil der for kode-udvikling ikke være en større kode-forskel.
Andre muligheder kunne f.eks. være MySQL eller SQLite, SQLite er en let og gratis database med

få funktionerne i forhold til MSSQL og MySQL. MySQL er en gratis database, MSSQL koster for ikke-udviklere.
En vigtig forskel, der er vigtigt, når der overvejes database, er hvor godt databasen understøtter de valgte datatyper for modellerne i produktet, f.eks. om en database understøtter ikke-UTC tidspunkter, om den understøtter float/single eller om den ikke gør. F.eks. understøtter visse udgaver af MSSQL ikke TimeOnly og DateOnly, da disse er nyere typer inde for C♯. Detter er en anden grund til valget faldt på MSSQL, den er udviklet af Microsoft, som også udvikler C♯, og dermed er der større chance for nyere udgaver af MSSQL vil understøtte nye datatyper i C♯.

4.4 Docker

Docker er en teknologi, der tillader at køre applikationer uden at de bliver påvirker af deres værts miljø. Docker virker ved at køre en applikation i en lukket enhed kaldt en container, som er bygget ud fra et Docker Image. Et Docker Image angiver normalt OS'en, som container skal benytter, og hvad end der ellers skal til for at køre applikationen. Dette burde betyde, at containen på en computer vil køre på sammen måde som på en anden computer, da containerne kører på et lukket system, adskilt fra vært-computerens system. Det, der køres inde i container, vil normalt blive påvirket af systemet på værten på sammen måde som i virkligheden, dette ville f.eks. være igennem åbne porte, som applikationen i container lytter til. På sammen tid kan flere applikationer køre på den samme vært, selv hvis de i virkeligheden benytter den samme port, hver Docker mapper den interne port til en anden port på værten og dermed slippes der for at ændre på applikationsopsætningen. Det er også muligt at mappe en eller flere mapper i en container til mapper på selve værten og dermed kan data deles via disse mapper, hvilket betyder, at hvis containeren bliver slette og genoprettet med kendskab til værtsmapperne, så vil intet data blive mistet.
Andre løsningsmuligheder var, at køre servicerne direkte på systemet, hvilket Seq, RabbitMQ og MSSQL kunne gøre, men dette kan skaber problemer, da systemet kunne påvirke hvordan servicerne opføre sig, samt gøre det mere svært at replikere opsætningen hos andre udviklere. Ellers kunne det være muligt, at havde en test-setup som alle udviklere forbundet til, men dette betyder at en udvikler kan påvirke en anden udvikler, hvilken kan skabe problemer.
En anden grund til at benytte Docker, er visse services, som Kafka, kræver flere enheder for at køre optimalt, noget man kan undgå via Docker, da hver Kafka-enhed har sin egen container.
Det skal peges ud at Docker containers ikke burde blive påvirket af selve værtens OS (Udover for Windows's Hyper-V for windows-platformer), men det blev fundet at Seq containeren, som ikke brugte standard brugeren, kørte anderledes på en Windows 11 pc end i forhold til en Windows 10 pc. En grund kunne ikke findes, der forklarede dette.

4.5 API Framework og REST

Frameworket, der benyttes til API'erne, i dette produkt er C♯ASP.Net Core. Dette er et framework Microsoft har udviklet for dotnet-produkter som REST-API'er og MPA'er. Et REST-API (Representatinal State Transfer) er en applikation arkitektur, der bruges meget inde for netudvikling. Normalt benyttes der HTTP-metoderne (F.eks. GET, POST, PUT) og URL-stien til at kontakte bestemte endpoints i et REST-API. REST-API sender/modtager kun de ressourcer, som klienter har behov for og beder om.
Modellerne, der benyttes af et REST-API, er enten en request eller response model. Request modellerne er for data der bliver sendt ind, normalt via Body'en på http-requesten, hvorimod response modellerne bruges for det data der bliver sendt ud. Data der sendes ind kan, som nævnt, blive sendt via en body, men de kan også sendes ind via query-delen af URL'en. ASP.Net Core tillader let opsætningen om hvorvidt data skal komme fra body eller query, dog er der visse krav fra HTTP, der giver mening at overholde, f.eks. at DELETE ikke kan indeholde en body og dermed skal ekstra data sendes via query.
I ASP.Net Core opdeles Endpoints efter hvilken Controller[2] de ligger i.

[2]En controller er en klasse med det formål at kunne kontaktes via HTTP(S)

4.6 Frontend Framework

Det valgte framework for frontend delen er Blazor, et framework udviklet af Microsoft, som benytter C♯, HTML og CSS til udvikle frontend web klienter med. Blazor benytter Microsofts Razor syntaks for at indsætte data/funktioner ind i HTML og manipulation af Document Object Model.
Blazor er et nyere frontend framework og har to udgaver, Blazor Web App og Blazor WebAssembly Standalone App. Forskellen mellem disse to er, at Web App er en Multi Page Application (MPA), hvor hver side hentes fra serveren, men data- og sidepåvirkninger (som ikke kræver mere data) udføres hos brugeren, f.eks. filtering af hentet data kan udføres af C♯kode uden behov for at kontakte serveren eller JavaScript. WebAssembly Standalone App er kørt 100 % hos brugeren, det vil sige rent klientkode, og er en Single Page Application (SPA), dermed er der intet kontakt med en server, når der udføres f.eks. sideskift. Pga. WebAssembly Standalone App er ren klientkode, betyder det at den ikke har mulighed for at snakke med en datakontekst direkte (Web App kan gøre dette, da kommunikationen foregår via dens server) og dermed er det nødvendigt at udvikle et API, som den kan kommunikere med, hvis frontend'en har behov for dynamisk data.
En anden mulighed for frontend framework kunne være Angular 2. Angular er udviklet af Google og benytter TypeScript i forhold til Blazors C♯.
Grundene til at Angular ikke blev valgt, var primært for at prøve en ny teknologi i form af Blazor, da Angular er udviklerens primært frontend framework. Angular har dog den fordel at være et ældre, dermed mere udviklet, sprog med mindre chance for bugs og exploits, samt mere/bedre dokumentation og vejledninger.

4.7 Test

Tests, både manuelle og automatiske, er vigtige for udviklingen af et produkt, da de tjener det formål, at fremvise at produktet overholde de krav, der er givet til produktet. På sammen tid tillader de finde fejl, før produktet bliver udgivet, og tillader udviklere at se om deres kode både fungere og ikke har stoppet andet kode med at virke.
Der findes forskellige slags tests, som Unit Tests, Integration Test, End-To-End Test og Acceptance Test.
En Unit Test tester den mindste del af noget kode, f.eks. hvis en metoder modtager en streng, om den håndtere en null-streng eller om en metode bruger over n tidsenheder før den returnere.
Integration Test tester om flere moduller virker korrekt sammen, f.eks. om Endpoint A på en kontroller kalder service B, som så sender besked C til service D.
End-To-End Test tester flere systemer på sammen tidspunkt, f.eks. om system A kan kommunikere med system B igennem message broker C.
Acceptance Test bruges til at teste om større dele af et eller flere system overholder deres krav og kan indeholde ikke-automatiskeret tests, som f.eks. om en bruger kan forstå GUI'en.
En vigtig ting med test er, at få teste alle stier igennem et stykke eller flere stykker kode for at sikre sig at alt virker, se 4.1 for et simple eksempel på hvorfor. Det givet eksempel fejlede, hvis den ikke kunne skabe en User, hvilket var overset før koden blev manuelt testet.

Listing 4.1: Kode der kan give undtagelse

```
1     public async Task<Result<UserAuthResponse>> CreateUserAsync(UserCreationRequest
          request)
2     {
3         var userData = await _unitOfWork.UserRepository.AllAsync(new UserDataQuery
              ());
4         UserValidationData uvd = new(userData);
5         var result = _userFactory.Build(request, uvd);
6         if (!result)
7             throw new NotImplementedException()
8         var user = result.Data;
9         _unitOfWork.UserRepository.Create(user);
10        _unitOfWork.Commit();
11        var comResult = _communication.TransmitUser(user);
12        var authResult = await _securityService.AuthenticateAsync(new
              UserLoginRequest() { UserName = request.CompanyName, Password = request
              .Password });
```

```
13          return authResult;
14      }
```

For dette projekt, er der lagt primært fokus på Integration Test og Unit Test, da disse er lettest at fremvise og automatiseres.

4.7.1 Automatiskeret Test Frameworks

Inde for C♯ findes der tre primære test frameworks, disse er MSTest, NUnit og XUnit. Udvikleren har benytte alle tre og fundet XUnit til at være en godt mellem punkt i mellem de to andre og derfor valgt at benytte XUnit.

4.8 Domain Driven Design

Domain Driven Design (DDD) er en arkitektur, som har fokus på domænerne og styrer udviklingen af produktet en del. Domain driven design f.eks. lægger en del, hvis ikke alt, forretningslogik ind i selve modellerne frem for f.eks. i en Business Logic Layer, hvilket betyder at logikken ikke vil blive spredt over produktet, samt at modellen styre alle ændringer på sig. F.eks. kan en model tjekke om data kan blive tilført eller fjernet fra en samling i sig [4].
Domain driven design benytter det der bliver kaldt en Aggregatrod, hvilket betyder at alle ændringer, oprettelser og fjernelsen af data foregår igennem et enkel punkt, selv andre objekter kan kun blive påvirket igennem roden [3].
Domain driven design har to vigtige begraber, bounded context og domain, domæne er et dækkende område inde for en forretning, hvorimod bounded context bruges til at opdele modeller [5]. I dette produkt, er der to domæner, User og Catering, hvor modeller for f.eks. User og Customer har den samme primære-nøgle, men nogle af deres værdier er forskellige, User har kendskab til dens Refresh Token og Customer har kendskab til ordre.
Domain driven design har også Value Object, hvor vigtigheden ligger i dens værdier frem for sin reference. F.eks. to value objects med de samme værdier ville anses for at være det samme objekt, da de ikke har en identitet, hvilket ikke er sandt for reference objekter, som altid har identiteter. På sammen tid kan værdier i en value object ikke overskrives, hvis der er ændringer overskrives hele objektet på en gang [2]. Det er muligt for et value object at indeholde forretningslogik [3].
Domain driven design er dog tungt og kan tage lang tid at implementere korrekt, da der er behov for domæne-eksperter for at kunne implementere de forskellige domæner korrekt, samt at domain driven design er meget komplekst, hvilket betyder at det nødvendigvis ikke er den bedste arkitektur for mindre løsninger, som f.eks. denne svendeprøve.
Domain driven design blev valgt, da udvikleren godt kan lide det og det passer godt sammen med andre mønster som Command Query Responsibility Segregation, se 4.11.1.
En ting med domain dreven design er, at forskellige modeller kan kun have kendskab til andre rødder[3] og dette burde helst være igennem en reference id frem for selve objektet [3]. Dette har det formål at forhindre at en aggregat kan påvirke en anden aggregat, men dette betyder at der ikke findes direkte reference til andre objekter og dermed miste en SQL-database lidt af sit formål med at kunne holde styr med relationerne. Data i en aggregat kunne pege på en anden rod der ikke længere findes, dermed er det meget vigtigt at produktet får slettet, ændret og tilført data korrekt. På sammen tid kan domain driven design også øge mængden af duplikeret data, da objekterne under en rod skal være gyldige og dermed kan visse data være nødsaget til at blive duplikeret. I dette produkt kan det ses med MenuPart og Dish, hvor MenuPart har værdien Name duplikeret fra Dish, grunden er, at undgå for meget kommunikation med databasen, men hvis en Dish ændre navn, skal alle MenuParts der peget på den findes og opdateres. Test der køres over mere eller hele løsningen, som Intergration Test, er dermed vigtige for at tjekke om alt data sættes korrekt, da en Unit Test vil nødvendigvis ikke fange det.

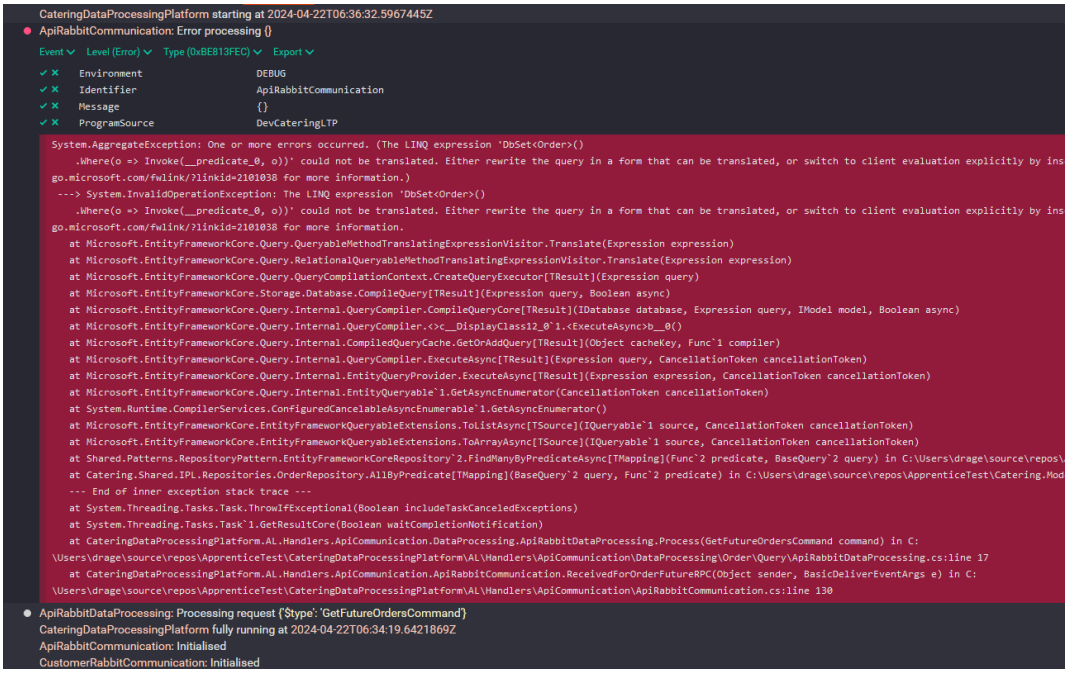
[3] Det vil sige at et objekt i en aggregat kan ikke havde kendskab til objekter i en anden aggreat ud over roden.

4.9 Logging

For dette projekt, blev det besluttet at logge data fra produktet og sende dem til en tredje-part system. Valget faldt på Seq. Seq benytter strukturet logs og kan hjælpe med at analyser og finde information fra indsendte logs. F.eks. kan alle Error logs for et bestemt tidspunkt fremhentes eller alle logs for et bestemt tidsperiode. Dette betyder, at det er lettere at finde fejl i software, mens det køre på live- eller testmiljø, samt at se hvad og hvor meget der forgå inde i de forskellige systemer. Seq gemmer alle logs i form af JSON og dermed tillader Seq let at søge i data, selv i komplekse logs, da Seq kommer med dens eget query sprog [7].

Figur 4.1: Seq dashboard

Grunden til at valget faldt på Seq, var primært fordi udvikleren allerede havde en smule kendskab til Seq og hvordan man indsendte data til det fra lærepladsen, men ikke rigtigt noget dybt kendskab til hvordan man brugte Seq og fik mest ud af det. En anden mulighed kunne har været Grafana, som også tillader logs og søgning/fremvisning af dem, men Grafana kan en masse mere og for en catering-virksomhed ville noget mindre, som Seq, nok give mere mening.



Figur 4.2: Seq fejl log

4.10 Konfigurationsstyring

Dette projekt benyttede Git som konfigurationsstyring, hvilket er et meget almindelig Version Control System (VCS). Git tager snapshots af hvordan filerne, den spore, ser ud, når en commit bliver udført, hvorimod andre muligheder, som CVS, gemmer forskellen fra sidste gang filen blev ændret[9]. Dermed slipper Git for at udregne hvordan filerne så ud i et bestemt commit ud fra alle tidligere commits, da

den direkte kan hente filerne.
Selve Git repository'et for produktet blev gemt på GitHub, hvilket er et normalt lager for Git repositories.
Grunden til at Git blev valgt, var fordi udvikleren havde en del kendskab til det og alle lærepladser, udvikleren havde været på, benyttede Git. I forhold til GitHub, så er det muligt at havde egen hoste sider for Git repository lager, men det blev ikke anset for at være nødvendigt for dette produkt.

4.11 Mønstre

Denne sektion vil forklare de valgte mønstre, hvorfor de blev valgt og plotentielle andre muligheder.

4.11.1 Command Query Responsibility Segregation

Command Query Responsibility Segregation (CQRS) er et mønstre, der opdeler en model i to dele, en skrivemodel og minimum en læsemodel. Retfærdiggørelsen for CQRS er, at en kompleks læse/skrivemodel kan være problematisk at benytte [1]. CQRS øger kompleksiteten i produktet, da der skal udvikles flere modeller, men der er bedre opdeling af hvad modellerne gør, samt at der undgås at unødvendige data sendes igennem systemet.
Command-delen af CQRS bruges til at udføre en ændring på en model og skal i princippet kun påvirke en enkel aggregatrod. Query-delen står for at sende forespørgelse til model(en/erne) med det formål at hente data fra dem og skulle aldrig ændre på gemte data [1].
Efter [1], så kan CQRS være brugbart i Domain Driven Design, men det behøves ikke at være brugt på hele systemet.
Gruden til at dette mønstre blev benyttede i dette projekt, var for at formindske koblingen mellem læse- og skrivedelene af koden, hvor læsemodellerne var i princippet DTO'er[4], hvor mappingen skete meget tidligt.
En anden valgmulighed, for at mappe data, kunne har været en auto-mapper som Mapster, men i forhold til det implementeret CQRS, så anså udvikleren Mapster til at være mere besværligt at håndtere, når kode blev refactoret, da Mapster ikke automatisk informere om casting-problemer.

4.11.2 Repository Pattern

Repository Pattern er et mønstre, som bruges til at styre hvad slags kommunikation, der kan foregå mellem datakonteksten og resten af systemet. F.eks. kan det sættes op til at kun kunne læse fra bestemte tabeller eller skrive til bestemte kolumner i en tabel. Mønstret tjener dermed formålet, at informere udvikleren med hvordan de kan interagere med datakonteksten.
Den valgte ORM, EntityFramework Core, se 4.2, har dette mønstre indbygget i sig og dermed er det ikke nødvendigt at implementerer mønstret selv. Det blev dog valgt at implementerer egen udgave for at lægge et lag mellem EntityFramework og resten af systemet, for at formindske koblingen og letgøre udviklingen af automatiske test. På sammen tidspunkt, tilladte selv-implementeret repositories at have bedre styr over hvordan udviklere kommunikerede med datakonteksten ved at ensforme kodeskrivningen.

4.11.3 Unit Of Work

Unit of Work er et mønstre med det formål at holde styr på alle ændringer til datakonteksten og overføre disse ændringer på en gang til datakonteksten [8].
EntityFramework Core har dette mønstre indbygget i sig, men det blev valgt at implementere egen udgave for at havde lidt mere styr over ting. Dette tilladte også interface og dermed blev koblingen formindsket.

4.11.4 Result Pattern

Result Pattern er et lille og let mønstre, som tillader en metode at sende forskellige slags data tilbage uden at skulle benytte exception handling, ikke at have klasser der har properties for alle muligheder

[4]DTO står for Data Transfer Object.

eller keywords som `ref` og `out`. Result Pattern kan på sammen tid opsættes til at passe produktets behov. F.eks. hvilken slags fejlbeskeder, hvis nogen, der kan blive sat, hvilken slags Results kan benyttes (f.eks. NotFoundResult eller SuccessResult) og mere.

4.12 Software Udvikling Framework

Inden for software udvikling findes der nogle forskellige framework, f.eks. Waterfall og agile metoder som SCRUM, KANBAN eller Crystal. Disse framework benyttes til at hjælpe med udviklingen af et produkt.
For dette produkt, blev intet software udvikling framework valgt pga. den korte tidsgrænse og at udvikkleren følte, at de kunne holde styr nok på opgaverne uden et framework. Udvikleren benytter normalt SCRUM med to uger sprint. Hvis produktet skulle udvikles for et rigtig virksomhed, så ville enten SCRUM eller Waterfall benyttes, alt efter hvad virksomheden helst ville havde. Waterfall er en lineær produkt-udvikling framework, hvorimod SCRUM, samt resten af agile, benytter iterationer til at udvikle produktet.

Kapitel 5

Realiseret Tidsplan

Den realiseret tidsplan er i billag A. Farverne betyder det følgende:

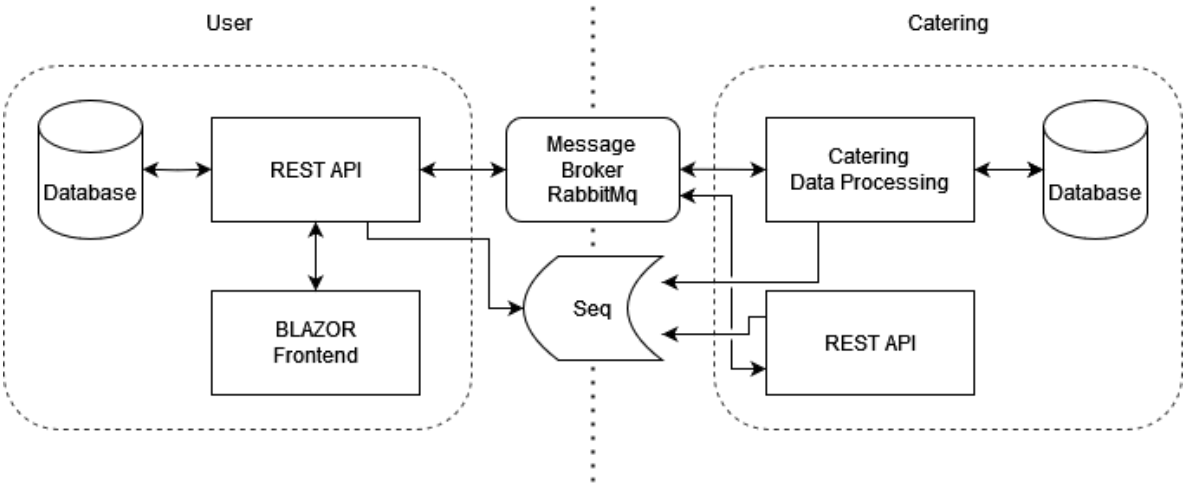
- Lilla - Aktivitet udført inden for afsat tid.
- Gul før lilla - Aktivitet startet før afsat tid.
- Gul efter lille - Aktivitet afsluttet efter afsat tid.
- Prikkede - Aktivitet blev udført hurtigere end forventet eller startede senere end forventet.
- Rød - Aktivitet fjernet

Som det kan ses, så blev de fleste af aktiviterne udført inden for planlagt tid og nogle få blev enten startet lidt tidligt eller tog lidt længere tid end forventet. Aktiviteten 'User Dataprocess projekt' blev startet og færdiggjort flere dage før forventet start.
Aktiviterne 'Catering REST API', 'Catering Dataprocess update', 'Kravspecifikationer update' og 'Testspecifikationer update' blev indsat senere i udviklingen, da et REST-API for Cateringen ikke opringelig var en del af planen.
Den røde aktivitet 'Catering Dataproces projekt' blev ikke udført, da den var en duplikation af en anden aktivitet, som ikke var blevet set, da den blev skrevet. Aktiviteten 'Database og context - Catering' tog nogle flere dage end forventet, det var pimært pga. problemer med at få Value Objekter i EntityFramework Core til at virke og få konsol-programmet til at kunne benytte datakonteksten. Frontend-delen gik hurtigere end forventet, men dette kom fra at udvikleren ikke rigtig havde kendskab til Blazor og dermed var der en del usikkerhed på hvor tid det ville tage, f.eks. startede frontend'en med at være Blazor server-side, da udvikleren ikke viste hvad forskellen mellem de to Blazor projekt-typer i Visual Studio 2022 var. Aktiviteterne vedrørende RabbitMQ blev også overestimere lidt, da udvikleren ikke rigtig havde arbejdet så meget med den message broker. På sammen tid havde udvikleren en vane med at overestimere hvor meget tid end opgave vil tag, da udvikleren havde tit oplevet at et eller andet uforudset dukkede op under aktivitet-arbejdet før.
På grund af hvordan udvikleren fortrækker at arbejde på, blev flere aktiviteter normalt arbejdet på på sammen tid.
Når der kigges på tidsplanen, ser det ud til at alt udviklingen, med undtagen af 'Sikkerhed' og 'Unit test', var færdig dag 10. Dette passer ikke helt, da udvikleren til tider lavede nogle små ændringer, som forbedringer, i koden hist og her efter dag 10, hvilket ikke kan blive set uden at kigge på Git commits eller dagbogen. F.eks. blev der lavet ændringer til RabbitMQ implementationen for aktiviteten 'Catering Dataprocess projekt' lang tid efter dens aktivitet var færdig og Gantt-diagrammet tillader ikke tidsperioder uden arbejder i en aktivitet, dermed burde der nok være oprettede nogle aktiviteter for kode-ændringer.

Kapitel 6

Diskussion

Det færdige produkt bestod af to hoveddele, User og Catering, samt et datalager for logs og en message broker for kommunikation mellem de to hoveddele, se figur 6.1.
I forhold til kommunikation med databaserne, foregik dette på forskellige måder i User- og Catering-delen. I User er det via REST-API'et og i Catering er det via konsol-programmet, Catering Data Processing. Grunden til dette, var fordi Caterings REST-API kom meget sent i udviklingen. Hvis der havde været mere tid, ville User-delen har fået sit eget konsol-program med det samme formål som hos Catering, men på sammen tid fremviste produktet forskellige måder at gøre ting på. Caterings REST-API havde ikke noget sikkerhed, hvorimod User REST-API benyttede JWT som sikkerhed.
Ingen af projekterne i produktet kommunikeret direkte med hinanden. Alt kommunikation foregik enten via HTTPS eller via message brokeren. Frontend'en kommunikeret med User REST-API'et, da



Figur 6.1: Løsningen

pga. sikkerhedsmæssige årsager ikke kunne gives adgang til RabbitMQ i frontend-delen. Begge REST-API'er kommunikeret med konsol-programmet via message brokeren. Dette betød at de to REST-API kunne kommunikeret med hinanden uden at de viste det.
I forhold til automatiske test, så var der kun nogle få, nok til at fremvise forskellige slags test, fra metoder til factories til endpoints, og testopsætninger. I et færdigudviklet produkt, ville der være mange flere tests, men mængden af udviklingtid begrænsede mængden af test. Der var ingen test, der benytte RabbitMQ. I princippet burde enten hver test med RabbitMQ havde sin egen message broker for at undgå at testene kunne påvirke hinanden eller en enkel RabbitMQ test blev udført af gangen.
Kravspecifikationerne kunne har fået bedre id'er. F.eks. Cat-Service-7 og Cat-Service-1 begge indikerede at de er den samme service, men 7'en er OrderService og 1 er CustomerService.
Repo-2 og Repo-1-b burde nok ikke har været krav, siden de var mere omkring måder at implementere nogle af de andre krav. For test sagerne, så kunne der har været en del flere sager. Der var ikke noget for kommunikation med databaserne og ingen RabbitMQ tests. På sammen tid dækkede visse test

sager, som Cat-Factory-1-a, over flere tests, som ville teste det samme koden, men med forskel slags data og ville har været opdelt i flere tests i test projekterne. Der skulle nok har været en test sag for hver eneste sti igennem koden, der kunne give forskellige resultater.

Kapitel 7

Konklusion

Det færdigudviklede produkt løste problemformulering og casen. Produktet bestod af en kunde single page application frontend, to REST-API'er og et konsol program. På sammen tid, var der et datalager for logs i form af Seq, to MSSQL-databaser og en message broker i form af RabbitMQ.
Produktet tillod virksomheder, at oprette dem selv som kunder og bestille ordre til deres adresse, samt at de kunne se deres ordre. FoodForAll A/S havde muligheden for at oprette nye menuer, som det var nødvendigt, og se ordre, alle og fremtidige.
Frontend'en design skulle nok har været forbedret for at optimere brugeroplevelsen, men dette var et projekt afgrænsning.
Produktet kunne havde brugt nogle flere automatiskeret tests, hvis produktet blev videreudviklet, samt en frontend del for FoodForAll A/S interne REST-API. På sammen tid var der ikke noget betalingsmåde opsat, endnu et projekt afgræsning, men ville være nødvendigt før produktet kunne tages i brug i virkeligheden.
Sikkerhedsmæssigt var der mangler, før produktet ville kunne tages i brug, primært med brugerne til databaserne, men også for det interne REST-API'et.
Dermed kan det konkluderes, at det færdigudviklede produkt, for svendeprøven, løste problemformuleringen, men for en rigtig virksomhed, ville produktet være en tidlig prototype og krævede mere udvikling.

Referencer

[1] M. Fowler. "CQRS." (2011), webadr.: https://martinfowler.com/bliki/CQRS.html (hentet 16.04.2024).

[2] M. Fowler. "Value Object." (2016), webadr.: https://martinfowler.com/bliki/ValueObject.html (hentet 17.04.2024).

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

[4] Abp. "Domain Driven Design." (2021), webadr.: https://docs.abp.io/en/abp/latest/Domain-Driven-Design (hentet 15.04.2024).

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

[6] Apache. "Apache Kafka." (), webadr.: https://kafka.apache.org/ (hentet 15.04.2024).

[7] Datalust. "Seq." (), webadr.: https://datalust.co/seq (hentet 23.04.2024).

[8] M. Fowler. "UnitOf Work." (), webadr.: https://martinfowler.com/eaaCatalog/unitOfWork.html (hentet 15.04.2024).

[9] Git. "Getting Started - WHat is Git?" (), webadr.: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F (hentet 25.04.2024).

[10] A. Kafka. "Documentation." (), webadr.: https://kafka.apache.org/documentation/#zk_depr (hentet 23.04.2024).

[11] RabbitMq. "RabbitMQ." (), webadr.: https://www.rabbitmq.com (hentet 15.04.2024).

[12] RabbitMq. "RabbitMQ tutorial - Remote procedure call (RPC)." (), webadr.: https://www.rabbitmq.com/tutorials/tutorial-six-dotnet (hentet 15.04.2024).

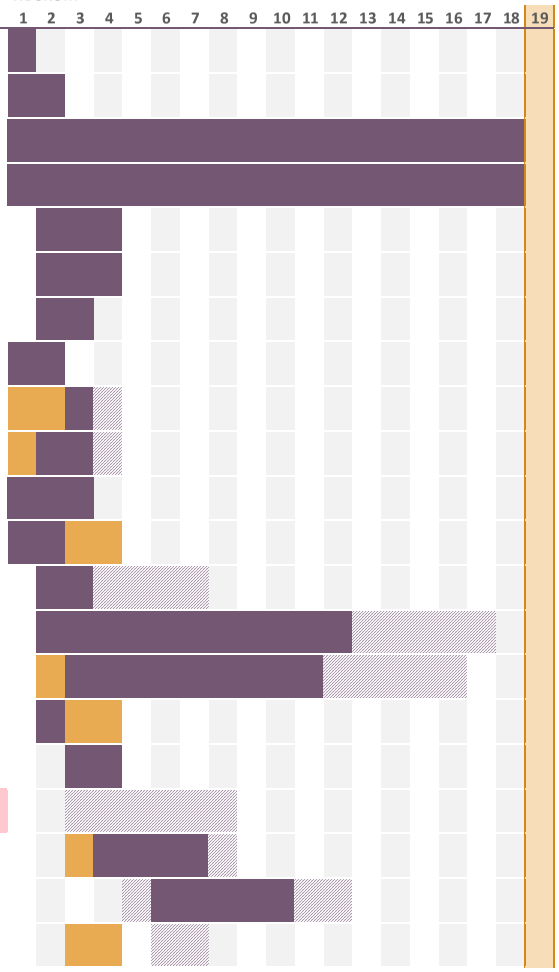
Projektplanlægger

Fremhæv et tidsrum til højre. Derpå følger en forklaring af diagrammet.

Fremhævning af tidsrum: 19

Varighed for plan | Faktisk startdato | % fuldført | Faktisk (ud over plan) | % fuldført (ud over plan)

AKTIVITET	STARTDATO FOR PLAN	VARIGHED FOR PLAN	FAKTISK STARTDATO	FAKTISK VARIGHED	PROCENTDEL FULDFØRT
Case beskrivelse	1	1	1	1	100%
Opret tidsplan	1	2	1	2	100%
Process rapport	1	18	1	18	100%
Produkt rapport	1	18	1	18	100%
Kravspecifikationer	2	3	2	3	100%
Testspecifikationer	2	3	2	3	100%
Docker Images	2	2	2	2	100%
Shared Projekt	1	2	1	2	100%
Catering Dataprocess	3	2	1	3	100%
Repositories	2	3	1	3	100%
Factories	1	3	1	3	100%
Result Pattern	1	2	1	4	100%
CQRS	2	6	2	2	100%
Sikkerhed	2	16	2	11	100%
Unit test	3	14	2	10	100%
Database og context - Catering	2	1	2	3	100%
Database og context - User	3	2	3	2	100%
Catering Dataproces projekt	3	6			0%
User API projekt	4	5	3	5	100%
User Frontend projekt	5	8	6	5	100%
User Dataprocess projekt	6	2	3	2	100%



Bilag A Tidsplan

AKTIVITET	STARTDATO FOR PLAN	VARIGHED FOR PLAN	FAKTISK STARTDATO	FAKTISK VARIGHED	PROCENTDEL FULDFØRT	TIDSRUM 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Catering REST API	8	3	8	3	100%	
Catering Dataprocess	8	3	8	3	100%	
Kravspecifikatione r update	9	1	9	1	100%	
Testspecifikatione r update	9	1	9	1	100%	

got 19 days in total
aiming for 18 days
total used to have some breathingroom

Bilag B

Dagbog

B.1 9/4/24

Startet på hovedforløb 6.
Skrevet problemformulering og case.
Oprettet projekt og indsat basisk struktur med nogle modeller.
Net gik ned, typisk.
Glemte at lave git init først.
Net kom tilbage, lavet commit.
Lavet tidsplan udkast.

B.2 10/4/24

Planen er at få skrevet krav- og testspecificationerne for Catering-delen af produktet. Catering-delen skal have lavet sine modeller færdige, sine factories og datacontext og adgang. Shared projektet skal have interfaces og implementationer for CQRS, Resultpattern og repository pattern. På sammen tid skal der laves docker images for MSSQL, RabbitMQ og Serilogger. Ellers skal der skrives på rapporterne.
Fik lavet det der var planlagt.

B.3 11/4/24

Planen er at få lavet rabbit-delen for catering. Få skrevet mere rapport. Databasen for User-delen og dataadgang. Factories for dens modeller.
Fik ikke skrevet rapport, da EntityFramework Core var problematisk. Den nye måde at gøre Value Object på viste sig at være lidt besværligt. Det tog noget tid at kunne lave migations og apply dem til en database via et konsol program, da udvikleren ikke havde kendskab til dette i forvejen.

B.4 12/4/24

Målet i dag er at få færdiggjort krav- og testspecifiktionerne. Få begyndt på REST-API'et og user-data-processing-delen.
Fik færdiggjort krav-og testspecificationerne, kan godt være der kommer noget mere på et tidspunkt.
User-data-procescing-delen virker til at fungere som det skal.
Fået begyndt på REST-API'et og tilført en ny kontekst-model for refresh-tokens.

B.5 15/4/24

Færdiggør user REST-API, forbedring til RabbitMQ, få skrevet på rapporterne og lavet powershell script for datakontekst ændringer. Primære arbejde kommer nok til at ligge at få JWT til at virker.
REST-API anses for at være færdig for nu, ting vil nok komme som arbejdet på frontend-delen starter op. Powerschell script lavet. JWT tog mindre tid at indføre end regnet med. Den planlagte ændring til RabbitMQ gik fint. Valgte at indsætte endnu en RPC, drillede lidt at få den til at virke, forkerte consumer modtog data'en. For rapporterne var det primært process-rapporten der blev skrevet på.

B.6 16/4/24

Startning på frontend delen. Opsætning af httpclient for kommunikation med User REST-API'et. Læsning af hvordan Blazor skal udføres.
Sikkerhed på Blazor har vist sig at være en del problematisk, da fundet dokumentation for sikkerhed har været af ringe kvalitet. JWT kan dog blive hentet og gemt for senere brug. Skulle har gået med Angular, der er i det mindste dokumentation som ikke undlager en masse. Lige nu virker det til at serveren gemmer JWT'en dog, hvilket ikke er det bedste.
Valgte at prøve Blazor WebAssembly Standalone App frem for Blazor web App for at se om det ville hjælpe.
Har valgt at holde en pause med sikkerhed implementation på Blazor indtil alt andet i Blazor er færdiggjort.
Blazor WEbAssembly Standalone App har vist sig at være bedre i forhold til lagering af JWT'en. Frontend-delen kan nu næsten kontakte alle endpoints i API'et (mangler logud).

B.7 17/4/24

Planen er at skrive på rapporterne, da de var længere bagud end i forhold til kodningen.
Fik skrevet en del produktrapport. Fik også fjernet det gamle Blazor web-server projekt og lavede nogle få ændringer til noget kode.

B.8 18/4/24

Vil forsøge at få implementeret korrekt sikkerhed på Blazor igen.
Viste sig at implementere sikkerhed var let nok, når man ikke blandede Server-side og WebAssembly sammen. Fik indført at en bruger kan opdateres deres lokation.
Begyndte at lave et REST-API for kunden FoodForAll A/S, således at de kan indsætte menuer og det.
Vil opdater krav- og testspecifikationerne i morgen.
Fiksede nogle problemer med nogle LaTex commands i Produktrapporten.
Fik implementeret unhandled exception handling i CateringDataProcessing, samt lukning af konsol-programmet. Fiksede to fejl i contexthandler.ps1

B.9 19/4/24

Planen er at skrive krav- og testspecifikationerne for det nye REST-API. Når det er udført, så udvikle videre på API'et.
En kort forstyrrelse af arbejderelateret ting.
RabbitMQ kommunikationsdelen er klar, det meste af REST-API'et skrevet.

B.10 22/4/24

Få skrevet resten af REST-API'et og skrevet flere automatiske tests.
Fik skrevet resten af API'et og nogle tests der fremviser forskellige ting.

B.11 23/4/24

Planen er at skrive på rapporterne.
Fik skrevet en del på processrapporten og i selve projektet indført en middleware for REST-API'erne, der logger endpoint kald. Fik lavet en system diagram.

B.12 24/4/24

Få skrevet på produktrapporten.
Fik skrevet en del og lavet nogle diagrammer og tabeller.

B.13 25/4/24

Få skrevet endnu mere på produktrapporten.
Produktrapporten har fået hele sin udkast og alle sine diagrammer og billeder. Fået læst noget grammatik og det på rapporten, samt overført noget tekst til processrapporten.

B.14 26/4/24

Få skrevet på processrapporten.
Blev færdig med første hele udkast for processrapporten. Ændrede navnene på nogle projekter i produktet.

B.15 29/4/24

Renskrivning af rapporterne.
Fik renskrevet på begge rapporterne og indført noget mere tekst i processrapporten. Føler processrapporten er lidt for kort, men usikker på hvad meningsfuldt tekst der kan tilføres. Rodder en del med tid i begge rapporter, skifter tit mellem datid og nutid. Skrivning var aldrig udviklerens stærke siden.
Ændrede lidt på en unit test.

B.16 30/4/24

Mere renskrivning af rapporterne planlagt.
En del grammatik rettet og lidt tekst tilført eller fjernet hist og her.

B.17 1/5/24

Endnu mere renskrivning.
Startede med at ændre lidt på noget kode. UserCreate i User frontend-delen fik noget kode, der vil fremvise fejlbeskeder.
Fik renskrevet noget mere på rapporterne.

B.18 2/5/24

Planen er at færdiggøre rapporterne og aflever produktet.
Færdiggjorde rapport-skrivningen. Ændret lidt på noget kode.