Produktrapport
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

Figurer

Tabeller

Kapitel 1

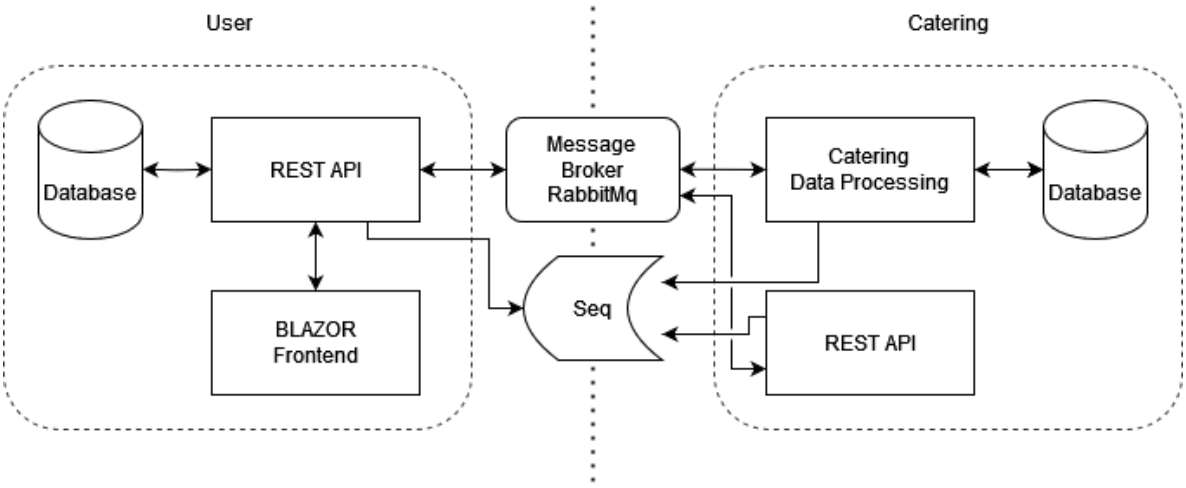
Læsevejledning

Denne rapport af en af to rapport for svendeprøven. Rapporten vil gennemgå arkitekturen, kravspecifikationerne og test sager, udvalgte teknologier og mønstre og hvordan de bruges, opsætningen af produktet, samt brugervejledning for produktet. Kapitlerne kan læses uden behov for at havde læst processrapporten først, men denne rapport vil til tider henvise læseren til bestemte dele af processrapporten. De valgte teknologier og mønstre bliver forklaret mere i dybden i processrapporten i forhold til hvad de er og redegjort for hvorfor de blev valgt. I forhold til kapitlerne, så kan de læses uden ordre, men visse områder vil give mere mening, hvis denne rapport læses fra starten.

Kapitel 2

Produkt Arkitektur

Løsningen består af to hoved-dele, User og Catering, se figur 2.1. Hver del har deres egen database og REST-API, skrevet i C♯ 12 dotnet 8.0. Forskellen mellem dem er, at User har en frontend skrevet i Blazor WebAssembly Standalone App og Catering har en CateringDataProcessing konsol program, også skrevet i C♯ 12.
Begge databaser er SQL-databaser[1], mere præcis MSSQL, og begge sider benytter en ORM[2] til at kommunikere med databaserne. Den valgte ORM er EntityFramework Core 8.x.
For at kommunikere imellem de to sider benyttes en message broker, RabbitMQ, og visse dele af begge sider sender logs til Seq via Serilog.



Figur 2.1: System oversigt

Som det kan blive observeret i figur 2.1, så foregår kommunikationen mellem databaserne og resten af systemet i forskellige steder, alt efter hvilken del der kigges på. Grunden til dette, er fordi Catering-delen oprindeligt ikke havde en REST-API.

2.1 Filstruktur

I produktet, er projekterne opdelt efter tre område, Shared, User og Catering. Hvis der kigges på projekt-løsningen i Solution-Explorer (Visual Studio 2022), så burde der være en mappe for hver del, hvor projekterne kan findes under.
Næsten alle projekter har deres eget test-projekt.
Begge Shared projekter i User og Catering har den følgende mappestruktur:

[1]Structured Query Language.
[2]Object Relational Mapping.

- Communication
- DL - Domain Layer
  - Factories
  - Models
- IPL - Infrastructure Persistance Layer
  - Context
  - Repositories
  - UnitOfWork
- Migrations

REST-API'erne er begge opsat med den følgende mappestruktur:

- Communication
- Controllers
- Middleware
- Models - Request og response modeller
- Services
- Sys - Modeller for data i appsettings.json

De andre projekter har helt forskellige mappestrukturere og bliver derfor ikke fremvist her.
Sektionerne under her vil give en kort forklaring på hvad hvert projekt, med undtagen for test-projekterne, er for.

2.1.1 Catering

Cateringdelen består af Catering.Shared, Catering.DataProcessingPlatform og Catering.API. Både Shared og Catering.DataProcessingPlatform har deres egen test-projekter.

Catering.Shared

Indeholder domæne modellerne, deres faktories, datakonteksten, Unit of Work, samt de repositories, der skal bruges for datakontekst-kommunikation.

Catering.API

Indeholder REST-API'et for Catering. Har servicere for at håndtere endpoint kald, hvor servicerne kan kalde et singleton RabbitMQ kommunikations-modul, som indeholder producers og consumers.

Catering.DataProcessingPlatform

Indeholder kode, der kommunikere med RabbitMQ via consumers og databasen via datakonteksten. Dette projekt håndtere også uhåndteret fejl i sig selv og lukning på en anden måde end resten af projekterne, da den vil logge disse ting.

2.1.2 User

User delen består af UserPlatform, UserPlatform.Shared og UserFrontend.Frontend. Både UserPlatform og UserPlatform.Shared har test-projekter.

UserPlatform.Shared

Indeholder det samme slags kode som Catering.Shared, da den tjener det samme formål.

UserPlatform

Indeholder REST-API'et for User. Udover at indeholde de samme ting som Catering.API, så har den også de nødvendige implementationer for at kunne kommunikere med datakonteksten.

UserFrontend.Frontend

Projektet der indeholder frontend-delen for User. Rent single-page application. Indeholder en HttpClient, som bruges, via servicer, til at kontakte UserPlatform.

2.1.3 Shared

Indeholde projektet Shared og dens test-projekt. Shared indeholder det kode som bruges af User- og Catering-delen. Det vil sige mønstre, kommunikationsmodeller, domain driven design klasser og kontrakter og logger.

2.2 Kommunikation mellem Programmer

Kommunikationen mellem Catering-delen og User-delen forgik via message broker'en RabbitMQ.
Dette betyder at der er intet kendskab mellem User og Catering, de ved kun hvordan de kan sende og modtage data, ikke hvem der sender eller modtager data, se figur 2.1 for kommunikationen mellem delene.

Kapitel 3

Kravspecifikationer og Test Sager

For svendeprøven blev der udviklet kravspecifikationer og test sager, se 3.1 og 3.2.
For at kunne fremvise både kravspecifikationerne og test sagerne i rapporten, var det nødvendigt at ikke visse alle kolonnerne. De fulde kravspecifikationer og test sager kan blive fundet under projektroden/Data i filerne KravSpecifikationer.xlsx og TestSager.xlsx

3.1 Kravspecifkationer

Kravspecifikationerne kan blive fundet i tabellerne A.1 og A.2. Hver krav har et id og tilhøre en kategori. Requirement forklarer hvad kravet skal opfylde for at være udført og Priority angiver hvor vigtig kravet er, laver tal er vigtigere. Krav, som har prioriteten 99, blev ikke udført som en del af produktet, men i et virkelig produkt ville diise har været implementeret.

3.2 Test Sager

Test sagerne blev udviklet ud fra nogle af kravspecifikationerne, se 3.1, og de udviklede test sager kan ses i tabel B.1. Grunden til at der ikke er en test sag for hver eneste kravspecifikation var pga. tidsbegrænsninger.

Kapitel 4

Teknologier og Mønstre

4.1 Backend

Denne sektion vil forklare implementeringerne af de teknologier der blev benyttet i backend-delen.

4.1.1 Datakonteskst

Datakonteksterne benytter MSSQL-databaser, som bliver styret af Catering.DataProcessingPlatform og UserPlatform. Begge af dem benytter Code First tilgangen, hvor modellerne, deres relationer og regler er opsat først i kode og derefter, via EntityFramework Core, oprettes/ændres databaserne, således at databaserne har de rigtige tabeller og regler. Dermed behøves en udvikler ikke at have database adgang eller at kunne SQL, hvilket formindsker muligheden for problemer. På sammen tidspunkt gør det let at oprette nye databaser, som f.eks. test-databaser, med den samme struktur som produktionsdatabaserne.
Produktet har to databaser, liggende på den samme database-server, en for User-delen og en for Catering-delen.

Seeding

Catering-datakonteksten seeder sin database med noget test data, hvis databasen er tom og hvis Configuration Manager'en er sat til andet end Release.

4.1.2 ORM

EntityFramework Core 8 er den nyeste udgave af Microsofts ORM. EntityFramework Core benytter sig af reflektion til at automatisk lave de nødvendige mapping fra objekt-model til relationel tabeller og kan selv finde ukendte objekter den skal mappe over. Den kommer med både Unit Of Work og Repository Pattern indbygget. Hvis det er nødvendigt at overskrive dele af mapperen er dette muligt via Fluent API i datakontekst filen.
I dette projekt har det været nødvendig at overskrive dele af den automastiske mapping. Gruden til dette er fordi Value Object fra Domain Driven Design benyttes og et Value Object skulle aldrig have en id, da den styres og er kun kendt af det objekt der ejer den. I tidligere udgaver af EntityFramework Core var det ikke muligt at skabe rigtige Value Objects i datakonteksten, da alle objekter skal have en nølge i datakonteksten. Dette er, til dels, løst i version 8 via en ny konfigurationsmulighed ComplexProperty som tillader ikke-samlinger af Value Object til ikke at have en nøgle. For samlinger af Value Objects er det nødvendigt at benytte den gamle måde, hvilket skaber nøgler i datakonteksten. Se 4.1 for opsætningen via den nye måde og den gamle måde via Fluent API i datakontekst filen.

Listing 4.1: Value Object Opsætning

```
1 modelBuilder.Entity<Customer>(e =>
2 {
3     e.ComplexProperty(e => e.Location);
4     e.OwnsMany(e => e.Orders);
```

```
5 });
```

Migrations

Migrations i EntityFramework Core er den måde den holder styr på hvilken ændringer, der skal gøres på en database. Der er to typer af filer, selve migration filerne og en ModelSnapshot. En migration fil indeholde de ændringer, der skal udføres i forhold til migrationen før, både for at lave ændringerne og for at fjerne ændringerne. ModelSnapshot filen har den nuværende opsætning af databasen, f.eks. nøgler, fremmede-nøgler, tabeller og mere og benyttes til at finde ændringerne der skal udføres, når en ny migration laves.
For at simplificere oprettelsen og overførelsen af migrations, blev et Powershell script udviklet, som ligger i løsningsroden med navnet contexthandler.ps1. Grunden er fordi der er to kontekster, User og Catering, samt at begge har deres datakontekst i et andet projekt end hvor logikken for at oprette en instans af datakonteksten og udførelsen af database opdatering ligger.

4.1.3 ASP.Net Core

I C♯ benyttes ASP.Net Core som framework for at skrive REST-API'er og visse former for multi-page application frontends, dem som kontakter serveren. Frameworket er udviklet af Microsoft og kommer med en basisk, men brugbart, opsætning som standard og kan derefter opstilles som der er behov for. Det er muligt at indsætte middlewares, både for request og response, ind i dens pipeline. CORS kan let sættes op, sikkerhed kan opstilles efter produktet behov og meget mere. ASP.Net Core håndtere også automatisk mapping af HTTP body og query til parameterne på en endpoint.
CORS er sat op for User REST-API'et for at tillade frontend-delen at kunne kommunikere med API'et, samt sikkerhed sat op med JWT. Begge API'er har også en middleware, der logger hvad for nogle endpoints der bliver kaldt.
ASP.Net Core kommer med Swagger, hvilket tillader letter at udføre manuelle test af endpoints.

Swagger

Alle REST-API'er i dette projekt benytter Swagger. Swagger giver et detaljeret overblik over API'et og tillader fremvisning af dokumentation. I dette produkt, er Swagger opsat til at fremvise alle endpoints, deres HTTPMethod og hvad for noget data de modtager. User REST-API'et fremviser hvilken endpoints kræver at brugeren er logget ind og eller ej og en enkel punkt for at indsætte login token og dermed slipper brugeren for at manuel indsætte den ved hver request.

4.1.4 Domain Driven Design

For en kort gennemgang hvad Domain Driven Design er, henvises der til sektionen Domain Driven Design i Processrapporten.
For at styre hvad for nogle objekter der er aggregate rødder blev en kontrakt, kaldt IAggregateRoot, oprettet, som aggregatrod modellerne så implementere. IAggregateRoot indeholder kun en enkel get property (GUID) for at hente id'et.
Når det kommer til 'renligheden' af modellerne i domain driven design og f.eks. validering af data, kan der opstå nogle problem. F.eks. i dette projekt kan kun en enkel User have et bestemt CompanyName, men problemet ligger i hvordan dette skal valideres. I sit reneste form, skal alt validere foregå i modellen og dermed skal en User have kendskab til alle brugte CompanyName og dette skal den enten gives eller kunne hente selv. Hvis den gives data'en kan den ikke validere om data'en er korrekt og hvis den skal hente listen, skal den have kendskab til datakonteksten, som intet har med modellen at gøre og modellen burde aldrig være koblet samme med kode som den ikke ejer. Løsningen i dette produkt var at have en factory for hver aggregatrod, som står for validering af data og oprettelse af nødvendige objekter. De forskellige factories modtager både skabelse request'en og valideringsdata'en.
I princippet skulle objekterne også valideres, når de hentes fra datakonteksten, men det blev valgt ikke at gøre dette for at lette udvikleren, men det skulles nok gøres i et virkelig produkt i det tilfælde nogen ændre på data'en i databasen.

For at give 'reference' til andre objekter ude for aggregaten, så bruges der en record objekt `ReferenceId` til at standardisere, hvordan det blev udført.

Value Objekt

Value objekter er implementeret på måden fremvis i UserLocation i 4.2. Som det kan ses, så kan værdierne i UserLocation ikke overskrives, det er nødvendigt at overskrive hele objektet. ValueObject er en record, hvilket betyder at dens sammeligningskode er overskrevet til at sammelige på objekternes værdier.

Listing 4.2: Value objekt overskrivning

```
public sealed class User : IAggregateRoot
{
    ...
    public bool UpdateCity(string city)
    {
        ...
        _location = new UserLocation(city, _location.Street);
        ...
    }
    ...
}
public sealed record UserLocation : ValueObject
{
    ...
    public string Street { get => ...; private set => ...; }
    public string City { get => ...; private set => ...; }
    ...
    internal UserLocation(string city, string street)
    {
        _city = city;
        _street = street;
    }
}
```

4.1.5 Command Query Responsibility Segregation

Command

Command-delen blev implementeret med et interface `ICommand`, som er en tom kontrakt. Den benyttes af de commands, der sendes via RabbitMQ for at påvirke consumer'en. Det er ikke strengt nødvendigt, men indført i det tilfælde, at der på et tidspunkt skulle sendes noget bestemt data med alle commands.

Query

Query-delen blev implementeret med to klasser, `BaseReadModel` og `BaseQuery<TEntity,TMapping>`. `BaseReadModel` er en tom abstrakt klasse, som bruges af `BaseQuery<TEntity,TMapping>` som begrænsning. `BaseQuery<TEntity,TMapping>` er en abstract klasse, der indeholder en abstract metode der kan bruges i `System.Linq.Queryable.Select`, se 4.3 for et simple implementation.

Listing 4.3: Konkret implementation af BaseReadModel og BaseQuery

```
public sealed class CustomerData(Guid id) : BaseReadModel
{
    public Guid Id { get; private set; } = id;
}

public sealed class CustomerDataQuery : BaseQuery<Customer, CustomerData>
{
    public override Expression<Func<Customer, CustomerData>> Map()
    {
        return e => new(e.Id);
    }
}
```

```
13
14 public class EntityFrameworkCoreRepository<TEntity, TContext> ...
15     public async Task<IEnumerable<TMapping>> AllAsync<TMapping>(BaseQuery<TEntity,
           TMapping> query) where TMapping : BaseReadModel
16     {
17         return (await _entities.ToArrayAsync()).AsQueryable().Select(query.Map());
18     }
19     ...
20 }
```

Som det kan observeres i 4.3 linje 16, så bliver objekterne først mappet efter de er blevet hentet. Det er dog muligt at 'mappe' dem direkte over i SQL-databasen, da EntityFramework Core 8.x kan oversætte Expression<Func<TEntity,TMapping> til SQL-query, som henter de nødvendige kolonner. Grunden til dette ikke blev gjort er nævnt i 4.1.7, men det er et problem med hvordan predicates er implementeret. Selv om der er metoder, der ikke tager predicates ind, blev det valgt at mappe disse over i softwaren frem for SQL-serveren for at holde koden ensformet.

4.1.6 Result Pattern

Result Pattern blev implementeret med seks type, disse er:

- Success
- SuccessNoData
- BadRequest
- NotFound
- Unhandled
- InvalidAuthetication

Der er dog ingen grænse på hvilken typer, der kan være implementeret.
Forskellen mellem Success og SuccessNoData er, at lettere kunne omdanne en `Result` til HTTPStatus 204.
For at bære fejl, så blev en binære fejl klasse, `BinaryFlag`, implementeret, som kun kan modtage `Enum` for at sætte sit flag. Grunden for dette valg, er fordi det er let at opsætte binære flag i `Enum`. Der er dog en konstruktør, der modtager en long, da det er nødvendigt at kunne sætte evt. fejlbeskeder, når fejl er sendt igennem RabbitMQ, da `BinaryFlag`, ikke bliver sendt. Klasssen har også to implicit operator overloads, en for boolean og en for long. Den overrider `ToString()`, så den kan give sit flag som en binær streng tilbage.
En talværdi blev valgt frem for strenge, da, hvis kalderen skal håndtere fejl, dette design gør det letter at håndtere fejl. Hvis værdien sendes til frontend, så kan der implementeres en konverter, der omdanner hver bit til en streng.
For at omdanne et `Result` til en `IAction` blev den følgende extension metode implementeret.

Listing 4.4: Result extension metode

```
1  public static class ResultResponseMapping
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.SuccessNoData => controller.NoContent(),
9              ResultType.BadRequest => controller.BadRequest((long)result.Errors),
10             ResultType.InvalidAuthetication => controller.Unauthorized(),
11             ResultType.NotFound => controller.NoContent(),
12             ResultType.Unhandled => controller.Problem(statusCode: 500, detail:
                   result.Errors.ToString()),
13             _ => throw new Exception("Internal server problem"),
14         };
```

```
    }
}
```

4.1.7 Repository Pattern

Repository Pattern benyttes til at overføre data mellem datakonkesterne og softwaren. Datakontekssterne i dette produkt er SQL-databaser. Den valgte ORM, EntityFramework Core, kommer med dens egen rappository pattern, men det blev valgt at lægge et lag over det, for at fjerne koblingen mellem EntityFramework Core og de klasser, der benytter repositories.
Lag-opsætningen er det følgende:

- IBaseRepository<TEntity> - Fuld generisk
- EntityFrameworkCoreRepository<TEntity, TContext> - EntityFramework Core, men generisk på model og EF datakontekst
- I{Model}Repository - Ikke generisk, for bestemt model
- {Model}Repository - Ikke generisk, for bestemt model, implementeret for at benytter IBaseRepository<TEntity>

Repository Pattern blev opsat sådan at de konkrete implementationer, af de forskellige kontrakter, ikke selv kontakter datakonteksten. I stedet for modtager de en konkrakt implementation af interfacen IBaseRepository<TEntity>, som står for hentningen af data, hvor de konkrete repositories vil benytte metoder på denne til at kontakte datakonteksten. Så i stedet for at have f.eks. en CustomerRepository og en TestCustomerRepository, så er der kun CustomerRepository, som så modtager et objekt som f.eks. peger på test-data eller en database-kontekst, se 4.5 for hvordan det er opsat for EntityFramework Core. Dette øger selvfølgelig kompleksiteten af koden og vedligeholdelsen, men fordelene anses for at være det værd.

Listing 4.5: Repository Pattern

```
public sealed class CustomerRepository : ICustomerRepository
{
    private readonly IBaseRepository<Customer> _repository;

    public CustomerRepository(IBaseRepository<Customer> repository)
    {
        _repository = repository;
    }

    public async Task<IEnumerable<TMapping>> AllAsync<TMapping>(BaseQuery<Customer,
         TMapping> query) where TMapping : BaseReadModel
    {
        return await _repository.AllAsync(query);
    }
    ...
}

public interface IBaseRepository<TEntity> where TEntity : class, IAggregateRoot
{
    public void Create(TEntity entity);
    ...
    public Task<TEntity> FindByPredicateAsync(Func<TEntity, bool> predicate);
    public Task<IEnumerable<TMapping>> AllAsync<TMapping>(BaseQuery<TEntity,
        TMapping> query) where TMapping : BaseReadModel;
    ...
}

public class EntityFrameworkCoreRepository<TEntity, TContext> : IBaseRepository<
    TEntity> where TEntity : class, IAggregateRoot where TContext : DbContext
{
    private readonly TContext _context;
    private readonly DbSet<TEntity> _entities;

    public EntityFrameworkCoreRepository(TContext context)
```

```
32      {
33          _context = context;
34          _entities = _context.Set<TEntity>();
35      }
36      ...
37  }
38
39  public class UnitOfWorkEFCore : IUnitOfWork
40  {
41      ...
42      public UnitOfWorkEFCore(CateringContext context)
43      {
44          ...
45          _customerRepository = new CustomerRepository(new
                EntityFrameworkCoreRepository<Customer, CateringContext>(context));
46      }
47      ...
48  }
```

Visse ikke-generiske repository metoder har argumenter som GUID og andre har Func<TEntity,bool>, se 4.6. Grunden til dette valg, var for at fremvise forskellige niveauer af kontrol udviklerne kunne gives. Metoderne med predicate giver mere kontrol til udviklerne, men kan skabe problemer, da en udvikler kan skrive `x => true`. Metoderne med ikke predicate-parameter er letter at bruge, det er lettere at lave det samme kald flere steder og har mindre chance for fejl, men kan kræve at mange metoder bliver oprettet.

Listing 4.6: Forskellige argumenenter

```
1       public async Task<Order> GetSingleAsync(Func<Order, bool> predicate)
2       {
3           return await _repository.FindByPredicateAsync(predicate);
4       }
5
6       public async Task<RefreshToken> GetTokenAsync(string token)
7       {
8           return await _repository.FindByPredicateAsync(x => string.Equals(x.Token,
                token));
9       }
10
11      public async Task<RefreshToken> GetTokenAsync(Guid userId)
12      {
13          return await _repository.FindByPredicateAsync(x => x.User.Id == userId && x
                .Revoked == false);
14      }
```

Et problem med at benytter Func<TEntity,bool> predicate er EntityFramework Core 8.x ikke kan oversætte dem til SQL-queries, da predicate kan f.eks. indeholde metoder-kald. Dette betyder, at det er nødvendigt at hente alt data'en først og derefter filter på det, se 4.7 for hvordan det håndteres i koden.

Listing 4.7: Predicate håndtering

```
1       public async Task<TEntity> FindByPredicateAsync(Func<TEntity, bool> predicate)
2       {
3           return (await _entities.ToArrayAsync()).FirstOrDefault(predicate)!;
4       }
```

4.1.8 Unit Of Work

Meningen med Unit Of Work er, at udføre alle database ændringer på engang. Som i 4.1.7, så kommer EntityFramework Core med dens egen Unit Of Work, men det blev valgt at indsætte egen Unit Of Work lag mellem EntityFramework Core og resten af softwaren.
Der er et Unit Of Work for både User-delen og Catering-delen. De indeholder properties for alle repositories i deres del. Det er muligt at finjustere hvad en udvikler har adgang til via Unit Of Work ved at kun indsætte visse repositories eller have forskellige repository kontrakter, som IDishReadRepository

og IDishWriteRepository. For dette produkt er der så få modeller, at det ikke gav mening at indføre mere finkontrol.

4.1.9 Kommunikation

For at kommunikere mellem de forskellige dele af løsningen, User og Catering, blev RabbitMQ valgt som message broker. RabbitMQ virker ved at sende data fra en producer, via en queue, til en consumer som håndtere data'en. På sammen tid er det også muligt at sætte op Remote Procedure Call (RPC), hvor consumer'en sender et svar tilbage til producer'en.
En producer er vist i 4.8 og dens consumer er vist i 4.9.

Listing 4.8: Producer

```
1  public Result TransmitUser(User user)
2  {
3      _logger.Information("{Identifier}: Transmitting user", _identifier);
4      UserCreationCommand command = user.ToCommand();
5      var body = command.ToBody();
6      try
7      {
8          _channel.BasicPublish(exchange: string.Empty, routingKey:
               CommunicationQueueNames.CUSTOMER_CREATION, basicProperties: null!, body
               : body);
9          return new SuccessNoDataResult();
10     }
11     ...
12 }
```

Listing 4.9: Consumer

```
1      private void ReceivedForCustomerCreation(object? sender, BasicDeliverEventArgs
           e)
2      {
3          var message = e.ToMessage();
4          try
5          {
6              var request = message.ToCommand<UserCreationCommand>();
7              if (request is null)
8              {
9                  ...
10                 _channel.BasicAck(e.DeliveryTag, false);
11                 return;
12             }
13             ...
14         }
15         ...
16         _channel.BasicAck(e.DeliveryTag, false);
17     }
```

For at hjælpe med at modtage svar fra RPC'er, blev en lille klasse Carrier oprettet. Som for Result Pattern, 4.1.6, så bære denne model en Enum om kaldet lykkedes eller ej, en long for fejlbeskeder, og en nullable streng, som kan indeholde en JSON streng, hvis data skulle sendes tilbage.

4.2 Klient

Klient-app'en er skrevet i Blazor WebAssembly Standalone App, hvilket betyder, at hele klienten køre over i brugerens browser via WebAssembly[1]. Dette betyder at klient-app'en er en single page application (SPA), dermed spare serveren ressourcer, dog skal brugeren benytte flere ressourcer for at køre klienten. At køre koden hos brugeren medbringer et sikkerhedrisiko, da brugeren har adgang til alt koden, hvilken betyder at kald til datakontekster, som en sql-database, ikke kan foretages via klienten og dermed skal der være en dedikeret API, som klienten kan kontakte for at modtage nødvendig data.

[1] WebAssembly er en teknologi, hvor binære instruktioner køres på en stack-based virtuel maksine i brosweren.

4.2.1 Blazor

Blazor kommer i to udgaver, Blazor WebAssembly Standalone App og Blazor Web App. Begge benytter sig af Microsofts Razor syntaks, men forskellen er, at Blazor WebAssembly Standalone App er en ren SPA, hvor alt koden køres i browseren, hvorimod Web App kontakter serveren, når den skifter siden, og er dermed en multi page application.
Blazor benytter komponenter, som indeholder både HTML og kode, til brugerfladen. Et eksemple på en Blazor komponent er givet i 4.10. En komponent består normalt af tre dele, den første del er @page og @using, den næste del er alt HTML koden og den sidste del er alt inde i @code. Razor syntaksen kan genkendes på kode, der starter med @ i den første og anden del, da dette symbol lader kompileren vide, at det ikke er HTML-kode.
I eksemplet kan der ses, på line 14, at den fremviste komponent kalder en anden komponent og overføre data til den via `Order="order"`. Dette kaldes Parent-Child struktur, hvor komponentet, der kalder andre kompontenter, er Parent og kaldte kompnenter er Children.
En forskel mellem disse to komponenter er, at OrderDetails-komponentet ikke har en @page og dermed ikke kan findes via url'en[2]. @page bruges til at definere stien til komponenten i url'en.

Listing 4.10: Blazor Komponent

```
1  @page "/order/placed"
2  @using UserFrontend.Frontend.Models.Order.Responses
3  @using UserFrontend.Frontend.Pages.Order.Orders.Details
4  @using UserFrontend.Frontend.Services.Contracts
5  <h3>Orders Placed</h3>
6
7  @if(orderCollection is not null)
8  {
9      <div class="container">
10         <div class="row">
11             @foreach (var order in orderCollection.Orders)
12             {
13                 <div class="col-3">
14                     <OrderDetails Order="order"></OrderDetails>
15                 </div>
16             }
17         </div>
18     </div>
19 }
20
21 @code {
22     private OrderCollectionResponse orderCollection;
23
24     [Inject]
25     public IOrderService OrderService { get; set; }
26
27     protected override async Task OnInitializedAsync()
28     {
29         await base.OnInitializedAsync();
30         var result = await OrderService.GetOrders();
31         if (result)
32             orderCollection = result.Data;
33         else
34             orderCollection = null!;
35     }
36 }
```

4.3 Kobling

For at sikre, at de forskellige moduler havde så løs-som-muligt kobling, blev Dependency Injection benyttet via konstruktører. På sammen tid blev interfaces brugt som kontrakter, som så blev implementeret af konkrete klasse. Dette tilladte letter udskiftninger af konkrete klasser, uden at skulle skifte parametrene i de klasser, der benyttede de erstattede moduler, hvilket letgjorde testning af modulerne. Siden produktet bestod af tre dele, User, Catering og Shared, så blev der opsat nogle mentale-regler

[2] En kaldt komponent kan dog havde en @page.

for hvordan disse kunne refererer til hinanden. User og Catering kunne aldrig refererer til hinanden, men begge kunne refererer til Shared og Shared havde ingen referencer til User og Catering.

4.4 Datalogning

Seq blev opsat til at benytte dens standard indstillinger i forhold hvor lang tid logs bliver gemt og ligende. Der blev oprettet seks api-nøgler. Hver projekt, der skulle logge, fik to nøgler, en for når de skulle køre i testmiljøet og en for når de skulle køre i livemiljøet.
Det blev valgt at logge kald til alle endpoints i begge REST-API'er via middleware. Alle RabbitMQ producers og consumer loggede fejl-beskeder. Alle producers loggede, når de blev kaldt. For consumers over i Catering.DataProcessing, så blev der oprettede debug logs, når en command begyndte at blive behandlet og når den var færdig med at blive behandlet.
For at skrive til Seq blev Serilog benyttet, da Serilog er den logger Seq fremviste i deres C♯ dokumentation. Nogle eksempler på forskellige logs kan ses i 4.11. Serilogs ILogger kommer med forskellige niveauer af logs, her fremvises Debug, Error og Fatel, men der findes også Warning og Information. `{ParameterName}` betyder at seriloggeren vil indsætte den næste argument på dette punkt og kalde ToString() på det. Hvis en argument mangler vil det dukke op i Seq som {ParameterName}. Hvis det sat op som `{@ParameterName}` vil arguementet blive serialiseret frem for at kalde ToString() på det. Parameternavnet inde i `{ParameterName}` vil bruges som parameternavn over i Seq. Hvis en exception er givet, så vil Seriloggeren sende exception-data med over og dermed gøres det letter at debugge bagefter. Serilog ser helst at en exception gives som det første argument.

Listing 4.11: Serilog

```
1         _logger.Debug("{Identifier}: Processed request {@OrderFuture}", _identifier
              , command);
2
3         _logger.Debug("{Identifier}: Processed request {OrderFuture}", _identifier,
               command);
4
5         _logger.Error(ex, "{Identifier}: Error processing {Message}", _identifier,
              message);
6
7         logger.Fatal(ex, "Unhandled exception encounted");
```

For dette produkt blev der altid skrevet til både Seq og konsollen på sammen tidspunkt. I virkeligheden ville det være bedst ikke at skrive til Seq, hvis en debugger er sat til. I de tilfælder, hvor der sendes mange logs på kort tid, kunne det har været en god ide ikke at skrive til konsollen, da den er langsommere end at sende til Seq.

4.5 Test

Næsten hvert eneste ikke-test projekt fik et test-projekt. Catering.API fik ikke en, da dette projekt blev oprettet sent i udviklingen, og UserFrontend.Frontend fik heller ikke en. Test-projekterne er alle XUnit projekter og tilsamme fremvise de nogle få tests. I virkeligheden ville alle stier igennem kode, både for unit tests og integration test, få automatiske test med både gyldig og ugyldig værdier.
Navngivningen for tests i dette produkt er metodenavnet, scenarioet og forventet opførelse i form af spørgsmål. Nogle eksempler fra produktet er disse to test metoder:
`Does_Valid_Create_User_Return_200_With_Auth_Data()`
`Does_Valid_Build_Success_Result_And_User()`
Begge navne forklare hvad der bliver kaldt og hvad der forventes som resultat.
Alle test er opbygget efter mønstret Arrange, Act, Assert (AAA). Det vil sige først opsættes de nødvendige objekter for testen, derefter kaldes det kode der testet og til sidst tjekkes der om testen gav det korrekte resultat eller ej. Der er nogle få gange, hvor dette mønstre ikke helt kan overholdes, f.eks. når der testes for exception throws, da XUnit forventer kaldet, der skaber exception'en i dens `Assert.Throws<TException>(...)`. Nogle andre steder, hvor AAA ikke bliver helt overholdt, er i Theory test metoder[3], hvor data'en der normalt går i Arrange allerede er sat op fra metodens parameter.

[3]Theory udføre den samme test flere gange med forskellige data i forhold til Fact som kun køre testen engang.

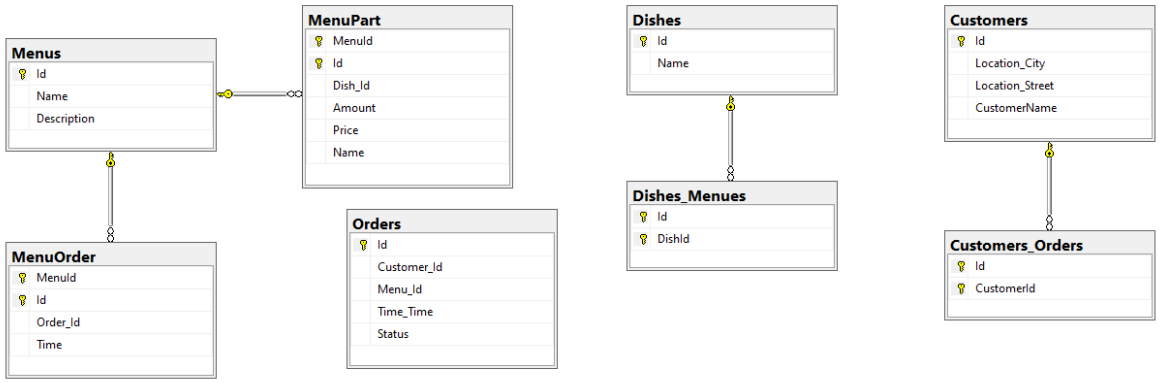
En vigtig ting at teste, som ikke bliver fremvist i produktet, er test af rigtige datakontekst-forbindelser, hvor test-projekterne ville havde test-databaser, der kunne bruges til at teste om den valgte ORM og repositories virkede som de skulle.
To ord der dukker op tit, hvis der kigges på testene, er 'dummy' og 'Mock'. Variabler der starter med ordet 'dummy', er variabler, der kun findes fordi de er nødvendige, men de bruges ikke i Assert delen og tit heller ikke af det kode under test [1]. Variablerne der slutter med ordet 'Mock', er test-udgaver af objekter, som er blevet opsat til at forvente visse kald og håndter kald på en bestemt måde [1]. De tjekkes normalt i Assert for at se om de blev kaldt som forventet. For dette produkt blev Moq valgt som mock-framework. Grunden til dette, er fordi udvikleren ved, fra erfaring hvor besværligt og tidskrævende det er, at skrive egen mock og Moq letter dette arbejde en del. Et andet ord der dukker tit op, er 'sut', hvilket står for 'System Under Test'.

4.6 Database

Databasernem der blev benytter for dette produkt, er SQL-databaser i form af MSSQL 2022. MSSQL er udviklet af Microsoft.
Begge databaser bliver oprettet og styret af EntityFramework Core.

4.6.1 Relationer

Relationerne mellem de forskellige objekter kan ses i figur 4.1 og figur 4.2. Der skal gøres opmærksom på at alle relationer er kun mellem objekter, der tilhøre bestemte aggregater, se processrapporten omkring Domain Driven Design. De tabeller der hedder {AggregaterodNavn}_{AndetNavn} er sat op i koden til at være reference nøgle objekter og er samlinger, det vil sige `HashSet<ReferenceId>` i et objekt model. EntityFramework Core 8.x kan ikke oprette disse, uden at skulle give dem deres egen tabel og dermed får de et id i databasen, selv om de er Value Object og dermed ikke burde have en. De kolonner der hedder {Navn}_{AndetNavn} er også Value Objects, men siden der kun er en enkel af dem i objekt modellen, kan EntityFramework Core oprette dem korrekt, i forhold til at være Value Object, i databasen. Som det kan ses i figur 4.2, så er modellen UserLocation, som i koden er sin egen model og kendt af User, blevet oprettet som kolonner i User tabellen i form af Location_City og Location_Street. Dermed er ikke alle objekt modeller mappet over til deres egene tabeller.

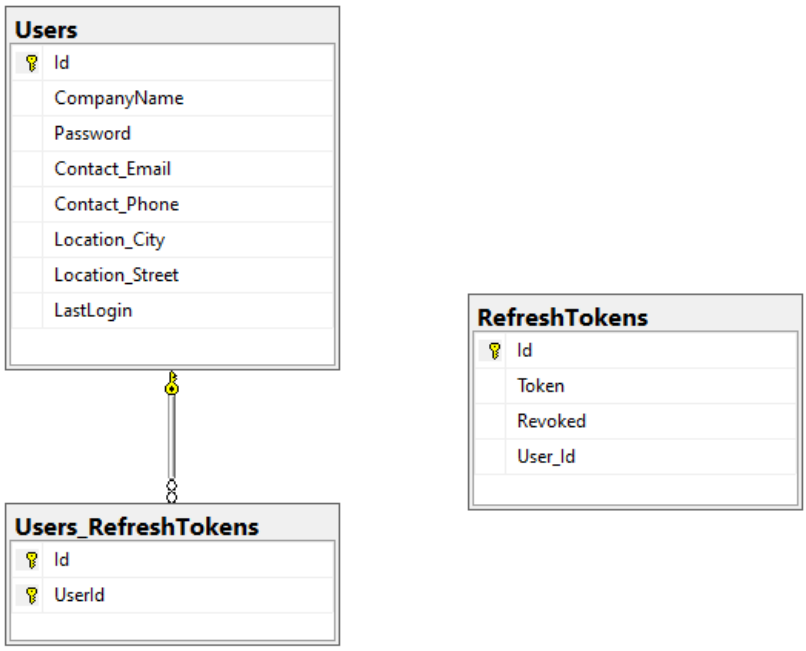


Figur 4.1: Catering ER Diagram

4.7 Konfigurationsstyring

De branches og git-flow, der blev brugt for dette produkt, kan ses i figur 4.3. Dette ville være for simple for et rigtig produkt[4]. Den næste sektion vil fremvise et bedre flow.

[4]Commits direkte på Developer eller Master kan øger changes for merge konflikter.



Figur 4.2: User ER Diagram

4.7.1 Forbedre Konfigurationsstyring

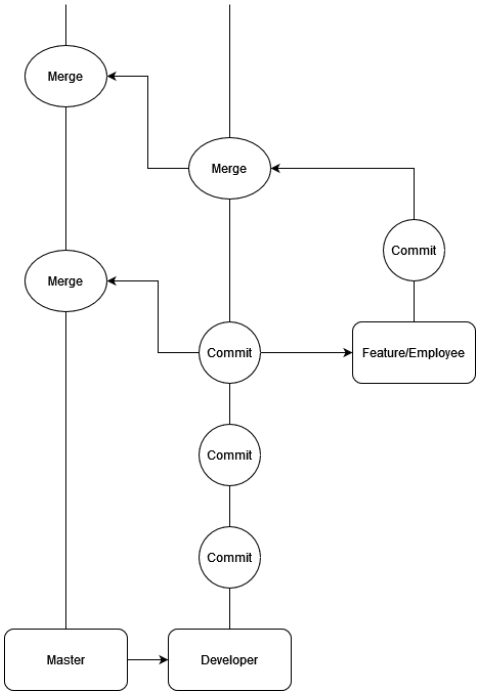
Et forbedre konfigurationsstyring, via Git, ville havde tre vigtige branches, Developer, Testing og Production. Developer ville være den branch alle udviklings-branches ville blive oprettet ud fra og merge ind i. Production er det kode, der køre ude i produktionsmiljøet. Testing ville være ledet mellem Developer og Production, hver gang Developer blev opdateret, skulle den skubbes over til Testning, som ville køre en auto-udgivelse af koden til testmiljøet, hvor testmiljøet ville rapporter om der var kørefejl (startede programmet op, crashede det og sådan), og tester kunne teste om brugerfladen og det virker. Når den havde kørt i noget tid, kunne Developer manuelt merges ind i Production og udgives. Denne opsætning kan ses i figur 4.4, der skal dog peges ud at figuren fremviser at merge Testing ind i Production, grunden til dette er fremvise er, at koden ville teste først i Testing og ikke overføres til Testing og Production på en gang.
De følgende præfiks ville bruges til branchnavne:

- Feature - En ønsket funktion
- Hotfix - En fejlfiks der skal ud hurtigt.
- Bugfix - En fejl der skal fikses.

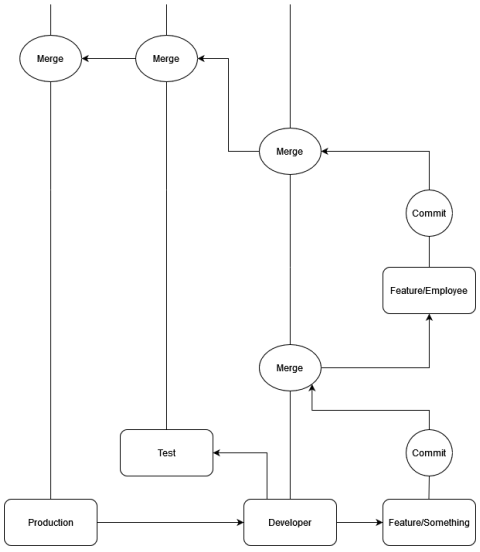
På samme tid skulle alle merges udføres, via pull-request med Peer-Review, hvor en eller flere andre udviklere går det kode, der skal merges ind, igennem og give feedback som nødvendigt.

4.8 Klassediagram

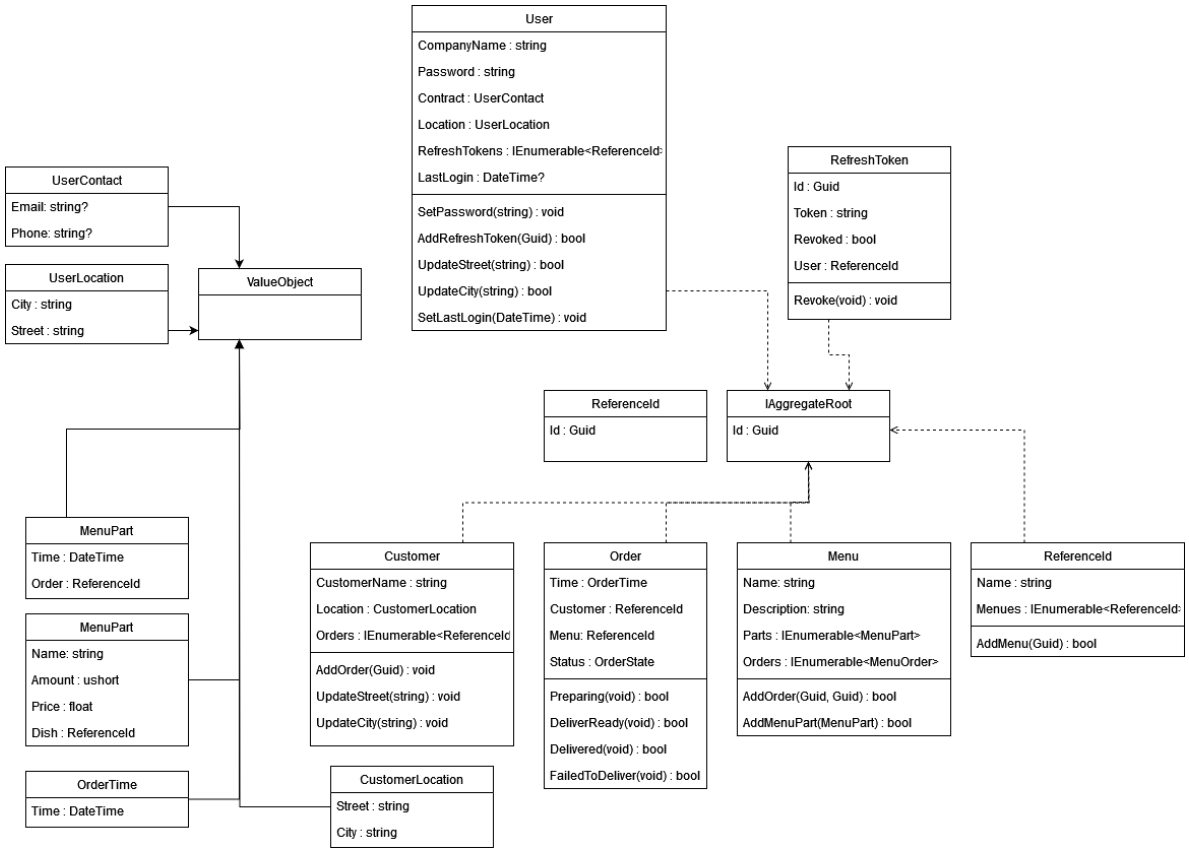
Klassediagramet for entity modeller, i bounded context User og Catering, kan ses i figur 4.5. Der bliver ikke fremvist private eller internal properties, da for det meste ikke er behov for dem for udviklerne, som ikke arbejder på modellerne. Alle properties har kun public getters, hvorimod deres setters altid er private.
Grunden til at ikke alle klasser blev medbragt, var fordi de fleste klasser kun har det formål at sende data rundt i systemet.



Figur 4.3: Nuværende versionsstyring



Figur 4.4: Versionsstyring for et rigtig produkt



Figur 4.5: Klasse Diagram

Kapitel 5

Sikkerhed

Denne sektion vil gennemgå sikkerheden og hvilken tænker der blev gjort.

5.1 Catering.DataProcessing

Dette konsol program har det sikkerhedsproblem, at dens datakontekst-forbindelse-strengen er liggende i selve koden, samt i appsettings.json filen. Grunden til dette, var problemer med at koden, der kaldes af dotnet EF, til få den til at modtage forbindelse-strengen fra kommandoprompten.

5.2 REST-API

REST-API'erne benytter HTTPS, da dette giver end-to-end kryptering mellem serveren og klient.
API'et for User er sat op til at benytte JWT (JSON Web Token) til at give adgang til alle endpoints, som ikke er Login eller User oprettelse.
JWT er en åben standard (RFC 7519) og er en enhed for at sende information mellem forskellige enheder på en sikker måde, da det er digitalt underskrevet med enten en hemmelig nøgle (symmetrisk nøgle) eller en offentlig/privat nøgle (asymmetrisk nøgle) [3].
En JWT består af tre dele, Header, Payload og signatur. Header'en angiver hvad for en type token det er og hvad underskrevning-algoritmen er. Payload inderholder påstande, f.eks. hvornår token udløber (exp), hvem det er omkring (sub) og mere. Signaturen indeholder den indkodet header, indkodet payload og en hemmelighed, som bliver underskrevet af den valgte algoritme [3]. Dermed kan en JWT ikke blive ændret af en trejde-part uden at de har nølgen. I dette produkt, er der benyttet en symmetriske nøgle og med HMAC SHA256 som underskrivning-algoritme. HMAC SHA256 er en almindelig algoritme for JWT.
JWT bliver sat i header'en som Authorization: Bearer <token> og skal bruges på alle endpoints med undtagen for login og brugeroprettelse.
Når en bruger logger ind eller bliver opretter, får de to JWT tilbage, en normal login JWT og en refresh JWT. Forskellen er, at refresh JWT'en ikke indeholder det nødvendige information for at kunne logge ind, men kan bruges til at oprette en ny login JWT uden behov for at logge ind igen. Refresh JWT bliver gemt server-side og kan blive ophævet, hvis det er nødvendigt. Refresh JWT'en har en længere levetid end login JWT'en har, i dette projekt 30 minutter i forhold til 15 minutter. Den korte levetid for login JWT'en sikre sig, at en kompromitteret login JWT kun kan bruges for kort tid. Koden er også sat op til at når en ny refresh JWT oprettes, for brugeren, at den tidligere bliver tilbagekaldt, så hvis en bad-actor får fat i refresh JWT og bruger den til at logge ind vil, så vil den rigtige bruger, hvis brugeren er aktiv, finde ud af det inde for 15 minutter, da de er nødsaget til at logge ind igen. Det har dog det problem, at hvis en bruger logger ind på to enheder, at den første enhed vil miste sin refresh JWT.
REST-API'et over i Catering-delen har intet sikkerhed sat op på sig. I et mere udviklet produkt, ville den havde en lignende JWT som User-delen, men JWT'en ville også indeholde en liste over brugerroller, da ikke alle bruger burde have adgang til alle endpoints. F.eks. køkkenpersonale ville havde adgang til at se fremtidige ordre, men ikke menu- eller ret-oprettelse.

Corss-Origin Resource Sharing

Cross-Origin Resource Sharing (CORS) kan benyttes til at styre hvilken apps og brower kan kommunikere med en server og med hvilken HttpMetoder og headers der er tilladte. CORS er sat i User REST-API'et til at tillader alle forbindelser med alle metoder og alle headers. Dette var for at simplificer testning og udviklingen af løsningen. I et færdig produkt ville CORS sættes op til at være mere streng i forhold til headers. HttpMetoder og origins kan der dog ikke gøres så meget ved, siden User REST-API'et peger ud til det offentlige.

Kodeord Håndtering

Brugerens kodeord bliver hashed og salted, før det gemmen i datakonteksten. Frem for at implementerer egen udgave, så benyttes Microsofts PasswordHasher<TUser>. Denne hasher og salter kodeordet med PBKDF2 HMAC-SHA512, har 128-bit salt og udføre 100.000 iterationer [2].

Login Forsinkelse

I koden for login-delen, er der sat en ventetid på minimum 500 ms. Grunden til dette er, at forsinke forsøg fra en bad-actor på at bruteforce sig ind, samt at finde ud hvilken brugernavne der er i brug, via at time hvor lang tid loginforsøg tager. Som koden er sat op vil den returner hurtigere, hvis Thread.Sleep ikke var der, hvis den ikke kan finde en bruger og dermed er det muligt, via svartiden, at kunne gætte om brugernavnet er i brug eller ej. Den indsatte tid, der skal gå, sikre sig at dette ikke er muligt, se 5.1. Sådan en kort tid burde ikke blive irriterende for en normal bruger.

Listing 5.1: Forsinkelse

```
1  public async Task<Result<UserAuthResponse>> UserLoginAsync(UserLoginRequest request
       )
2  {
3      TimeSpan sleepLength = TimeSpan.FromSeconds(0.5);
4      Stopwatch sw = new();
5      sw.Start();
6      var result = await _securityService.AuthenticateAsync(request);
7      sw.Stop();
8      var timePassed = sw.Elapsed;
9      var missingTime = sleepLength - timePassed;
10     if(missingTime.TotalMilliseconds > 0)
11         Thread.Sleep(missingTime);
12     return result;
13 }
```

Bruger Information

Når User REST-API'et skal benytte brugerens id, bliver id'et hentet fra den JWT der er sat i Authorization header. Grunden til dette valg er, at undgå at en bad-actor kan angive en anden bruger end sin egen, hvis id'et blev sat i response body'en eller i query'en. På sammen tid er id'et også krypteret.

5.3 Blazor

I forhold til lagering af data, som JWT, blev der benytter en singleton instans af en klasse, der gemte både login-token og refresh-token. I Blazor WebAssembly Standalone App er dette valg sikker nok, da en singleton instans kun eksistere for en bestemt tab i en broswer og dermed ikke delt med andre [4]. Applikationen benytter både route-level og component-level autorisation for at styre hvad en bruger kan se. 5.2 fremviser route-level og 5.3 fremviser component-level. I _Imports.razor er der indsat @attribute [Authorize], hvilket betyder at alle komponenter, som ikke har @attribute [AllowAnonymous], kræver at brugeren er logget ind. Som det kan ses i 5.3, så bliver forskelligt HTML fremvist alt efter om brugeren er logged ind eller ej.

Listing 5.2: Route-level autorisation

```
1      <CascadingAuthenticationState>
2      <Router AppAssembly="@typeof(App).Assembly">
```

```
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(
                MainLayout)">
                <NotAuthorized>
                    <div>Access Denied</div>
                </NotAuthorized>
                <Authorizing>
                    <div>Authorizing...</div>
                </Authorizing>
            </AuthorizeRouteView>
            <FocusOnNavigate RouteData="@routeData" Selector="h1" />
        </Found>
        ...
    </Router>
</CascadingAuthenticationState>
```

Listing 5.3: Component-level autorisation

```
    <PageTitle>Home</PageTitle>
<AuthorizeView>
    <Authorized>
        <div>
            <UserDetails User=UserAuthenticationStateProvider.CurrentUser />
        </div>
    </Authorized>
    <NotAuthorized>
        <div>
            <p>Please login or create a user.</p>
        </div>
    </NotAuthorized>
</AuthorizeView>
```

Pga. Blazor WebAssembly Standalone App køre rent på klienten, så kan den ikke have adgang til databaser og Seq, da en bruger kunne få fat i de nødvendige forbindelse-informationer. På sammen tid skal data fra en klient altid anses for at kunne være usikker og dermed valideres.
På sammen tid tjekker frontend-delen ikke om login JWT'en er underskrevet med den rigtige nøgle, hvilket betyder at det er muligt for en bad-actor at forfalske en JWT og benytte den til snyde frontend'en til at tror personen er logget ind. REST-API'et ville selvfølgelig fange dette.

5.4 MSSQL

I produktet er der kun en smule sikkerhed på på den valgte SQL-server. Det eneste sikkerhed er, at det er nødvendigt at bruge sysadmin-kontoen for at logger ind, men denne konto kan alt. Dette vil være en sikkerhedsrisiko i virkeligheden.
Den mest optimale sikkerhed ville være at hvert projekt, f.eks. Catering-data-processen, User-API'et osv., har deres egen konto med så mange begrænsninger som muligt. F.eks. kunne det være at ingen at dem kunne oprette, slette og ændre på deres datakontext-tabeller og en udvikler skulle udføre dette eller at deres forbindelse-streng blev udskiftet med en der kunne disse ting, når det var nødvendigt. En anden ting kunne være at sikre sig, at en bruger kun har læse-adgang, hvis den bruger ikke skulle skrive på nogle tidspunkter. F.eks. hvis Catering-delen havde en API til at se retter med bestillinger og tidspunkt, så behøves dens bruger ikke at have skrive-rettigheder, da dette kunne ske igennem RabbitMQ.
På sammen tid ville det selvfølgelig være en god ide at skifte kodeordene til tider, f.eks. kunne appsetting-filen, med login-informationerne, sættes op til at blive læst i run-time af programmet, så filen kunne opdateres uden et genstart.
Når det kommer til det data, der sendes til en SQL-database, er det normalt en god idé at sanitise data'en, f.eks. indsætte en ekstra ' i strenge, hvis en bliver fundet og tjekke for --", da i MSSQL --"er en kommentar. Dette projekt benytter dog EntityFrameworkCore og dermed bliver dette håndteret automatisk.

5.4.1 Seq

Sikkerheden for Seq kan sættes op sådan, at man skal benytte en bruger for at komme ind. I projektet ligger der to powershell scripts, en med bruger sat og en uden. Grunden til dette, er fordi Seq med bruger ikke virkede korrekt på en computer og virkede korrekt på en anden.
I forhold til programmer som skriver til Seq, via Serilog, så benyttes der API-nøgler. Hvis en API-nøgle bliver komprimeret kan den udskiftes.

5.4.2 RabbitMQ

RabbitMQ tillader opsætningen af bruger og det er disse bruger, der bruges til at forbinde med RabbitMQ. Standardbrugeren for alle RabbitMQ har brugernavnet 'guest' og adgangskoden 'guest'. I dette produkt benyttes standardbrugeren pga. det er et test produkt og ikke et rigtig produkt. RabbitMQ klienten, i C♯ , kan få angivet brugernavn og adgangskode, som dermed kan ligge i appsettings filerne.

5.4.3 Appsetting

API-nøgler, kodeord, url'er, og mere vigtig information ligger i appsettings.json filerne i de forskellige projekter. Dette er nødvendigvis ikke det mest sikre måde at gemme sådan data, da appsettings.json, som standard, ikke har noget indkodning og dermed kan læses at alle. Dermed er det en god idé at appsetting.json filerne, der er en del af Git, kun kan bruges til enten lokalle enheder eller test-miljøer. Appsetting filerne, for produktion-miljøet, burde ligge på en sikre placering og aldrig overskrevet, når projekterne bliver udgivet, og disse placeringer er godt beskyttet.
Andre muligheder for sikkerhed, af appsettings, kunne være at placere dem i en keyvault som Azures KeyVault og sætter sikkerhed op, som f.eks. kun enheder med bestemte certifikater kan få adgang.

5.5 General Data Protection Regulation

Der blev ikke gjort noget for at sikre sig, at produktet overholdt General Data Protection Regulation, men for et virkelig produkt ville dette være gjort.

Kapitel 6

Brugervejledning

Dette kapitel vil gennemgå brugervejledningen for begge REST-API'er og frontend'en, samt nødvendig opsætning for at kunne køre alle projekterne.

6.1 Forbindelse

De følgende porte er i brug:

- 5342 - Seq
- 81 - Seq
- 1434 - MSSQL
- 15673 - RabbitMQ
- 5673 - RabbitMQ
- 7298 - User REST-API
- 7142 - Catering REST-API

Hvis MSSQL porten skal udskiftes, skal dette ændres i dens Docker fil, se 6.5 for fil-lokationerne, samt skal forbindelse strengene i UserPlatform/appsettings.json, Catering.DataProcessingPlatform/appsettings.json og Catering.DataProcessingPlatform/IPL/ContextEFFactory ændres.
For Seq skal både dens Docker fil og UserPlatform/appsettings.json og Catering.DataProcessingPlatform/appsettings.json opdateres. For de to appsettings.json filer er det kun nødvendigt, hvis 5342 ændres. Port 81 er dens GUI.
For RabbitMQ skal både dens Docker fil, UserPlatform/appsettings.json og Catering.DataProcessingPlatform/appsettings.json opdateres. For de to appsettings.json filer er det kun nødvendigt, hvis 5673 ændres. Port 15673 er dens GUI.
Hvis User REST-APIs port ændres, skal den også ændres over i Userfrontend.Frontend i program.cs.

6.1.1 Docker og MSSQL

Forbindelse-strengene for MSSQL benytter host.docker.internal til at få den rigtige ip-adresse de skal bruge. Hvis der ikke kan oprette forbindelse, og det ikke er et port-problem, kan det være nødvendig at ændre host.docker.internal til 127.0.0.1.

6.2 Visual Studio

Hvis Visual Studio benyttes, skal Visual Studio 17.8 eller højre benyttes, da dotnet 8 kræver dette.

6.3 Automatiseret Test

Denne forklaring er baseret på Visual Studio 2022. For at køre testene, skal fanen 'Test' vælges og derefter 'Run All Tests'. Det er implementeret 26 test i produktet, alle skulle bestå.

6.4 Powershell

For at lette udviklingen og udgivelse, blev der oprette nogle powershell scrips. For at køre disse, kræves det at Powershells ExecutionPolicy sætts til Bypass, undefined eller unrestricted via Set-ExecutionPolicy <Policy> i Powershell. Hvis dette ikke ønskes at gøre eller ikke kan gøres, så kan teksten i filerne kopiers over til lokalle oprettede filer og kørt eller sat direkte ind i Powershell ISE og kørt igennem ISE'en.

6.5 Docker

Docker skal være installeret og kørende på maskinen, som skal køre produktet.
Der ligger powershell-scripts under mappen /Data/Docker/{rabbitmq/seq/sql}. Når disse køres, vil der blive oprettet de nødvendige docker containers. I seq mappen findes der to powershell filer. Den eneste forskel er, at seqNoPassword ikke sætter en kodeord på Seq. Gruden til dette, er fordi der opstod problemer med SEQ med kodeord på udviklerens ene computer.

6.6 Project-Setup

For at sætte de rigtige projekter til at starte op, skal der højre-klikkes på løsningen 'ApprenticeTest' og vælges 'Properties' i Visual Studio. Under 'Startup Project' skal vælges 'Multiple startup projects:'. Her skal UserFrontend.Frontend, Catering.DataProcessingPlatform, Catering.API og UserPlatform sættes til 'Start' i Action kolonnen.
Før projekterne startes op, burde RabbitMQ docker container'en havde kørt i minimum 30 sekunder for at sikre, at den er klar til at modtage forbindelser.

6.7 Database

For at oprette og tilføre migrations til de nødvendige databaser, henvises der til scriptet 'contexthandler.ps1'[1] i roden af løsningen. Siden migrations allerede findes, skal der trykkes det følgende:

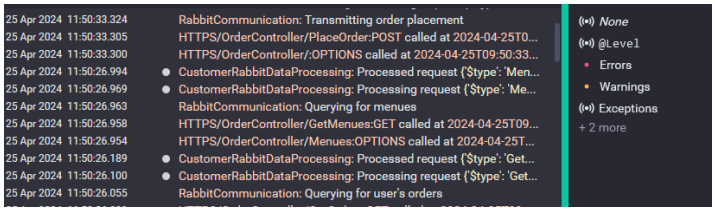
- U -> Enter, U -> Enter
- U -> Enter, C -> Enter

Dette vil skabe databaserne, hvis de ikke findes, og opdaterer dem.
Hvis porten ikke er ændret, kan der oprettes forbindelse til database-serveren på 127.0.0.1,1443, brugernavnet er 'sa' og kodeordet er 'Test123!'.

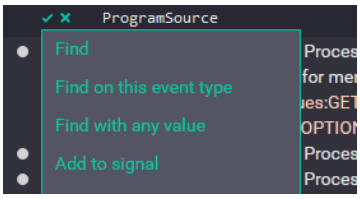
6.8 Seq

For at logge ind på Seq, skal brugernavnet 'admin' og kodeordet 'Test123!' angives. Dens GUI er placeret, hvis porten ikke er blevet ændret, på http://localhost:81.
For at oprette en signal for et bestemt kilde, skal der vælges en log, se figur 6.1, og trykkes på den. Derefter find ProgramSource og tryk på fluebenet. Dette vil åbne en liste, se figur 6.2, og tryk på 'Add to signal'. På højre siden vil en menu åbnes, her skal der trykkes på gemmeikonnet, se figur 6.3. Dette vil placer signallet for kilden under ProgramSource, se figur 6.4, og dermed kan logs fra denne kilde hurtig kunne filtes frem.
For at søge på logs skal query-vinduet i toppen benyttes, se figur 6.5.

[1] For at benytte scriptet, skal maskinen have EntityFramework Core command-line tools installeret.

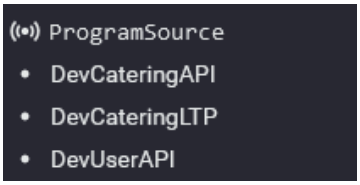


Figur 6.1: Log og signal oversigt

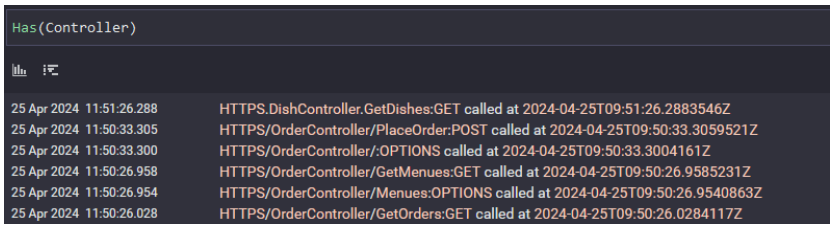


Figur 6.2: Tilfør kilde til signal

Figur 6.3: Oprettelse af signal



Figur 6.4: Liste over kilder



Figur 6.5: Query

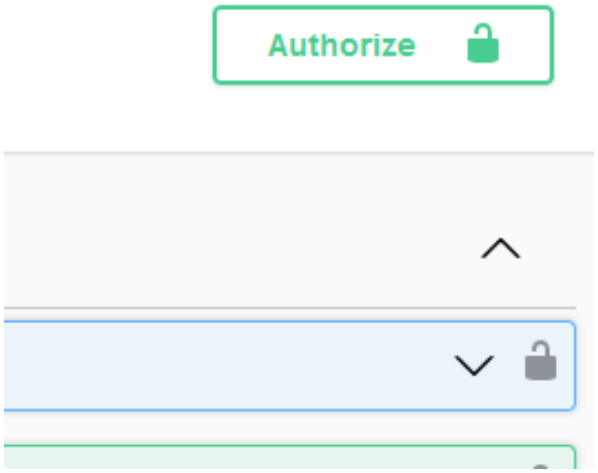
6.9 RabbitMQ

Den brugte RabbitMQ Docker Image kommer med RabbitMQ Management. Denne kan logges ind på http://localhost:15673/, hvis porten ikke er blevet ændret, med brugernavnet 'guest' og kodeordet 'guest'. GUI'en tillader at se oprettede queues, forbindelser, hvor mange beskeder der i systemet og mere.

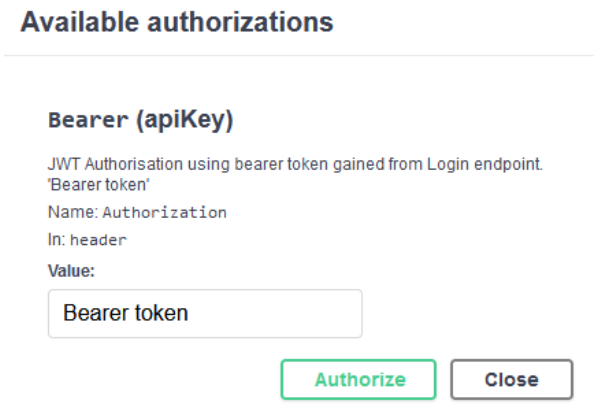
6.10 User

6.10.1 REST-API

Det forventes at brugeren vil benytte den fremviste Swagger for at interagere med API'et. For at benytte REST-API'et kræves det først at brugeren enten benytter Login-endpointet, hvis brugeren allerede har en bruger, eller CreateUser i User-sektionen. Begge endpoints returner et objekt, der indeholde flere felter. Værdien i Token skal bruges til at logge ind med, for at få adgang til alle andre endpoints. For at benytte Token henvises der til figurerne 6.6 og 6.7. Ordet 'token' skal udskifte med strengen i 'Token' fra CreateUser eller Login. Ordet 'Bearer' skal altid være først.
Endpointed Logout i User-sektionen forventer RefreshToken, som kom fra kaldet til Login/CreateUser. Når dette endpoint er kaldt, kan de andre endpoint stadigvæk kaldes indtil login JWT løber ud. Kaldet betyder at den modtaget refresh JWT ikke længere kan bruges i endpointet RefeshToken i User.



Figur 6.6: Login

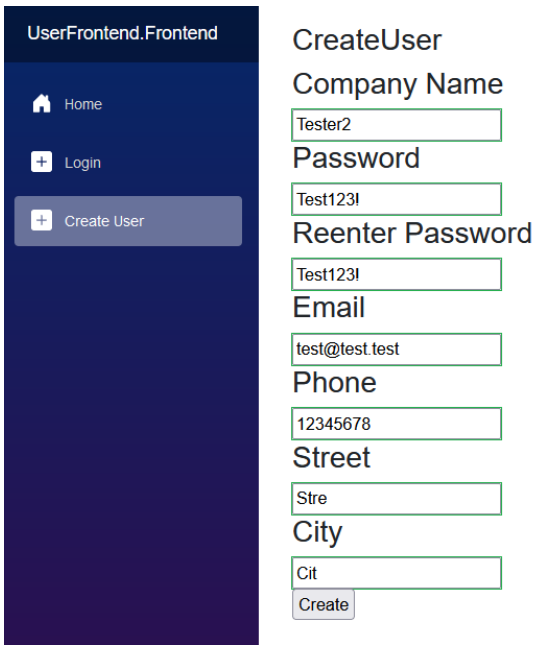


Figur 6.7: Opsætning af JWT

6.10.2 Frontend

Brugeroprettelse

For at oprette en bruger, skal der trykkes på 'Create User' på venstre side og derefter udfylde felterne og trykke på knappen 'Create', se figur 6.8.
Kodeordet skal være mellem 8 og 128 tegn, indeholde minimum 1 stor og 1 lille bogstav, minimum 1 tal og en eller flere af de følgende særlige tegn ! + - # . , * ^.
En eller begge af Phone og Email skal være udfyldt. På nuværende tidspunkt validere koden ikke om de overholder krav for at rigtige telefonnummer eller email-adresse. Hvis brugeren kan oprettes, vil

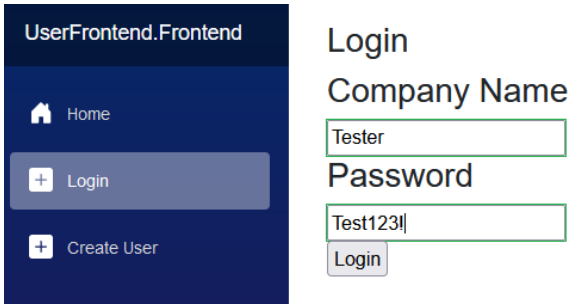


Figur 6.8: Brugeroprettelse

brugeren blive automatisk logged ind og overført til Home-siden.

Login

For at logge ind, skal der tykkes på 'Login' på venstre side, hvorefter felterne skal udfyldes og trykke på 'Login'-knappen. Hvis det lykkes, bliver brugeren automatisk overført til Home-siden.

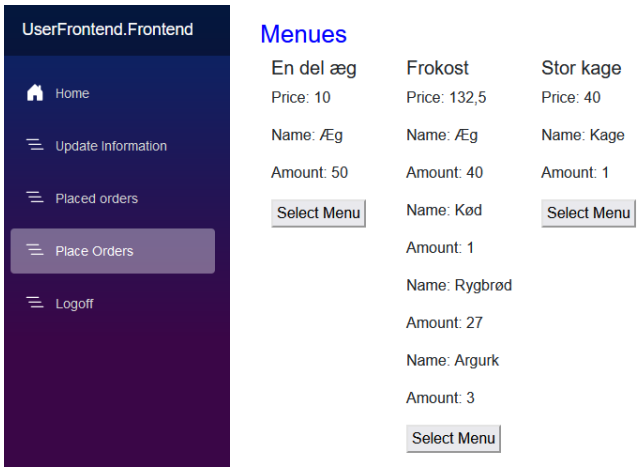


Figur 6.9: Login

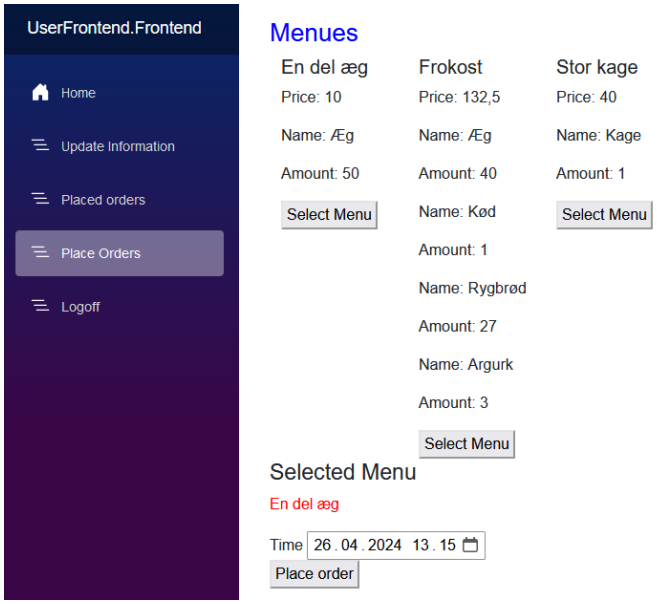
Bestil Menu

For at oprette en bestilling, kræves det at brugeren er logget ind og derefter går til 'Place Orders' på venstre side. På denne side vises alle menuer. For at vælge en menu, skal der trykkes på 'Select Menu'.

Dette vil fremvise en form nederst på siden, hvor brugeren skal indtaste en dato og tid og derefter trykke på knappen 'Place order'. Datoen skal være minimum to dage frem. Hvis ordren bliver oprettet, vil formen forsvinde.



Figur 6.10: Oversigt over menuer



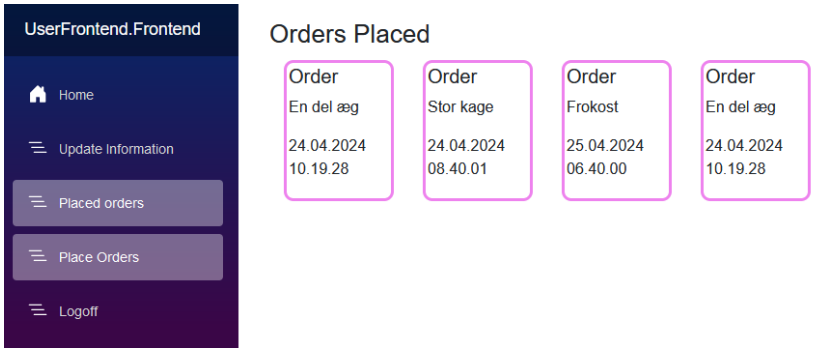
Figur 6.11: En menu valgt

Ordreoversigt

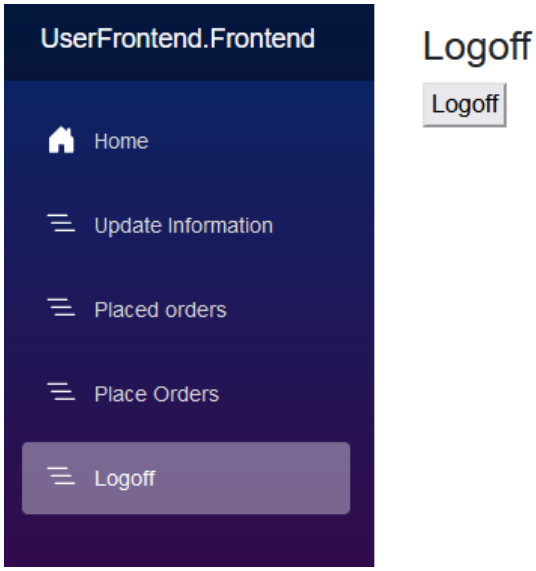
For at se ordreoversigten, kræves det at brugeren er logget ind og derefter vælger 'Placed Orders' på venstre siden. Dette vil åbne en ny side, der fremviser alle ordre brugeren har bestilt, se figur 6.12.

Logud

For at logge ud, skal brugeren først være logged ind. Derefter skal der trykkes på 'Logoff' på venstre siden og derefter på knappen 'Logoff'. Når dette er gjort, vil brugeren blive sendt over til Home-siden, se figur 6.13.



Figur 6.12: Orderoversigt



Figur 6.13: Logud

6.11 Catering

6.11.1 REST-API

REST-API'et for Catering har intet sikkerhed på sig og dermed skal brugeren ikke gøre noget, for at kunne benytte de fremviste endpoints i Swagger.

Referencer

[1] M. Fowler. "Test Double." (2006), webadr.: https://martinfowler.com/bliki/TestDouble.html (hentet 25.04.2024).

[2] Microsoft. "PasswordHasher." (), webadr.: https://source.dot.net/#Microsoft.Extensions.Identity.Core/PasswordHasher.cs,8908fe97c6332908 (hentet 17.04.2024).

[3] Okta. "Introducion to JSON Web Tokens." (), webadr.: https://jwt.io/introduction (hentet 15.04.2024).

[4] B. University. "Singleton dependencies." (), webadr.: https://blazor-university.com/dependency-injection/dependency-lifetimes-and-scopes/singleton-dependencies/ (hentet 16.04.2024).

Bilag A

Kravspecifikationer

Id	Category	Requirement	Priority
Cat-Factory-1	Factory	Entity factories, one for each aggregate root	1
Cat-Factory-1-a	Factory	Can produce Dish	1
Cat-Factory-1-b	Factory	Can produce Menu	1
Cat-Factory-1-c	Factory	Can produce Order	1
Cat-Factory-1-d	Factory	Can produce Customer	1
Result-1	ResultPattern	Implementation of Result Pattern	1
Result-1-a	ResultPattern	Can map to HTTP type	2
Repo-1	RepositoryPattern	Generic interface with CRUD	1
Repo-1-a	RepositoryPattern	Generic interface with CQRS	1
Repo-1-b	RepositoryPattern	Generic interface, constrained to aggregate root	1
Repo-2	RepositoryPattern	Base implementation of generic interfaces	1
Cat-Repo-1	RepositoryPattern	Interface for Unit of Work	1
Cat-Repo-2	RepositoryPattern	Entityframework Unit of Work	2
Cat-Repo-3	RepositoryPattern	Customer EF repo	1
Cat-Repo-4	RepositoryPattern	Menu EF repo	1
Cat-Repo-5	RepositoryPattern	Dish EF repo	1
Cat-Repo-6	RepositoryPattern	Order EF repo	1
Factory-1	Factory	All factories use ResultPattern to prevent handling exceptions	1
Error-1	Error Handling	All errors are logged to an external service	2
Log-1	Logning	All exceptions are logged	2
Cat-Service-1	Service	Can add user	1
Cat-Service-2	Service	Can remove user	1
Cat-Service-3	Service	Can add dish	1
Cat-Service-4	Service	Can remove dish	1
Cat-Service-5	Service	Can add menu	1
Cat-Service-6	Service	Can remove menu	1
Cat-Service-7	Service	Can add order	1
Cat-Service-8	Service	Can remove order	1
Cat-Service-9	Service	Can pull all menues and transform them	1
Cat-Service-10	Service	Can pull single menu	1
Cat-Service-11	Service	Can pull single customer	1
Cat-Com-1	Communication	Can read from RabbitMQ	2
Cat-Com-2	Communication	Can answer a RPC call	3
Cat-Com-2-a	Communication	Can answer RPC call for menues	3
Cat-Service-12	Service	Service for User	1
Cat-Service-13	Service	Service for Menu	1
Cat-Service-14	Service	Service for Dish	1
Cat-Service-15	Service	Service for Order	1
Log-2	Logning	All entity creations are logged	3
Log-3	Logning	All entity deletions are logged	3
Log-4	Logning	All entity changes are logged	3
User-Repo-1	RepositoryPattern	Interface for Unit of Work	1
User-Repo-2	RepositoryPattern	Entityframework Unit of Work	1
User-Repo-3	RepositoryPattern	User EF repo	1
User-Service-1	Service	Can create user	1
User-Service-2	Service	Can remove user	99
User-Service-3	Service	Can login	3
User-Service-4	Service	Can place order	1
User-Service-5	Service	Can see orders	2
Cat-Service-12	Service	Can add orders to menu	1
Cat-Service-13	Service	Can add orders to customer	1
User-Factory-1	Factory	Can produce User	1

Tabel A.1: Kravspecifikationer Del 1

Security-1	Security	Can hash and salt password	1
User-Service-6	Service	Can logout	3
Security-2	Security	Can generate login JWT	1
Security-3	Security	REST-API middleware validates if user has logged in if needed	3
User-Endpoint-1	Endpoint	Can login	1
User-Endpoint-2	Endpoint	Can logout	3
User-Endpoint-3	Endpoint	Can get list of menues	1
User-Endpoint-4	Endpoint	Can place order	1
User-Endpoint-5	Endpoint	Can create user	1
User-Endpoint-6	Endpoint	Can see orders	1
User-Frontend-1	Frontend	Can login if not logged in	3
User-Frontend-2	Frontend	Can logout if logged in	3
User-Frontend-3	Frontend	Can create user if not logged in	3
Security-4	Security	Validate all user input and sanitise as needed	2
User-Frontend	Frontend	Logged in user can see their orders	3
User-Frontend	Frontend	Logged in user can place orders	3
Cat-Rules-1	DDD	Menu cannot be modifed if future orders have been placed	2
Security-4	Security	Password is validated	1
Security-5	Security	Can get salt from salted and hashed password	1
User-Factory-2	Factory	Can produce Token	3
Cat-Endpoint-1	Endpoint	Can get all menues	4
Cat-Endpoint-2	Endpoint	Can get all dishes	4
Cat-Endpoint-3	Endpoint	Can place dish	4
Cat-Endpoint-4	Endpoint	Can place menu	4
User-Endpoint-7	Endpoint	Can update user	4
User-Frontend-4	Frontend	Can update user if logged in	4
User-Frontend-5	Frontend	Can see all menues if logged in	4
Cat-Endpoint-5	Endpoint	Can login	99
Cat-Factory-2	Factory	Can create employee	99
Cat-Service-16	Service	Service for Employee	99
Cat-Repo-7	RepositoryPattern	Employee EF repo	99
Cat-Com-2-b	Communication	Can answer RPC call for customer orders	3
Cat-Com-3	Communication	Only got RPC calls	4
Cat-Com-3-a	Communication	Can fetch dishes	4
Cat-Com-3-b	Communication	Can fetch menues	4
Cat-Com-3-c	Communication	Can create menues	4
Cat-Com-3-d	Communication	Can create dishes	4
Cat-Com-2-c	Communication	Can fetch menues for display	3
Cat-Com-2-d	Communication	Can place order	3
Cat-Com-2-e	Communication	Can create customer	3
Cat-Com-2-f	Communication	Can update customer	3
Cat-Com-3-e	Communication	Can create employee	99
Cat-Service-16	Service	Can update customer's location	3
User-Service-6	Service	Can update user's location	3
Security-6	Security	Employees have different roles	99
Security-6-a	Security	Can only see orders	99
Security-6-b	Security	Can create and modify dishes and menues	99

Tabel A.2: Kravspecifikationer Del 2

Bilag B

Test Sager

Id	Description	Criteria
Cat-Factory-1-a	Get ResultDish	Should return either SuccessResultt or BadRequest
Cat-Factory-1-b	Get ResultMenu	Should return either SuccessResultt or BadRequest
Cat-Factory-1-c	Get ResultOrder	Should return either SuccessResultt or BadRequest
Cat-Factory-1-d	Get ResultCustomer	Should return either SuccessResultt or BadRequest
Cat-Service-1	Create customer and get Result	Throws no exceptions
Cat-Service-3	Create dish and get Result	Throws no exceptions
Cat-Service-5	Create menu and get Result	Throws no exceptions
Cat-Service-7	Create order and get Result	Throws no exceptions
Cat-Service-9	Can get all orders transformed	Collection of data, if none found empty
Cat-Service-12	When creating order, add id to menu	Throws no exceptions
Cat-Service-13	When creating order, add id to customer	Throws no exceptions
Log-2	Read logs for each method call and chek if any logs are present	A single log should be present
Log-3	Read logs for each method call and chek if any logs are present	A single log should be present
Log-4	Read logs for each method call and chek if any logs are present	A single log should be present
User-Factory-1	Get ResultUser	Should return either SuccessResultt or BadRequest
User-Service-1	Create user and get Result	Throws no exceptions
User-Service-3	Can login and gets a token	It should take a minimum of 500 ms every time and the token should be valid
User-Service-4	Can place order	Throws no exceptions
User-Service-5	Can see orders	Throws no exceptions
Security-1	Can hash and salt a password	Password should be unique for each run
Security-2	Can generate a JWT	Only generate JWT for valid login else BadRequest
User-Endpoint-3	Returns a collection of transformed menues or empty	Throws no exceptions
User-Endpoint-4	Can place an order	Throws no exceptions
Cat-Rules-1	Cannot modify menu if it is in use	Returns InvalidRequest if the menu has orders for today or in the future
Security-5	Can relog in	Can use the salt of a salted-hashed password to generate the same salted-hashed password
Security-4	The password requirements are tested	Each rule broken rule should return a unique binary value
User-Factory-2	Get ResultToken	Should return either SuccessResultt or BadRequest
User-Service-6	User location has been changed.	Null and white-space strings should not change the location
Cat-Service-16	Customer location has been changed.	Null and white-space strings should not change the location
User-Endpoint-1	Attemp to log in	200 with Auth data if valid user else 401
Result-1-a	Mapping ResultT/Result to correct HTTPResponse	Correct mapping, 200 should have data, all non-200 and non-204 should carry the errors
User-Factory-1	Get ResultUser	Should return either SuccessResultt or BadRequest

Tabel B.1: Test Cases