CDIO Final

Gruppe 21

Marie Seindal
s185363

Peter Revsbech
s183760

Christian Kyed
s184210

Ida Schrader
s195483

Anthony Haidari
s141479

Danmarks Tekniske Universitet

CDIO-opgave

02324 Videregående programmering F20

GitHub: `https://github.com/Software2020Hold21/21_CDIOFinal/`

26/06-2020

1 Timeregnskab

	Peter	Marie	Christian	Ida	Anthony
08-06	6	6	6	6	6
09-06	8	8	8	8	8
10-06	8	8	8	8	8
11-06	8	8	8	8	8
12-06	8	8	8	8	8
13-06					
14-06					
15-06	8	8	8	8	8
16-06	8	8	8	8	8
17-06	8	8	8	8	8
18-06	8	8	8	8	8
19-06	8	8	8	8	8
20-06					
21-06					
22-06	8	8	8	8	8
23-06	8	8	8	8	8
24-06	8	8	8	8	8
25-06	2	2	2	2	2
TOTAL	104	104	104	104	104

Figur 2: Timeregnskab

2 Abstract

This is a paper done in june 2020 at the Danish Technical University. It focuses on the creation of a softwaresystem for a medical company. The softwaresystem is used by 4 different types of staff in the company: Laboratory Technicians, Production Managers, Pharmaceuts and an Administrator. The overall function of the program, is to maintain data for creation medical recipies and helt the users save these data correctly. The software used is MySQL for the database and a restful API build with JQuery. The UI is a webpage build with html, css and javascript. The paper has three major steps: analyzation, design and implementation. For analyzing and designing we have used UML diagrams. The implementation is described and discussed throughout the chapters.

Indhold

3 Indledning

Dette er indledningen på en rapport på baggrund af et projekt udfærdiget i juni 2020 på kurset 02324, Videregående Programmering, Danmarks Tekniske Universitet. Vi har fået til opgave at udarbejde et fungerende system til en medicinalvirksomhed med et user interface i form af en hjemmeside. Softwaresystemet skal implementeres således at dette skal kunne anvendes af 4 aktører, med 6 overordnede brugsscenarier: Brugeradministration, råvareadministration, receptadministration, råvarebatchadministration, produktionbatchadministration og afvejning.

Rapporten er udarbejdet med henblik på at skabe et overblik, over de valg vi har truffet med hensyn til design og selve implementeringen. Da det har været et krav fra DTU at der skulle stå forfatter på de individueller afsnit har vi valgt at placere disse i fodnoter ved hver overskrift. Det kan også siges om vores timeregnskab at vi alle i gruppen har arbejdet alle hverdage fra 8-16, og derfor har haft lige stor indflydelse på projektet.

Lige efter denne indledning findes en guide til opsætning af systemet. Ellers består rapporten af tre store dele, et konfigurationsafsnit, et testafsnit og en konklusion. I bilagene findes hele SQL-scriptet med tabeldefinitioner og testdata.

I den første del starter vi ud med at analysere kundens krav, og udarbejder en kravsspecifikation i prioriteret rækkefølge samt usecases og domænemodel. Denne fase har vi vendt tilbage til flere gange i løbet af udviklingen, for at konsultere og redigere i krav eftersom vi løbende fik mere forståelse for domænet. Derefter gennemgås designet af systemet, med mellemliggende diskussioner af fordele og ulemper ved de valg vi har truffet. Her kan man se et klassediagram for Data Transfer Object- klasserne og et databaserelateret EER-diagram. Vi ønsker i disse to afsnit at tydeliggøre hvordan systemet udvikler sig fra abstraktion til konkrete javaklasser.

Den tredje store del af rapporten redegør i detaljer for vores implementering af systemet. Her findes mange eksempler fra koden og gennemgange deraf. Den er ydermere delt op i to underkategorier: front end og back end, hvor især front enden tager afsæt i de fire roller. På samme måde som i designafsnittet er her også smådiskussioner undervejs hvor de er relevante. Efter implementeringen kommer et afsnit om maven, git og systemkrav efterfulgt af et Testafsnit.

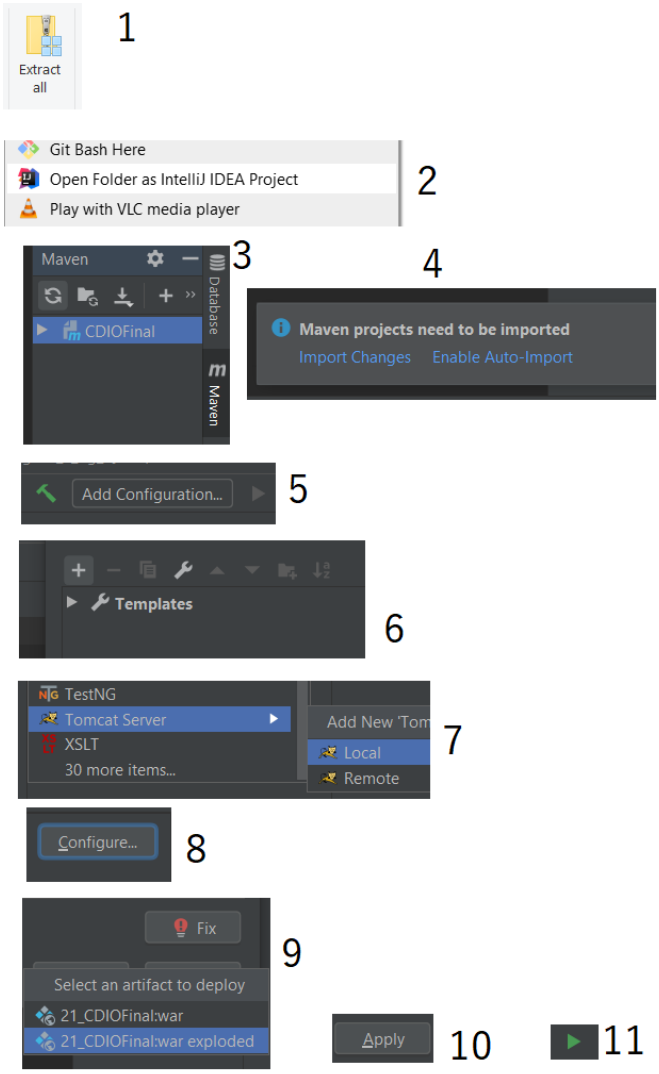
Sidst kommer en konklusion som indeholder en opsamling på projektet, både ifht det færdige produkt og arbejdsprocessen. Her står lidt om hvad der gik godt og skidt, og hvordan man kunne have forbedret. God læselyst.

4 Opsætning[1]

Som forudsætning for at køre programmet skal man have installeret IntelliJ som kører java version 1.8, og Apache Tomcat version 8.5.11. Når man åbner projektet første gang er det nødvendigt at definiere hvilken SDK der benyttes. Dette indstiller man i File - Project Structure - Project og derefter kan man vælge Project SDK

Download zip filen, pak den ud (fig 1) og åbn projektmappen i intelliJ ved at højreklikke på den (fig 2).
Klik på maven ikonet til højre og tryk på de to små pile (reimport all maven projects, fig 3). Hvis der kommer en pop-up i bunden af siden så tryk på Enable Auto-Import (fig 4).

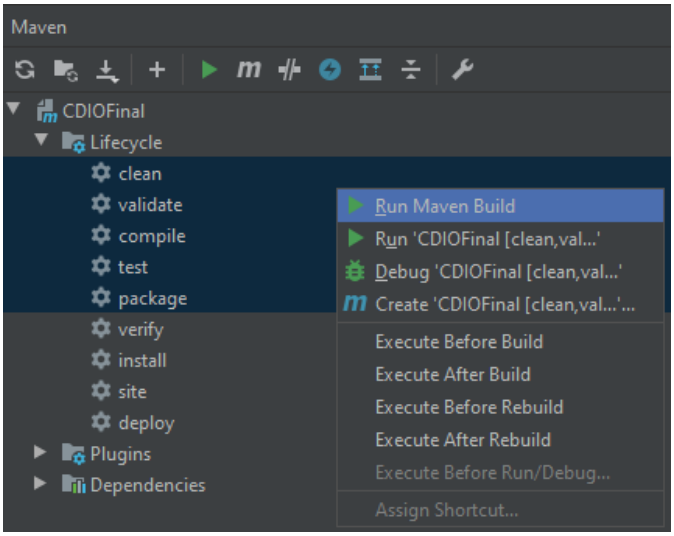
Tryk derefter i øverste højre hjørne på Add Configuration (fig 5). Derefter klik på det lille kryds i venstre hjørne af boksen (fig 6), scroll ned og find Tomcat Server, vælg Local Server (fig 7). Tryk evt på configure for at angive stien til tomcat (fig 8). Når det er gjort så find knappen Fix i nederste højre hjørne og vælg den mulighed der slutter med :war exploded (fig 9). Klik på apply (fig 10). Til sidst klik på den grønne play-knap, og da burde siden gerne komme frem.



Figur 3: Guide til opsætning

[1]Forfatter: Ida

For at projekt skal virke skal det første bygges vha Maven. Klik derfor på Maven ude i højre side af IntelliJ. Marker derefter følgende felter vist på billedet, højreklik og tryk på Run Maven Build



Figur 4: Build med Maven

4.1 Nuværende brugere i systemet

Til at teste systemet, er det nødvendigt at logge ind med en eksisterende bruger. En oversigt over disse kan ses nedenfor. Der er tilføjet en del brugere af typen Administrator. Dette er for at undgå at man "låser"sig selv ude, hvis man sætter Administratoren til inaktiv når systemet bliver testet.

BrugerID	Rolle
1	Administrator
2	Farmaceut
3	Produktionsleder
4	Laborant
5	Administrator
6	Administrator
7	Administrator
8	Administrator

5 Krav[2]

Under analyse af opgaveformulering har vi udarbejdet en kravsspecifikation, som først præsenteres og derefter efterfølges af en kort diskussion på udvalgte punkter.
Vores krav er opdelt i 2 hovedkategorier:

- **Funktionelle krav**
- **Ikke-funktionelle krav**

Derudover er kravene opdelt i 3 kategorier, der viser kravenes prioritet.

- **Prioritet A**
  Krav som står under denne prioritet anses som de mest basale funktionaliteter, der er ikke taget hensyn til brugervenlighed endnu. Kravene kan forekomme løst formuleret, da domænet var relativt nyt for alle udviklere, og derfor vil nogle af disse krav gå igen mere specifikke i senere prioriteter.
- **Prioritet B**
  Krav under denne prioritet tager i højere grad hensyn til brugervenlighed, og begynder at tage højde for at opfylde nogle mere detaljerede krav til funktionalitet.
- **Prioritet C**
  Krav under denne prioritet er krav der har været nedprioriteret i arbejdsprocessen af forskellige årsager, mest af alt pga tidspres og ressourcemangel. Den mest optimerede udgave af systemet havde implementeret alle disse.

Kravlisten vil benytte den syntax, at alle krav med en * ud for sit krav-ID er krav, gruppen ikke har nået at efterleve.

5.1 Funktionelle krav[3]

A

K1 Interfacet skal kun kunne bruges som én af følgende fire roller: Administrator, Produktionsleder, Farmaceut eller Laborant

K2 På forsiden kan man frit vælge hvilken rolle man vil agere som

K3 Alle ID'er af en bestemt type er unikke

K4 Brugeradministrator har rettighed til at se, oprette og redigere brugere

K5 En bruger kan ikke slettes, men derimod deaktiveres

K6 Farmaceut kan se, oprette og redigere i råvarer

K7 Farmaceut kan se og oprette recepter

K8 Produktionsleder kan oprette råvarebatches og produktbatches

K9 Laboranten skal kunne afveje råvarer til produktbatches

K10 Laboranten kan ikke redigere i et afvejet produktbatch når det er afsluttet

K11 En råvare består af et unikt råvareID, navn og leverandør

K12 En råvarebatch består af unikt ID, mængde og en råvare fra databasen

K13 En recept består af en eller flere råvarer inkl ID, nettovægt i kg og tolerance i procent

[2]Forfatter: Ida
[3]Forfatter: Ida

K14 En produktbatch består af et unikt ID, en dato for oprettelse og en recept

B

K15 Man registrere sig som en rolle ved at indtaste brugerID på forsiden og bliver automatisk sendt videre afhængig af den rolle der er knyttet til ID'et

K16 Når en bruger er deaktiveret kan brugeren ikke logge ind

K17 Brugeradministrator har udover egne rettigheder samme rettigheder som de andre tre roller

K18 Farmaceut har udover egne rettigheder også rettigheder svarende til produktionsleder og laborant

K19 Produktionsleder har udover egne rettigheder også rettigheder svarende til laborant

K20 Laborant skal godkende sit brugerID og navn før vedkommende kan begynde afvejning

K21 Når et produktbatch oprettes sættes tilstanden til "Pprettet"

K22 Når laboranten begynder afvejning sættes tilstanden til "Under produktion"

K23 Når laboranten har afvejet sidste ingrediens og gemmer til databasen sættes tilstanden til "Afsluttet"

K24 Lagerstyring: beholdningen af alle råvarer i hvert batch skal løbende opdateres

C

K25 Når et produktbatch er oprettet skal det kunne printes til PDF med oplysninger som står beskrevet i "Krav til kvittering"

K26 Når laboranten indtaster vægt meddeles om det indtastede er indenfor tolerancen

K27 Hvis en bruger forsøger at benytte et ID, som allerede er i brug, skal systemet komme med en fejlmeddelelse

K28 Når laboranten har indtastet recept ID, kan systemet lave en drop-down menu med de ingredienser der skal bruges

K29 * Når laboranten har valgt en ingrediens at afveje skrives mængdeintervallet på siden

K30 * Når et råvarebatch er brugt op skal det ikke være en gyldig valgmulighed for laboranter fremover

K31 * Hvis der kun er én aktiv brugeradministrator tilbage kan den ikke deaktiveres

5.2 Ikke-funktionelle krav[4]

K32 Data gemmes i en database på internettet

K33 User interfacet er en hjemmeside

K34 Databasen og interfacet snakker gennem en restful API

[4]Forfatter: Christian

5.2.1 Krav til kvittering

Kvitteringen indeholder...

K35 Dato for print af kvittering

K36 Produktion Status (Startet/Løbende/Afsluttet)

K37 Recept ID

K38 Produktbatch ID

K39 For hver råvare:

- Råvarenavn
- RåvareID
- Mængde i kg
- Tolerance i procent
- Vægtens tara i kg
- Nettovægt i kg
- Råvarebatch ID
- Laborantens ID

5.3 Diskussion af krav[5]

På nogle områder afviger kravsspecifikationen fra opgavebeskrivelsen. Disse begrundes her. Brugeradministrator har adgang til alle funktioner, da vi synes det giver god mening at der findes en brugertype som har alle rettigheder. Der kan dog også sagtens argumenteres for at det ikke er gavnligt at en administrator har rettigheder som en farmaceut uden at have den faglige kundskab. En anden lille hage der kan opstå i denne situation er, at en brugeradministrator kan komme til at inaktivere sig selv, og dermed låse sig ude af systemet. I den situation må man henvende sig til os, som leverandører af systemet med adgang til databasen.

Kvitteringen mangler enkelte informationer, som f.eks. terminal og specifikke tidspunkter for oprettelse og afslutning. Det fremgår ikke af opgaveoplægget hvad "terminal"indikerer, og dette er derfor ikke blevet antaget som værende vigtigt. Tidspunkterne for de individuelle afvejninger er også blevet nedprioriteret, da det ville tilføje ekstra data til databasen, mens det ikke har været tydeligt af opgaveoplægget, om det overhovedet er vigtigt.

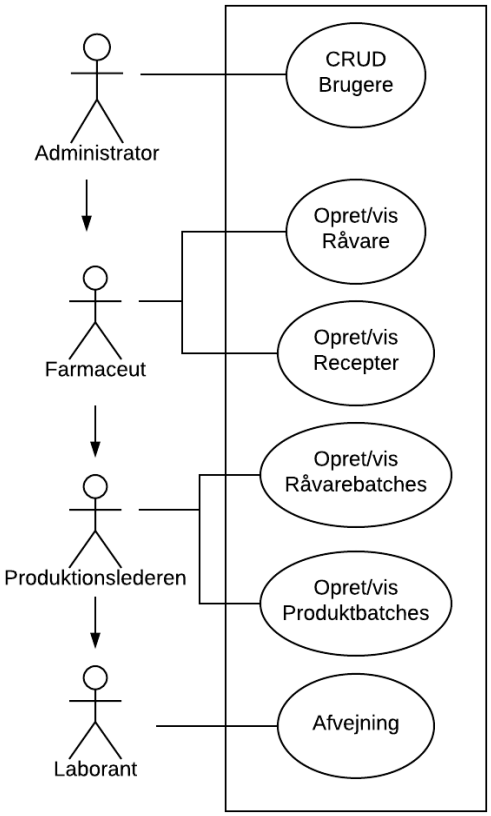
[5]Forfatter: Ida

6 Analyse

6.1 Use Case Model[6]

I dette afsnit, betragtes de forkellige usecases der kan opstå ved brug af systemet. De fleste af disse vil blive beskrevet på et "brief"niveau, mens der vil blive gået mere i dybden med nogle enkelte. I figur 5 ses et UseCase-diagram, som illustrerer sammenhængen mellem systemets aktører og use cases.

6.1.1 Brief beksrivelse af alle usecases[7]



Figur 5: UseCase-Diagram

1. UC1: Bruger administration
   Administratoren logger ind på hjemmesiden og vælger rollen "Administrator", og kan nu se en oversigt over alle systemets brugere. Derefter Opretter administratoren en bruger i systemet. Når brugeren er oprettet, kan den redigeres og aktiveres/deaktiveres efter behov.

2. UC2: Råvare administration
   Farmaceuten logger ind i systemet og navigerer til råvareadministration. Derefter opretter faramaceuten en råvare og får vist listen over råvarer.

3. UC3: Recept Administration
   Farmaceuten logger ind i systemet og navigerer til receptadministration. Derefter opretter faramaceuten en recept og får vist liste over recepter.

[6]Forfatter: Marie
[7]Forfatter: Christian

4. UC4: Råvarebatch Administration
   Produktionslederen logger ind i systemet og navigerer til Råvarebatch Administration. Derefter opretter produktionslederen en råvarebatchs og får vist liste over de eksistrerende råvarebatches.

5. UC5: Produkt Batch Administration
   Produktionslederen logger ind i systemet og navigerer til at oprette produktbatches. Produktionslederen angiver et ID på den recept, produktbatchen skal laves ud fra, og får vist en forhåndsvisning som kan udprintes og gives til en laborant, hvorefter produktbatchen gemmes i systemet. Efterfølgende ser produktionslederen i listen over recepter og produktbatches den nye produktbatch.

6. UC6: Afvejning
   Laboranten logger in i systemet, vælger en produktbatch og enten begynder eller fortsætter afvejningen af denne. For hver ikke-afvejet råvare i produktbatchen, indtaster laboranten data for en afvejning til systemet, og hvis produktbatchens sidste råvare afvejes, kan den afsluttes. De afvejede produktbatches mængder bliver opdateret undervejs.

6.1.2 UC1 og UC3 - Casual beskrivelse[8]

UC1 og UC3 vil i dette afsnit blive beskrevet på et casual niveau.

Usecase:	UC1
Titel	Bruger Administration
Primør aktør	Administrator
Postconditions	En bruger er oprettet, redigeret og deaktiveret
Main success scenario	Administrtoren logger ind i systemet. Derefter vælger administratoren "brugeradministration". Administratoren opretter en bruger med et unik ID og de andre tilhørende attributter. Systemet hjælper vha. fejlbeskeder administratoren til at vælge korrekte værdier til attributterne. Administratoren klikker på "gem". Administratoren vælger at redigere en eller flere attributter for en bruger med et specifikt id. Når brugeren ikke længere skal bruges, sættes dennes status til "inaktiv".

Usecase:	UC3
Titel	Recept Administration
Primør aktør	Farmaceut
Postconditions	Farmaceuten har oprettet en recept og fået vist recepten
Main success scenario	Farmaceuten logger ind i systemet, med sit brugerID, hvorefter de navigerer til "Opret Recept". Farmaceuten indtaster da ID, navn og udfylder receptlinjerne der skal bruges. Efter recepten er oprettet, går farmaceuten tilbage til farmaceut-forsiden og vælger "Vis alle recepter", hvorefter farmaceuten ser alle recepter. På samme vis går farmaceuten tilbage til farmaceut-forsiden og vælger "Se indholdet af en recept". Her indtaster farmaceuten et specifikt receptID og kan da se indholdet af den ønskede recept.

[8] Forfatter: Marie

6.1.3 UC6 Afvejning - Detaljeret beskrivelse[9]

UC6 omkring afvejning foretaget at laboranter, vil nu blive uddybet i større detaljeringsgrad, da denne kan virke noget kompliceret.

Usecase	UC6
Titel	Afvejning
Primær aktør	Laborant
Preconditions	En produktionsleder har oprettet en produktbatch.
Postconditions	Produktbatchen er afsluttet, og alle dens råvarer er afmålt.
Flow	1. Laboranten logger ind i systemet ved at indtaste sit ID. 2. Systemet beder laboranten godkende sin identitet. 3. Laboranten godkender. 4. Systemet beder laboranten angive ID på den produkbatch, der skal afvejes. 5. Laboranten angiver et gyldigt ID. 6. Systemet fremviser produktbatchen i sin nuværende status. 7. For hver uafvejet komponent af produktbatchen gør laboranten nu følgende. 7.1 Laboranten indtaster data for den foretagede afvejning. 7.2 Systemet kontrollerer, at afvejningen er udført korrekt. Dette indebærer følgende: 7.2.1 Det kontrolleres, at alle inputs kommer fra gyldige domæner. 7.2.2 Det kontrolleres, at den angivne råvarebatch eksisterer. 7.2.3 Det kontrolleres, at den angivne råvarebatch omhandler den afvejede råvare. 7.2.4 Det kontrolleres, at den angivne råvarebatchs mængde ikke er mindre end den målte nettovægt. 7.2.5 Det kontrolleres, at den målte nettovægt ligger inden for receptens tolerance. 7.3 Hvis den er udført korrekt, opdaterer systemet råvarebatchens mængde. 7.4 Hvis der stadig mangler flere afvejninger, gentages processen for en ny råvare (7.1). 7.5 Hvis den ikke er udført korrekt, bedes laboranten forsøge igen (7.1) 8. Når alle råvarer er afvejet korrekt, vælger laboranten af afslutte produktbatchen.

6.1.4 Domænemodel[10]

Der er blevet lavet et domæne-klassediagram (figur 6) ud fra opgaveoplæggets beskrivelse af de objekter som indgår i systemet. Objekternes relationer er fundet ved at læse opgavebeskrivelsen og analysen af af usecases.
De omtalte objekter er som følgende:

- Recept
- Raavare
- RaavareBatch
- ProduktBatch
- Bruger

Domænet vil nu blive beskrevet startende ved en recept. Recepten har, som det fremgår af figuren, et ID og nogle komponenter. Relationen til Råvarer indikerer, at receptens komponenter indeholder hver en råvare. Forholdet er derfor en-til-mange. I praksis fungerer det ved, at der til at lave en recept skal bruges nogle bestemte råvarer i bestemte mængder.

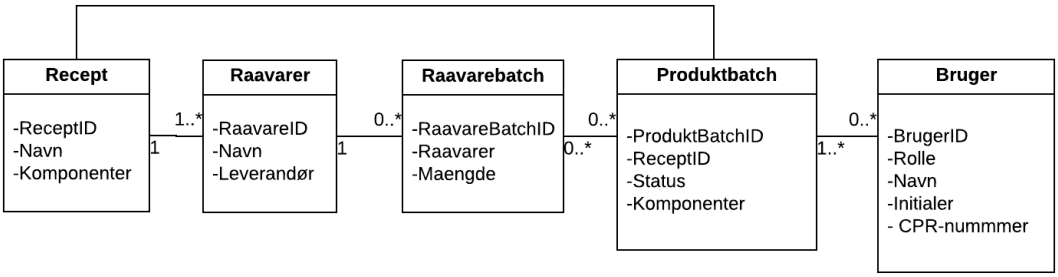
Råvarer fås fra RåvareBatches. I en bestemt RåvareBatch produceres netop én råvare i en bestemt mængde. Batchens mængde holder løbende styr på, hvor stor den resterende mængde af

[9] Forfatter: Christian
[10] Forfatter: Marie

batchen er.

Et produktbatch er et forsøg på at producere én bestemt recept. Produktbatchet knytter sig derfor til én recept, hvorigennem den knyttes til en-til-mange Råvarer. Hver Råvare, skal tages fra et bestemt RåvareBatch, afvejes og indgå i en komponent af ProduktBatchet. Når Produkt-Batchet produceres, tages altså noget af mængden af de pågældende RåvareBatches altså.

Brugere har en relation til ProduktBatches, fordi der altid er en bestemt bruger, som afvejer en bestemt komponent af produktbatchet. Dette vil for det meste være en laborant.



Figur 6: Domæne klassediagram

Som det kan ses ud fra navngivningen i domænemodellen (Figur 6) er elementerne skrevet på dansk. Dette er et valg vi har taget, da mange af navnene i forvejen var givet på dansk. For at mindske forvirring og rod i koden, benytter navngivningen derfor så vidt muligt fagtermerne, som blev givet i opgaveoplægget. Bogstaverne æ, ø og å er skrevet som ae, oe og aa.

7 Design

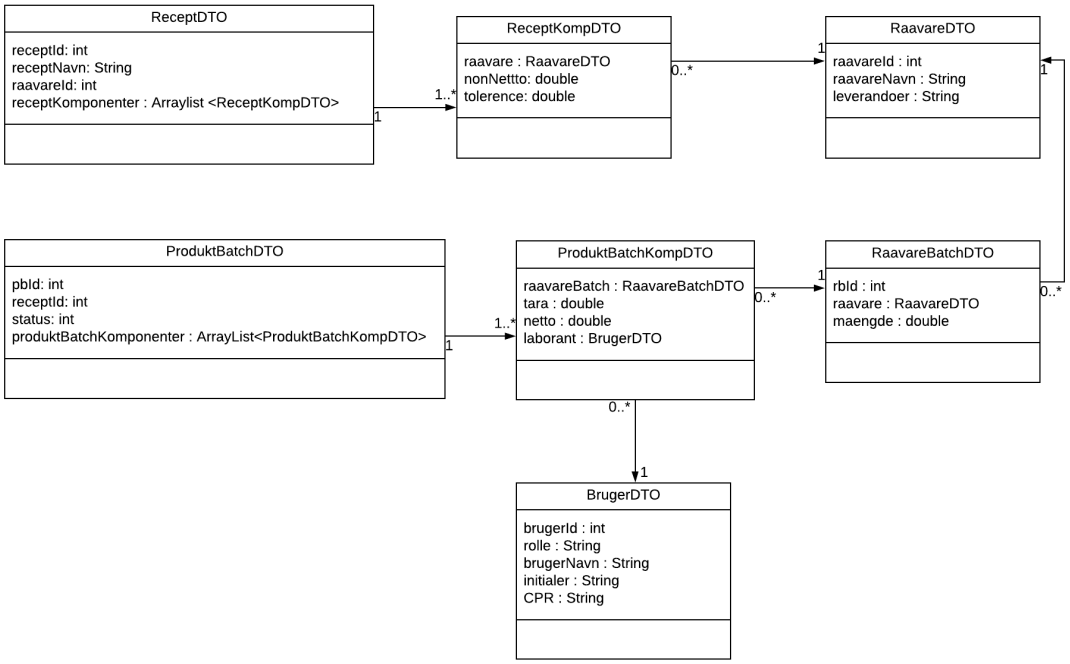
I dette afsnit, vil det blive beskrevet, hvilke designmønstre, der er blevet benyttet i systemet, samt hvilke valg og overvejelser der ligger til grund for det. Afsnittet er opdelt i underafsnit svarende til programmets lag-deling.

Lagdelingen tager udgangspunkt i en 2-lags-model med et datalag og et UI-lag. Den klassiske 3-lags-model, som også inkluderer et business-logic-lag bliver altså ikke benyttet her. Programmets business-logic er nemlig så simpel, at det ikke har været set som nødvendigt at lave et decideret lag til dette. Den programlogik som f.eks. validerer brugerinputs er derfor delt ud i backend og frontend-delene.

Udenfor hele lagdeling ligger DTO-klasserne, der er essentielle for alle systemts dele. Disse vil blive gennemgået først.

7.1 Data Transfer Objects[11]

Analysen af systemets domæne i udviklingen af domænemodellen har givet et overblik over de "entiteter"som det færdige program skal behandlde, bl.a. brugere, råvarer osv. Disse entiteter er centrale for alle dele af programmet, og skal behandles i alle lag - både i front end-delen og i backend-delen. Der er derfor blevet udviklet 7 Javaklasser til 7 DTO'er (Data Transfer Objects). Et UML-klassediagram over disses ses i figur 7.



Figur 7: Klassediagram for DTO'erne

De 7 entiteter svarer altså her til 7 javaklasser. I diagrammet ses disse beskrevet ved deres attributter, mens de ingen metoder har. De skal altså ikke ses som "klasser"i traditionel objekt-orienteret forstand, der har ansvarsområder osv. De er blot beholdere, som skal indeholde data (Svarende til en Struct i programmeringssproget C).

[11] Forfatter: Peter

De fleste af relationerne mellem DTO'erne er designet sådan, at et objekt af den ene klasse indeholder et objekt den klasse, den har en relation til. Eksempelvis indeholder en ReceptDTO en liste af dens komponenter. Hver af disse indeholder den råvare, der står i recept-komponenten osv.

Der er både fordele og ulemper ved dette design. Den største fordel er, at når f.eks. recepten hentes til front-end-delen fra databasen med et GET-kald, indeholder den alt det data der er vigtigt for den. Det er ikke nødvendigt at lave yderligere GET-kald eller søgninger efter ID for f.eks. at finde dens komponenter.

En ulempe ved dette design er, at de JSON-objekter der bliver sendt frem og tilbage mellem klienten og serveren er store og til tider unødigt store. Nogle steder i programmet er det måske unødvendigt at have alle receptens komponenters råvarerer, da det bare er recepten og dens navn, der skal vises. I så fald, har det taget unødigt lang tid at hente så meget data.

Kompromiset for disse fordele og ulemper blev, at langt de fleste relationer håndteres ved at det ene objekt indeholder det relaterede objekt. Den eneste undtagelse er ProduktBatchDTO'en som ikke indeholder den recept, der er lavet ud fra, men blot dens ID. Det skyldes at netop ProduktBatchDTO og ReceptDTO i forvejen er programmets "tungeste"klasser, altså dem som indeholder mest data.

Måden DTO'erne er implementeret på vil selvølgelig blive vist i implemeterings-afsnittet, men endnu en vigtig pointe er, at de er lavet sådan at de kan serialiseres, og dermed formatteres som JSON-objekter. Det muliggør, at de også kan behandldes som objekter i front-end-delen i Javascript. Vores RESTful Webservice benytter netop JSON-formatet til payloadet i de 4 vigtigste HTTP-metoder, hhv. GET, POST, PUT og DELETE.

7.2 Den relationelle SQL database[12]

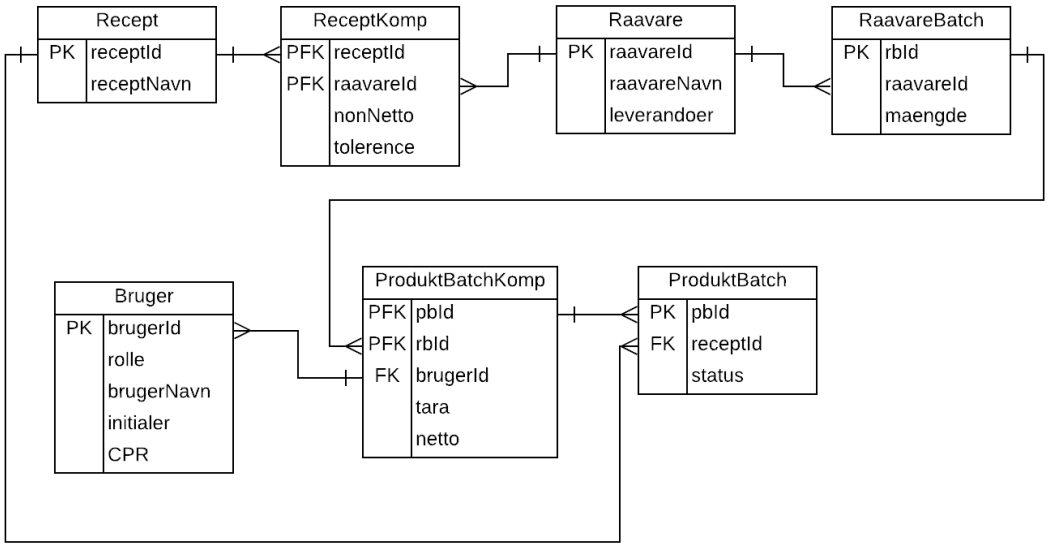
Extended entity relation model (EER-model) er anvendt til design af databasen. Denne model repræsenterer databasens entiteter og attributter og hvordan disse er relateret i forhold til hinanden. Måden vi har påtaget opgaven, at lave et funktionelt database design på, er at vi først og fremmest har kigget på de krav der er blevet stillet til det samlede softwaresystem og hvilke data der skal gemmes. Vha. den statiske domænemodel udarbejdede vi databasedesignet som ses i figur 8.

EER-diagrammet viser nemlig 7 entiteter med deres respektive attributer, svarende til de 7 javaklasser i DTO-laget, der udgør vores programs database som helhed. Som det fremgår af EER-Diagrammet er databasen en relational database der består af 7 tabeller. Disse tabeller repræsenterer både data og forholdet mellem disse. Hver tabel har flere kolonner, og hver kolonne har et unikt navn. For eksempel, tabellen "Bruger"har i alt 5 kolonner, svarende til attributternes navne og hver række der består af disse 5 attributter har et unikt navn, som i dette tilfælde er attributten "brugerId". Relationerne mellem disse tabeller er angivet vha. linjerne der forbinder to eller flere tabeller (entiteter) med hinanden.

Derudover ses det af EER-diagramet at for hvert binært forhold er der angivet cardinaliteter. Cardinaliteter er et udtryk for antallet af entiteter, som en anden entitet kan tilknyttes via et forholdssæt. For eksempel beskrives forholdet mellem entiteter "ReceptKomp"og "RaavareBatch"vha. entiteten "Raavare". Cardinaliteten er mange-til-mange. Det betyder at en råvare kan indgå i en eller mange Råvarebatch- og ligeledes kan en råvare indgå i en eller mange Receptkomponenter.

Denne måde at tilgå data på og gemme data i MySQL databasessystem adskiller sig signifikant fra måden data gemmes på i DTO'erne. Som det også er blevet beskrevet i sektionen ovenover er DTO'erne blevet designet således at en instans af en klasse indeholder et objekt af den klasse, den har relation til. Dette design gøre at de Json-objekter der indeholder data kan til tider være

[12]Forfatter: Anthony



Figur 8: EER-Diagram

ret omfangsrige. Det er ikke altid man har behov for at indhente al data, hvorfor tilknytning til MySQL database gøre det muligt at indhente data efter behov. På den måde undgår man redundans i databasen, og sørger for at alle dele er opdateres synkront.

7.3 Backend applikationen i Java[13]

Backend-delen af programmet er temmelig omfattende i den forstand at den indeholder mange java-klasser, og at disse hver indeholder meget kode. Relationerne mellem klasserne er imidlertid meget simple, idet der stort set er et mønster, som gentages 7 gange - en for hver af programmets entiteter. Der er derfor ikke i rapporten medtaget noget stort klassediagram som viser alle klasserne. Dette vil blive alt for uoverskueligt og desuden have for mange gentagelser.

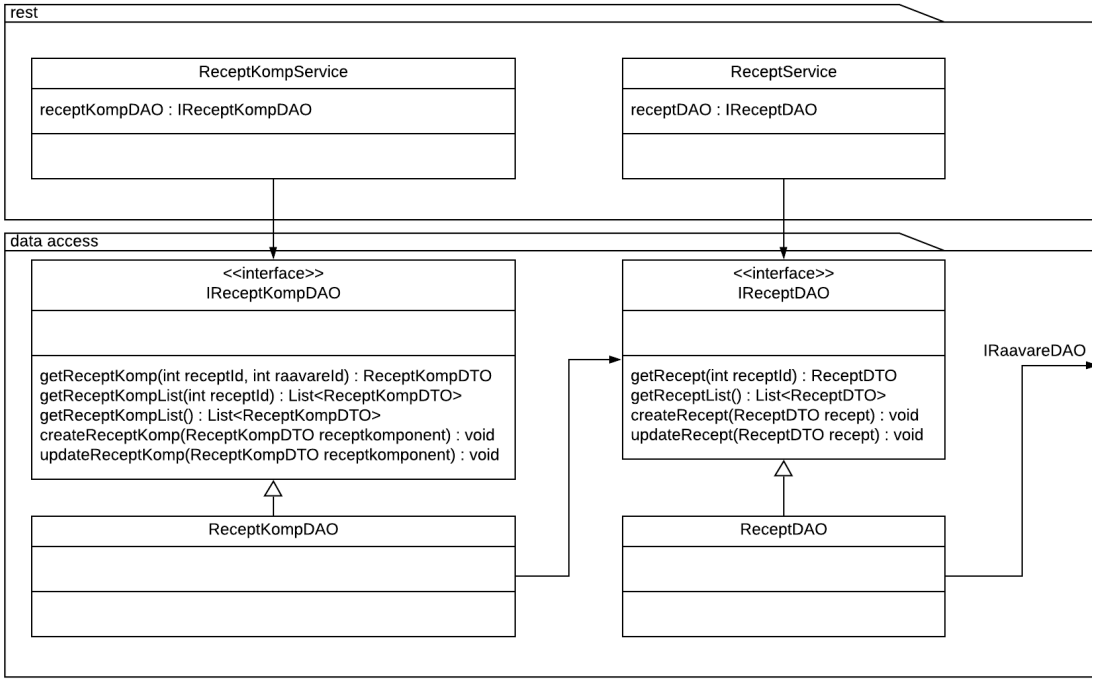
Hele backend-delens arkitektur vil til gengæld nu blive forsøgt beskrevet ved et lille udsnit at et sådant stort klassediagram. Udsnittet kan ses i figur 9.

I figur 9 tages der udgangspunkt i entiteten "recept". Klassen ReceptKompDAO er et Data Acess Objekt, hvis eneste formål er at tilgå databasen og hente, opdatere og oprette recepter deri.

Idet der er tale om en relationel database, mens ReceptDTO'en er lavet på en ikke-relationel måde (recepten indeholder alle dens komponenter), er ReceptDAO'en nødt til at benytte en ReceptKompDAO til leve op til sin opgave. Det er nemlig ReceptKompDAO'en der har ansvaret for at tilgå ReceptKomponenter i databasen. På helt tilsvarende måde er dette ikke muligt uden at ReceptKompDAO'en skal benytte en RaavareDAO, fordi ReceptKomponenterne indeholder råvarer.

Alle DAO'erne implementerer et interface, der beskriver de metoder DAO'en skal have som minimum. Det er lavet på den måde, da det vil være langt nemmere i fremtiden at udskifte hele databasen med en anden database. Hvis man f.eks. ønskede at overgå til en MongoDB-database som gemmer alle entiteter på en ikke-relationel måde, skulle der blot implementeres nye DAO'er mens alt alt andet i programmet kunne beholdes som det der.

[13]Forfatter: Ida



Figur 9: Udsnit af klassediagrammet for backend-delen af programmet

De klasser som benytter DAO'erne findes i rest-pakken og er alle services. Deres primære ansvar er derfor at tage imod klientens REST-kald og besvare disse. Hertil benytter de selvfølgelig DAO'erne gennem deres respektive interfaces. Det er lavet således, at der er netop 7 services helt svarende til de 7 DAO'er.

7.3.1 Data access laget[14]

Data-access laget indeholder alle de klasser hvis ansvar er at kommunikere med databasen. Designet er lavet sådan, at dette er disse klasser er de eneste, der har kontakt med databasen for at sikre at det sker på en ensformet og kontrolleret måde. Alle DAO-objekterne implementerer interfaces, så de let vil kunne udskiftes.

Konkret er alle DAO-objekternes ansvar at understøtte Create, Read og Update for deres tilsvarende entitet i databasen. Ingen af dem understøtter en Delete-funktion. Det er en designbeslutning, der er blevet truffet for hele programmet, at det ikke skal være muligt at slette fra databasen.

For det første skyldes dette, at det ikke er vigtigt for virksomheden at kunne slette noget af deres data. De vil gerne kunne deaktivere og aktivere brugere, men har frabedt at kunne slette dem. Ting som Recepter eller RåvareBatches er aktuelle i et stykke tid, men vil for altid være relevante for virksomheden at gemme i databasen ifm. almindelig arkivering.

For det andet skaber det en række udfordringer at tillade slettefunktioner i den relationelle database. Der skal nemlig tages stilling til, hvad der sker, hvis et refereret objekt slettes. Enten slettes også alle objekter som refererer til objektet. Alternativt sættes deres reference til "null". Ingen af mulighederne er specielt smarte for en medicinal-virksomhed, der gerne vil kunne backtrace alle steps i hvordan deres produkter er blevet lavet.

[14]Forfatter: Anthony

DAO'ernes metoder er alle temmelig selv-forklarende i deres navne, og vil ikke blive beskrevet her.

7.3.2 REST-laget[15]

Designet af klasserne i REST-laget er meget intuitivt. Deres metoder er ikke medtaget i figur 9, idet de næsten 1:1 tilsvarer de metoder som er i deres tilsvarende DAO'er. Metoderne skal blot blive kaldt af de korrekte HTTP-metoder, hhv. GET, POST osv. og benytte deres DAO til at svare på kaldet. Hvordan dette konkret sker, vil blive forklaret i implementeringsdelen.

Der er ingen relationer direkte imellem Service-klasserne idet de hver især blot kommunikerer med deres DAO.

[15] Forfatter: Peter

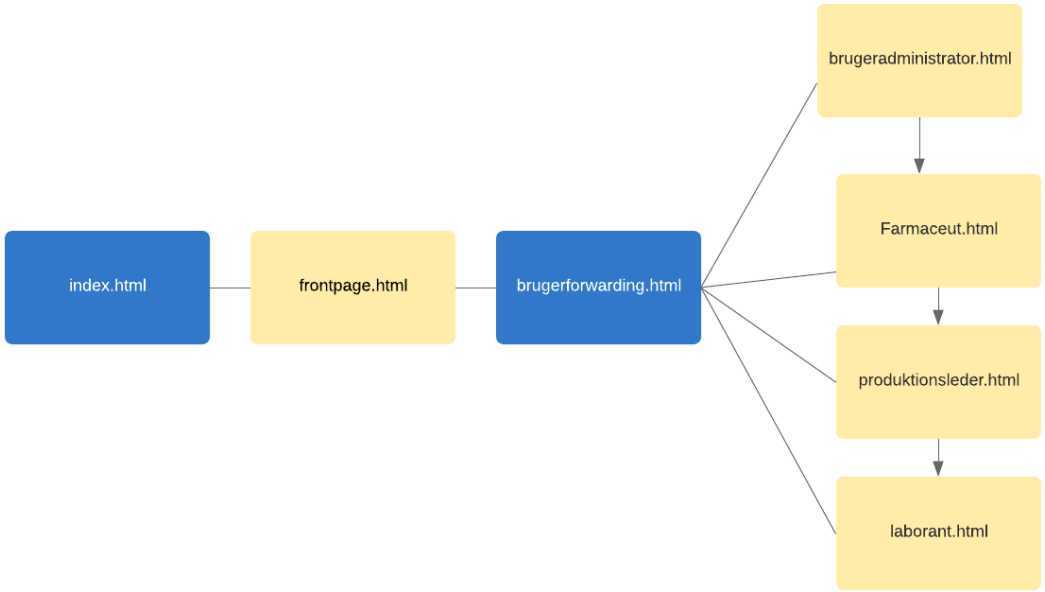
7.4 Webapplikationen[16]

I dette afsnit gennemgås ganske kort nogle design valg i forbindelse med front-end delen samt en diskussion af disse.

Frontend dækker over al HTML, CSS og javascript. Vi har valgt at lave en singlepage applikation, da det er nemmere at videreføre data på den enkelte bruger til de næste sider. Dette gør at en bruger kan logge ind på forsiden og stadig huskes til de efterfølgende undersider, uden brug af mellemliggende serverkald og funktionalitet.

Nogle ulemper der kan opstå ved at bruge singlepage er, at sessionen afbrydes hvis en bruger kommer til at genindlæse siden, og at URL'en altid er den samme. Den første ulempe anerkender vi som en mulig irritation, mens den sidste ikke regnes for at være at stort problem, da vi ikke regner med at brugerne af siden skal sende links rundt til hinanden. Dog kan det også påvirke den generelle brugervenlighed, da mange brugere har lært at 'holde øje' med URL'en løbende i besøget af en hjemmeside og anvende den som en slags 'bread crumbs' der viser, hvor på siden personen nu er. Vi har afgjort at fordelene ved en singlepage opvejer ulemperne, men ulemperne ved en lavere brugervenlighed, som følge af den statiske URL skulle overvejes ved videreudvikling.

Vi har designet vores webside så den ser anderledes ud afhængig af hvilken rolle man er logget ind som. Dette er fordi rollerne har forskellige rettigheder. På figur 10 ses et overblik over sidestrukturer og brugeradgang. De nedadgående pile indikerer, at reglerne for visning og adgang er differentieret imellem roller, hvor eksempelvis 'brugeradministrator' har adgang til alle 4 undersider, imens 'laborant' kun har adgang til 'laborant.html'. På figuren er de sider, brugeren ikke ser, markeret med blåt, og dem man ser er gule.



Figur 10: Oversigt over brugeradgang på hjemmesiden

[16]Forfatter: Ida

8 Implementering: Databasen[17]

Databasen til vores system er en MySQL-database, som kører gratis på en server gennem Amazon. Oprettelse af database er sket ved at køre en MySQL-fil, som indeholder både tabellerne og deres informationstyper, efterfulgt af endnu en MySQL-fil der indeholder test data (se bilag).

Et eksempel på hvordan tabellerne er oprettet, kan ses nedenfor. Tabellen her opretter de forskellige attributter som en bruger skal have, hvilket inkludere attributternes navne og datatyper.

```
create table Brugere
(brugerId decimal(3,0),
brugerNavn Varchar(20),
ini Varchar(4),
cpr char(10),
rolle ENUM('Administrator','Farmaceut','Produktionsleder','Laborant'),
aktiv TINYINT(1),
Primary key(brugerId)
);
```

Datatyperne i tabellerne vælges med omtanke, da det sætter en begrænsning på for input type og længde. I databasen er der nøje udvalgt datatyper som sørger for at indholdet i databasen overholder kravene til dens elementer.

Nedenfor her, ses et eksempel på den test-data, som tilføjes til databasen.

```
insert into Brugere Values('1','Morten Jeppesen','MJ',
                           '1210854535','Administrator',1);
insert into Brugere Values('2','Julie S rensen','JS',
                           '0209903217','Farmaceut',1);
insert into Brugere values('3', 'Annie Hansen', 'AH',
                           '2807751213', 'Produktionsleder',1) ;
insert into Brugere values('4', 'Karl Max', 'KM',
                           '2302987832', 'Laborant',1) ;
```

Her kan man også se hvordan databasens datatyper sørger for at inputtet er korrekt. ID'et skal kun kunne være tal, brugernavn og initialer skal være af variabel længde, cpr-nummeret skal være præcis 10 tal og man kan ikke have en rolle, som ikke er blandt de 4 specificerede roller i tabellen.

SQL-scriptet med tabeldefinitioner og testdata kan findes i sin fulde længde i bilagene.

9 Implementering: Backend

Det følgende afsnit vil gennemgå hvordan implementeringen af Backend-delen i Java er lavet. Afsnittet er opdelt efter en kategorisering af java-klasserne, der er i systemet, idet disse oplagt kan opdeles i meget ensartede grupper.

9.1 DTO[18]

DTO'ernes implementering er meget enkel. Alle DTO'erne i systemet indeholder udelukkende de autogenererede getters og setters samt 2 autogenererede konstruktører, hhv. den "tomme"konstruktør og en almindelig konstruktør som initialiserer alle attributter. De autogenererede getters og setters samt den tomme konstruktør er nødvendige for seialiseringen af objekterne til JSON-formatet.

Et udsnit af BrugerDTO ses nedenfor som eksempel. Der er naturligvis truffet nogle valg ift. datatyperne og navnegivning af attributter. Det er feks. blevet valgt, at et CPR-nummer skal være en string, da foranstående 0'er ignoreres i tal-datatyper som int. Desuden er attributten "akitiv"implementeret som en int for lettet konvertering til SQL, da SQL alligevel betragter en boolean som en TINYINT(1), altså en int af længden 1 bit.

[17] Forfatter: Marie
[18] Forfatter: Peter

```
1  public class BrugerDTO {
2      private int brugerID;
3      private String rolle;
4      private String brugerNavn;
5      private String initialer;
6      private String CPR;
7      private int aktiv; //0 means false, 1 means true
8
9      public BrugerDTO() {
10     }
11
12     public BrugerDTO(int brugerID, String rolle,
13         String brugerNavn, String initialer, String CPR, int aktiv) {
14         ...
15     }
16 }
```

BrugerDTO'en er simpel, idet den ikke indeholder nogen andre DTO'er, men så at sige er "uafhængig". I modsætning hertil betragtes nu ProduktBatchDTO nedenfor. Den indeholder ikke bare en, men en liste af ProduktBatchKompDTO'er.

Som nævnt i desgin-afsnittet er dette også den eneste DTO' der har en relation, der ikke er løst ved at gemme en instans af det referede objekt. ProduktBatchDTO indeholder nemlig en int receptId, som refererer til en recept.

```
1  public class ProduktBatchDTO {
2  int pbId;
3  int receptId;
4  int status;
5  List<ProduktBatchKompDTO> produktBatchKomponenter;
6
7      public ProduktBatchDTO() {
8      }
9
10     public ProduktBatchDTO(int pbId, int receptId, int status,
11         ArrayList<ProduktBatchKompDTO> produktBatchKomponenter) {
12        ...
13     }
```

9.2 Interfaces[19]

Som nævnt i designafsnittet, implementerer alle DAO'erne i systemet et interface, der gør at det vil være nemt at udskifte DAO'en, hvis f.eks. der blev skiftet til en ny type database. Et eksempel på et sådant interface ses nedenfor. Interfaces indeholder ingen logik, men blot navne, returtyper og argumenter til metoder, som DAO'erne skal implementere.

```
1  public interface iBrugerDAO {
2      BrugerDTO getBruger(int oprId) throws DALException;
3      List<BrugerDTO> getBrugerList() throws DALException;
4      void createBruger(BrugerDTO opr) throws DALException;
5      void updateBruger(BrugerDTO opr) throws DALException;
6  }
```

9.3 DAO[20]

Et eksempel på en DAO-klasse vil nu blive vist og diskuteret. Der vil blive tage udgangspunkt i getRaavarebatch() -metoden i RaavareBatchDAO, som ses nedenfor. Det ses på linje 1, at RaavareBatchDAO naturligvis implementerer sit interface, og desuden at den gennem interfacet iRaavareDAO benytter en RaavareDAO til at hente den RaavareDTO som svarer til en bestemt RaavareBatchDTO.

Kort sagt gør metoden getRaavareBatch() det, at den benytter et objekt af typen DBconnector,

[19] Forfatter: Peter
[20] Forfatter: Peter

på linje 7, til at oprette forbindelse til databasen og dernæst eksekverer den et SQL SELECT-statement, linje 12, hvor der søges efter en entitet med det rbId som er givet som argument til metoden.

DBconnectoren returnerer et Resultset, hvoraf dataene kan udtrækkes og gemmes i et ny-oprettet DTO-objekt af typen RaavareBatchDTO, linje 14 til 17. Det er det objekt, der skal returneres. For at kunne udfylde alle RaavareBatchDTO'ens felter, skal metoden som sagt også benytte RaavareDAO'en til at hente den tilsvarende RaavareDTO. Det sker ved, at RaavareDAO'en har en metode, getRaavare(), som helt på samme måde som getRaavareBatch() returnerer en bestemt RaavareDTO ud fra et ID.

Undervejs kan der meget naturligt opstå SQL-exceptions. Disse exceptions fanges, og der kastes en DALException. Exceptionhåndteringen vil blive forklaret videre i et underafsnit for sig selv. SQL-fejlbeskeden printes så til java-konsollen, og den tekniske fejlbesked sendes med til de øvre lag.

```
1  public class RaavareBatchDAO implements iRaavareBatchDAO{
2      private iRaavareDAO raavareDAO = new RaavareDAO();
3
4
5      @Override
6      public RaavareBatchDTO getRaavareBatch(int rbId) throws DALException {
7          DBconnector dBconnector = new DBconnector();
8          RaavareBatchDTO raavareBatchDTO = new RaavareBatchDTO();
9
10         try{
11             Statement statement = dBconnector.connection.createStatement();
12             ResultSet resultSet = statement.executeQuery("SELECT * FROM
       RaavareBatches WHERE rbId = "+rbId+";");
13             resultSet.next();
14             raavareBatchDTO.setRbId(resultSet.getInt(1));
15             raavareBatchDTO.setMaengde(resultSet.getDouble(2));
16             int raavareId = resultSet.getInt(3);
17             raavareBatchDTO.setRaavare(raavareDAO.getRaavare(raavareId));
18
19         }catch (SQLException e){
20             e.printStackTrace();
21             throw new DALException("Kunne ikke opdatere den oenskede
       RaavareBatch",e.getMessage());
22         }
23
24         dBconnector.closeConnection();
25         return raavareBatchDTO;
26     }
27
28 ...
29 }
```

DBconnector

Da SQL ikke er en del af pensum i dette kursus, vil beskrivelsen af DBconnectoren være meget kortfattet.

DBconnectoren er en temmelig simpel klasse, der udelukkende benyttes til at oprette og lukke forbindelsen til gruppens SQL-server. Den indeholder mest af alt en del tekniske detaljer, som skal bruges til at identificere serveren og oprette forbindelse på en stabil måde. Dens metoder kan selvølgelig også kaste DALExceptions, hvis f.eks. SQL-serveren er nede.

```
1  public class DBconnector {
2      Connection connection;
3      String host = "medicinaldb.cxhgltlvl9j3.eu-central-1.rds.amazonaws.com";
4      String port = "3306";
5      String username = "admin";
6      String password = "HurraForMig";
7      String driver = "com.mysql.cj.jdbc.Driver";
8      String url = "jdbc:mysql://"
9              + host + ":" + port + "/MedicinalDb" + "?characterEncoding=latin1"
```

```
10              + "&useJDBCCompliantTimezoneShift=true" +
11              "&useLegacyDatetimeCode=false&serverTimezone=UTC";
12
13      public DBconnector() throws DALException{
14          try {
15              Class.forName(driver);
16              connection = DriverManager.getConnection(url, username, password);
17          }catch (Exception e){
18              throw new DALException("Fejl med at oprette forbindelse til SQL
    server",e.getMessage());
19          }
20      }
21
22      public void closeConnection() throws DALException {
23          try {
24              connection.close();
25          }catch (SQLException e){
26              e.printStackTrace();
27              throw new DALException("Fejl med at lukke forbindelse til SQL
    server",e.getMessage());
28          }
29      }
30  }
```

9.4 REST[21]

Til at udveksle information mellem vores frontend og backend benytter vi os af vores RESTful API. Her benytter vi overordnet os af HTTP-metoderne GET, POST og PUT. Hver af vores data access objekter har en tilhørende REST service. Disse services har ansvaret for at kalde de rette metoder, for at sende og returnere de korrekte variable, samt for hvilken path som frontend'en kan tilgå dem på. De er derfor blot en form for mellemmand som bestemmer hvilke muligheder man har når man benytter sig af vores API Når man benytter sig af REST-kaldene kræver det den korrekte sti(Path). Alle vores kald starter i stien "/rest". Hver af de enkelte services vil have deres egen sti, f.eks "/rest/recept".

I implementationsafsnittet om Brugeradministration beskrives hvordan BrugerService bliver benyttet i vores frontend.

Vi tager udgangspunkt i vores BrugerService klasse. Som man kan se på linje 1 i nedestående upluk fra vores kode, er stien til BrugerService "bruger". Det vil sige at for at tilgå disse metoder skal stien være "/rest/bruger". I BrugerService benytter vi os af GET, POST og PUT, som hver især har forskellige formål og funktionaliteter.

```
1   @Path("bruger")
2   @Consumes(MediaType.APPLICATION_JSON)
3   @Produces(MediaType.APPLICATION_JSON)
4
5   public class BrugerService {
6       iBrugerDAO brugerDAO = new BrugerDAO();
7
8       @GET
9       public List<BrugerDTO> getBrugerList() throws DALException {
10          return brugerDAO.getBrugerList();
11      }
12
13      @GET
14      @Path("{id}")
15      public BrugerDTO getBruger(@PathParam("id") int id) throws DALException {
16          return brugerDAO.getBruger(id);
17      }
18
19      @POST
20      @Produces(MediaType.TEXT_PLAIN)
21      @Consumes(MediaType.APPLICATION_JSON)
```

[21] Forfatter: Christian

```
22      public Response createBrugerJson(BrugerDTO brugerDTO) throws DALException {
23          brugerDAO.createBruger(brugerDTO);
24          return Response.ok("Tilf jet").build();
25      }
26
27      @PUT
28      @Produces(MediaType.TEXT_PLAIN)
29      @Consumes(MediaType.APPLICATION_JSON)
30      public Response updateBrugerJSON(BrugerDTO brugerDTO) throws DALException {
31          brugerDAO.updateBruger(brugerDTO);
32          return Response.ok("Opdateret").build();
33      }
34  }
```

Det første GET kald vi har kalder metoden getBrugerList() på objektet brugerDAO og returnerer et array med brugere til applikationen som lavede kaldet.

Vi har også en mere specifik GET kald hvor vi også benytter os af parameter som vi får via stien (@PathParam). Måden det fungerer på er, at når man laver GET kaldet så tilføjer man en værdi i stien, som så vil blive givet med. Dette kunne f.eks se sådan her ud "/rest/bruger/3"På den måde vil tallet 3 blive givet med. I dette tilfælde benytter vi tallet til at returnere en bestemt bruger, med det brugerID matchene til PathParam.

I BrugerService benytter vi os af POST til at oprette brugere. Som man kan se i linje 20 og 21 har vi defineret hvilken type data POST kaldet benytter sig af, samt hvilken type den producerer. I dette tilfælde benytter vi os af et JSON objekt, og derudover vil den returnere plain tekst til frontenden. POST kaldet kalder en metode som hedder createBruger() på objektet brugerDAO, hvor det er nødvendigt at sende en brugerDTO med. Derfor er det nødvendigt at JSON objetktet er på samme form som vores brugerDTO'er. Hvis det lykkes at oprette en bruger vil den returnere teksten "Tilfoejet".

Til sidst er det også muligt at opdatere brugerne ved hjælp af et PUT. PUT kaldet minder rigtig meget om POST kaldet og kræver den samme type data, men den vil i stedet opdatere en eksisterende bruger.

9.5 Exceptionhåndtering[22]

DALException er en Javaklasse, der bruges som fællesbetegnelse i programmet til alle Exceptions, som kan opstå i Data-access-laget. Nedenfor ses et eksempel på, hvordan den kastes.
SQL-exceptions vil opstå i DAO-klasserne af mange forskellige årsager. Systemet vil i alle tilfælde kaste en DALException og vedhæfte SQLExceptionen's fejlbesked til DALException'en

```
1  try {
2          //SQL Statement execution
3          ...
4
5           }catch (SQLException e){
6               e.printStackTrace();
7               throw new DALException("Kunne ikke finde bruger med det
       ID",e.getMessage());
8           }
```

Nedenfor ses DALException-klassen. Denne nedarver fra Exception, og har en ekstra attribut, technicalMSG, som foruden den almindelige fejlbesked videresendes til front-end-udviklere, der kan få indsigt i hvorfor der blev kastet en exception i Data Access -laget.

```
1  public class DALException extends Exception {
2      public String technicalMSG;
3
4      public DALException(String userMSG, String technicalMSG) {
5          super(userMSG);
6          this.technicalMSG = userMSG + "\nDatabase says: " + technicalMSG;
7      }
```

[22] Forfatter: Peter

```
8
9       public String getTechnicalMSG() {
10          return technicalMSG;
11      }
12  }
```

DALExceptions vil blive kastet videre til rest-laget, hvor Service-klasserne kaster dem videre til en ExceptionMapper, DALExceptionMapper. Denne ses i kodeeksemplet nedenfor. DALExceptionMapperen benytter en lokal klasse, ErrorMSG til at videresende både den brugervenlige fejlbesked og den mere tekniske fejlbesked til front-end-laget som et JSON-objekt. På den måde, er det muligt i frontend-udviklingen at tilgå disse to fejlbeskeder seperat.

```
1   @Provider
2   public class DALExceptionMapper implements ExceptionMapper<DALException> {
3
4       public static class ErrorMsg{
5           String userMSG;
6           String technicalMSG;
7           //Autogenerated constructor, getters and setters
8           ...
9       }
10
11      @Override
12      public Response toResponse(DALException exception) {
13          ErrorMsg errorMsg = new ErrorMsg();
14          errorMsg.userMSG =exception.getMessage();
15          errorMsg.technicalMSG = exception.getTechnicalMSG();
16          return Response
17                  .status(Response.Status.NOT_FOUND) //404
18                  .entity(errorMsg)
19                  .type(MediaType.APPLICATION_JSON)
20                  .build();
21      }
22  }
```

Havde der været mere tid til at udvikle projektet, havde det stået meget højt på prioriteringslisten at gøre dette system bedre. Generelt kan det siges, at det er ærgerligt kun at bruge en Exception-klasse. Da der kun er en Exception-klasse, er der også kun en ExceptionMapper, hvorfor alle exceptions udløser den samme fejlkode, 404, der i dette projekt (lidt mod god konvention) bruges som en generisk fejlkode. Et langt bedre system havde naturligvis været at lave mange forskellige Exceptions og dermed mange ExceptionMappers's, der kunne sende forskellige fejlkoder, som var sigende for den bestemte fejl.

10 Implementering: Front end

Dette implementeringsafsnit om front-end-delen af programmet, vil blive opdelt i 2. Første del vil handlde JavaScipt og HTML, der beskriver hvordan web-applikationens logik og elementerne på hjemmesiden fungerer. Anden del vil handle om CSS, der beskriver hvordan den er stylet.

10.1 Javascript og HTML[23]

I de følgende underafsnit, vil den del af implementationen af front-end-delen som er skrevet i Javascript og HTML, blive gennemgået. I intelliJ-projektet, er ligger denne implementation i mappen "webapp", hvori, der bl.a. ligger 4 undermapper - en til hver brugertype. Disse indeholder de HTML og JavaScript-dokumenter, som hører til den bestemte brugertype.

På trods af at alle fire brugertyper kan noget forskelligt, er der flere "design-mønstre"i front-end-delen som går igen. Eksempelvis er brugeradministrationen funktionelt meget det samme som råvareadministrationen. I de følgende underafsnit, vil det derfor primært blive prioriteret at beskrive front-end-delens forskellige funktionaliteter én gang, mens "gentagelser"vil blive sprunget over.

[23]Forfatter: Peter

10.1.1 Login[24]

For at benytte vores system er man nød til at have en bruger med et dertilhørende brugerID. Brugeren skal kende sit brugerID på forhånd for at kunne logge ind på systemet. Programmet har ikke skulle håndtere sikkerhed så derfor er et kodeord ikke nødvendigt.

Brugeren bliver mødt af en login boks hvor han skal indtaste sit bruger ID og derefter trykke på login. Der bliver herefter lavet et GET kald til vores API som enten returnerer en bruger som JSON objekt eller returnerer en fejlmeddelelse, som f.eks kunne være i tilfælde af at man indtaster et bruger ID, på en bruger som ikke findes.

I vores program findes der 4 forskellige brugertyper, brugeradministrator, farmaceut, produktionsleder og laborant. Laboranten har som den eneste kun adgang til 1 funktionalitet, afvejning, hvor resten af dem har adgang til flere afhænigt af deres rolle. Derfor vil brugeradministrator og farmaceut og produktionslederen bliver videresendt til en side, hvor de her kan vælge hvilken funktionalitet de ønsker at benytte sig af, hvorimod laborant vil blive direkte videresendt til afvejningen da han ikke har andre muligheder. I vores "script.js"har vi en metode der hedder "checkLogin()"som håndterer det meste af vores login funktionalitet. Denne er vedhæftet i en forsimplet version længere nede.

Programmet håndterer det ved at den læser brugertypen i JSON objektet og herefter bliver der kørt en switch case med de dertilhørende matchende 4 brugertyper. Når switch casen matcher gemmes brugertypen i en variabel, som senere skal bruges. Brugeren vil nu få vist hvilke valgmuligheder den har. Når den nye side med valgmuligheder bliver indlæst, bliver der kørt et for loop som benytter sig af den tidligere gemte brugertype til at fjerne de muligheder som brugeren ikke har rettigheder til.

```
1  function checkLogin() {
2      var id =document.getElementById("loginBrugerID").value;
3      var errorMessage;
4      errorMessage = document.getElementById("errorMessage");
5      errorMessage.innerHTML="";
6      $.ajax({
7          method: 'GET',
8          url:'rest/bruger/'+id,
9          success: function (data) {
10             //set global variable "data" for later use
11             user = data;
12             brugerID=data.brugerID;
13             brugerNavn=data.brugerNavn;
14             if(data.aktiv==1){
15             switch (data.rolle) {...}
16             }
17             else if(data.aktiv==0){
18                 errorMessage.innerHTML = "Denne bruger er inaktiv og du kan
       derfor ikke logge ind med denne"
19             }
20         },
21         error: function (   jqXHR) {
22             errorMessage.innerHTML= jqXHR.responseText;
23         }
24     })
25 }
```

;

I systemet kan en bruger blive sat som inaktiv og vil dermed ikke længere have mulighed for at logge ind. I tilfælde af at en bruger er sat som inaktiv vil han få en fejlbesked om at han ikke kan logge ind grundet at statussen er inaktiv. Som man kan se i linje 14 i vores checkLogin, starter vi med at checke om en bruger er aktiv inden brugere bliver videresendt. Hvis brugeren derimod ikke er aktiv vil linje 17+18 blive eksekveret og brugeren vil få en fejlmedelse.

[24]Forfatter: Christian

10.1.2 Brugeradministration[25]

I dette afsnit gennemgås tre simple kald til API'en: GET, PUT og POST med udgangspunkt i html siden brugeradministration. Flere af de andre undersider benytter lignende kald, og i det kommende afsnit refereres derfor hertil.

Brugeradministrationsdelen er implementeret med en enkelt html side og en dertilhørende javascript. Html siden indeholder to formler: createBruger og updateBruger og en tabel over nuværende brugere. Brugerinputtet valideres både gennem html og javascript. I html gøres det eksempelvis ved at lave en dropdown menu når man skal vælge rolle. I javascript er der flere metoder dedikeret, bl.a. validateBrugerInput som sørger for at input har rigtige længder og typer med passende fejlmeddelelser.

GET
Det første der sker på brugeradministration er et kald til denne metode. Kaldet gentages også når man har oprettet eller redigeret en bruger. I funktionen (se figur nedenunder), benyttes et simpelt GET-kald til API'en. Den kalder på stien rest/bruger, som vil returnere alle brugere. For at generere tabellen benyttes et for each loop, som tilføjer rækker til tabellen med hjælp fra funktionen generateBrugerTable.

```
function loadBruger() {
    $.get('rest/bruger', function (data, textStatus, req) {
        $("#brugertable").empty();
        $.each(data, function (i, elt) {
            $('#brugertable').append(generateBrugerTable(elt));
        });
    });
}

function generateBrugerTable(bruger){
    return '<tr><td>' + bruger.brugerID + '</td>' +
        '<td>' + bruger.rolle + '</td>' +
        '<td>' + bruger.brugerNavn + '</td>' +
        '<td>' + bruger.initialer + '</td>' +
        '<td>' + bruger.cpr + '</td>' +
        '<td>' + bruger.aktiv + '</td>'
}
```

POST
Nedenfor ses et eksempel på et POST kald til API'en. Først gemmes de input brugeren har tastet ind i variable for at de kan bruges i kaldet på validateBrugerInput. Hvis den returnerer true betyder det at alle data er valideret. Vi tager dataen fra brugerformlen og serialiserer dem til et JSON objekt. Herefter foretages et POST-kald og brugeren oplyses i en alert om det lykkedes eller ej. Hvis det lykkedes kaldes loadBruger() funktionen, så brugertabellen automatisk opdateres.

```
function createBruger() {
    var brugerID = document.getElementById("opretBrugerID").value;
    var brugerNavn = document.getElementById("opretBrugerNavn").value;
    var ini = document.getElementById("opretInitialer").value;
    var CPR = document.getElementById("opretCPR").value;

    if (!validateBrugerInput(brugerID,brugerNavn,ini,CPR)){
        return;
    }
    event.preventDefault();
        var data =$('#brugerform').serializeJSON();
        $.ajax({
            url: 'rest/bruger',
            method: 'POST',
            contentType: "application/json",
            data: data,
            success: function (data) {
                alert(JSON.stringify(data));
```

[25] Forfatter: Ida

```
                loadBruger();
            },
            error: function (jqXHR) {
                alert(jqXHR.responseText);
            }
        })
}
```

PUT
Et PUT kald adskiller sig fra et post ved at gå ind og ændre i et JSON objekt, i stedet for at oprette et helt nyt. Her ses det hvordan dataen igen serialiseres til et JSON objekt og der laves et PUT kald. Hvis det lykkedes skrives ud i en alert til brugeren og loadBruger() kaldes, så tabellen opdateres.
Note: denne metode vises uden inputvalidering, da det allerede kan ses i createBruger.

```
function updateBruger() {
    event.preventDefault();
    var data =$('#brugerformUpdate').serializeJSON();
    $.ajax({
        url: 'rest/bruger',
        method: 'PUT',
        contentType: "application/json",
        data: data,
        success: function (data) {
            console.log("Det lykkedes")
            alert(JSON.stringify(data));
            loadBruger();
        },
        error: function (jqXHR) {
            alert(jqXHR.responseText);

        }

    })
}
```

10.1.3 Farmaceut[26]

Farmaceuten har 4 valgmuligheder efter vedkommende er logget ind, "Råvareadministration", "Opret recept", "Se indhold af en recept"og "Vis Recepter og Produktbatches".

Funktionaliteten af siden "Se indhold af en recept"er ikke videre interessant, og vil blive sprunget over her. Siden "Vis Recepter og Produktbatches"er lidt mere kompliceret, men er helt det samme som siden af samme navn under Produktionsleder. Denne vil derfor blive gennemgået i afsnittet om produktionslederens front-end-implementering.

Råvareadministration
Under Råvareaministration får man mulighed for at oprette og opdatere samt se en tabel over råvarerne. Når en råvare bliver oprettet eller redigeret, bliver den først tjekket for fejl. Der er flere forskellige steder, hvor inputtet bliver testet for fejl, både i HTML og Javascript-delen. Et eksempel på kode til fejlfinding i brugerens input er figur 11.

Råvareadministrationens funktionalitet er meget lig den tidligere gennemgåede brugeradministration, og den vil ikke blive beskrevet yderligere her.

Opret Recept
Når farmaceuten vælger at oprette en recept, vil vedkommende blive vist en side hvor de kan oprette recepten. Her kan de indtaste informationer om recepten, samt tilføje/fjerne linjer i recepten og se hvad den indeholder.

[26] Forfatter: Marie

```
function validateOpdateInputs() {
    //Input validation
    if (!document.getElementById('opretRaavareID').value) {
        alert("Venligst! Angiv et gyldigt R vare ID.")
        return
    } else if (!document.getElementById('opretRaavareNavn').value) {
        alert("Venligst! Skriv navnet p  det r vare du gerne vil oprette.")
    } else if (!document.getElementById('opretLeverandoer').value) {
        alert("Skriv venligst navnet p  leverand ren")
    } else {
        createRaavare()
        return;
    }
}
```

Figur 11: Oversigt over brugeradgang på hjemmesiden

Denne funktionalitet skiller sig lidt ud fra den sædvanlige opret-objekt-funktionalitet, der f.eks. er i brugeradministrationen, idet en recept har et variabelt antal receptkomponenter. Dette bliver håndteret ved, at javascriptfilen "farmaceutCreateRecept.js"har en global variabel raavareList, som benyttes til at styre drop-ned-menuen af råvarer, raavareOptionList.

I kodeeksemplet nedenfor ses hvordan funktionen loadRaavarer(), der kaldes når dokumentet indlæses, laver et GET-kald efter alle råvarer. Dette er lavet på den måde, da det giver farmaceuten mulighed for at vælge mellem de mulige råvarer i en overskuelig drop-ned-menu. Det har til gengæld den ulempe, at det kan tage lang tid at hente alle råvarer, og at drop-ned-listen kan blive for lang. Skulle man videreudvikle programmet, kunne man f.eks. have et søgefelt kombineret med en drop-ned-menu. Man kunne også havde undladt at indlæse alle råvarer, men blot dem som farmaceuten skriver ID på. Det havde givet en hurtigere indlæsning, men mindre brugervenlighed (fordi farmaceuten selv skal huske ID'erne).

Når råvarerne er indlæst, kaldes generateRaavareOptionList(), der genererer selve drop-ned-menuen som en option-list. Hver option har er en attribut "value", der sættes til indekset af råvaren i råvarelisten, og altså identificerer en råvare på listen unikt. Dette bruges, når råvarer skal tilføjes og fjernes fra option-listen.

```
function loadRaavarer() {
    //empty existing list
    raavareList = [*/*;

    //Load via GET-call to database
    $.get('rest/raavare', function (data, textStatus, req) {
        $("#raavaretable").empty();
        $("#raavare").empty();
        $.each(data, function (i, elt) {
            raavareList.push(elt);
        });
        generateRaavareOptionList();
    });
}

function generateRaavareOptionList() {
    $("#raavare").empty();
    raavareOptionList = document.getElementById('raavare');

    $.each(raavareList, function (i, elt) {
        var raavareOption = document.createElement("OPTION");
        raavareOption.setAttribute("value", i);
        var t = document.createTextNode(elt.raavareNavn + "; " + elt.raavareID);
        raavareOption.appendChild(t);
        raavareOptionList.appendChild(raavareOption);
    });
}
```

Nedenstående kodeeksempel er ud udpluk af funktionen addReceptKomp(), der tilføjer en valgt komponent til recepten. Selve opsætning af tabellen er her sprunget over. Sidst i funktionen ses det, hvordan råvareOptionListen opdateres ved at den valgte råvare fjernes derfra. Således sikres det, at farmaceuten ikke kan tilføje den samme råvare til recepten flere gange.
En lignende funktionalitet sørger for, at en råvare, der slettes fra recepten, kommer tilbage på råvareOptionListen.

```
function addReceptKomp() {
        ...
    var table = document.getElementById("receptkomptablebody");
    //Table setup
        ...

    //Remove the option from the raavareOptionList
    raavareOptionList = document.getElementById('raavare');
    const index = document.getElementById("raavare").value;
    $.each(raavareOptionList, function (i, elt) {
        if (elt.value == index){
            raavareOptionList.removeChild(elt);
        }
    });
}
```

10.1.4 Produktionsleder[27]

Produktionslederen har 3 muligheder ved login, hhv. "Råvarebatch administration", "Produktbatch administration"og "se recepter".

Råvarebatch-administrationen er en simpel side, hvor man kan oprette og se en tabel over råvarebatches. Funktionaliteten i denne er den samme som blev gennemgået i brugeradministrationen med oprettelse af brugere og visningen af tabellen af brugere.

Opret Produktbatch

Opret Produktbatch -siden er lidt anderledes end de almindelige sider for oprettelse. Det skyldes at produktbatches skal oprettes til en bestemt recept, og på baggrund af recepten skal en speciel forhåndsvisning, der skal kunne printes ud, genereres.

Inden en produktbatch kan oprettes, skal man derfor indtaste den ReceptID, som produktbatchen skal laves ud fra. På trods af, at blot ReceptID'et og ikke selve recepten gemmes i en ProduktBatchDTO, laves nu et GET-kald efter den pågældende recept, da det er nødvendigt for forhåndsvisningen (samt for at inputvalidere ReceptID'et). I kodeudsnittet nedenfor ses den funktion, der kaldes, når brugeren beder om forhåndsvisningen.

```
function showProduktBatchFromReceptID() {
    console.log("Kalder funktionen showProd...");
    var receptId = document.getElementById("pb_receptId").value;
    console.log("hentet receptId: "+ receptId);

    $.ajax({
        url: 'rest/recept/'+receptId,
        method: 'GET',
        success: function (data) {
            console.log(data);
            showPrintablePB(data);
            receptIdValid=true;
        },
        error: function (jqXHR) {
            alert(jqXHR.responseText);
            receptIdValid=false;
        }
    })
```

[27] Forfatter: Peter

```
19
20 }
```

Funktionen henter den ReceptID, som brugeren har angivet og bruger som sagt denne til at lave et GET-kald efter recepten. I successcenariet, hvor recepten findes, kaldes funktionen showPrintablePB() (hvor PB er ProduktBatch). showPrintablePB() laver blot et tabel-view af den hentede recept med nogle tomme felter, som en laborant skal kunne udfylde.

For at undgå problemer med asynkronitet, er det nødvendigt at kalde showPrintablePB() i succesScenariet. Havde man kaldt den i linje 19 i stedet, ville recepten sandsynligvis ikke være hentet endnu, når forhåndsvisningen skal genereres.

Når forhåndsvisningen er lavet på baggrund af en gyldig recept, samt når det resterende brugerinput er valideret, kan brugeren oprette ProduktBatchen ved at trykke på knappen "Opret". Produktionslederen har knappen "Print siden"til rådighed, som på ethvert tidspunkt kan printe den aktuelle side, der ses. Hensigten er at denne skal bruges, når en produktbacth er gemt.

Vis recepter og produktbatches
Denne side er helt magen til siden i Farmaceut-delen af samme navn. Det er lavet sådan at de begge har adgang til deres egen version af siden, så deres tilbage-knapper virker, samt fordi det er en essentiel del af deres arbejde for begge brugertyper. Idet farmaceuten også har adgang til produktionslederens side, kunne farmaceuten også bare bruge denne til at se recepter. Det ville imidlertid virke underligt, at skulle logge ind som en anden brugertype for at udføre sit arbejde.

Visningen af recepter og produktbatches er lavet som en samlet tabel, da det er essentielt for en produktionsleder ikke blot at vide hvilke produktbatches der er, men også hvilke recepter de er oprettet til samt deres status. Tabellen giver er glimrende overblik over netop dette.
For at undgå problemer med asynkronitet, er flowet af funktionernes eksekvering lidt speciel her. Når dokumentet åbnes, kaldes funktionen loadProduktBatches(). I dennes succes-scenarie kaldes loadRecepter(), som i sit succes-scenarie gennemgår alle de fundne recepter og genererer en tabelrække ud fra disse samt fra listen af Produktbatches.

Disse to load-funktioner henter altså alle recepter og alle produktbatches i databsen. Pga. DTO'ernes opbygning, hentes faktisk al data i hele databasen på den måde, og siden kan derfor godt tage lidt tid om at loade.

10.1.5 Laborant[28]

En laborant har til opgave at afveje de forskellige råvare der skal bruges til en recept. Virksomheden har givet os nogle forretningsregler(bilag 4 i opgave beskrivelsen) og disse har været udgangspunktet i hvordan vi har udviklet afvejningsfunktionen. Derfor er det en forudsætning for at kunne lave en afvejning at man på forhånd kender et produktbatch ID. Forretningsreglerne tolkes sådan, at laboranten har fået udleveret dette på et stykke papir.

Til laborantens afvejningsproduceduere hører javascriptfilen "laborant.js", som er forholdsvist lang. Dens vigtigste to elementer vil nu blive gennemgået.

getProduktBatch()

Når der trykkes på knappen "Vis produktbatch", kaldes funktionen getProduktBatch(). Denne foretager inputvalidering af det angivne ProduktBatchID og laver dernæst en GET-kald efter ProduktBatchen. I GET-kaldets successcenarie, kaldes funktionen getRecept(), der henter produktbatchens recept. Det er vigtigt, at getRecept() kaldes netop der, for at undgå problemer med de asynkrone kald. Når recepten og produktbatchen er hentet, opsættes tabelfremvisningen af produktbatchen og informationen om dens recept.

[28] Forfatter: Christian

validateAfvejningInput()

Når en laborant har lavet en afvejning og vil gemme denne, trykkes på knappen "Gem afvejning". Denne kalder funktionen validateAfvejningInput(), som igangsætter en del logik. Et udsnit af validateafvejningInput() kan ses nedenfor. De trivielle dele af inputvalideringen i starten er sprunget over (markeret med ...).

Når råvarebatch-ID'et skal valideres, sker det af flere omgang. Først startes et GET-kald efter den angivne råvarebatch.

```
function   validateAfvejningInput() {
  //Check that the raavare is not already afvejet
      ...
  //Validation: tara
      ...
  //Validation: netto
      ...
  //validation: raavarebatchID
  var rbId = document.getElementById("rbId").value;
  $.ajax({
      method: 'GET',
      url:'rest/raavarebatch/'+rbId+'/',
      success: function (data) {
          console.log("Get-kaldet var en succes");
          //Check that it contains the correct raavare and continue
          if (data.raavare.raavareID === currentReceptKomp.raavare.raavareID){
              currentRaavareBatch = data;
              //If the batch's maengde is too small
              if (currentRaavareBatch.maengde< netto){
                  alert("Den angivne nettovaegt er stoerre end den resterende
  maengde af den angivne ravarebatch.\n" +
                      "Der maa vaere sket en fejl.");
                  return;
              }
              //Update the used raavarebatch - now the maengde is smaller
              console.log("currentRaavareBatch:" + currentRaavareBatch.rbId);
              console.log("Netto:" + netto);
              updateRvbMaengde(currentRaavareBatch,netto);

              //Succes - raavareBatchID matches raavare of receptKomp
              saveAfvejningToDatabase();
          } else{
              alert("Den angivne raavarebatch ID findes i databasen, men svarer
  ikke til den raavare, du er ved at afveje.\n Tjek Raavarebatch ID't.")
          }
      },
      error: function () {
          console.log("Get-kaldet var en fiasko");
          alert("Der kunne ikke findes en raavarebatch i systemet med det
  angivne ID.")
      }
  })
}
```

saveAfvejningToDatabase

Når afvejning er blevet valideret kaldes metoden saveAvejningToDatabase() som er ansvarlig for at få sendt den pågældende produkt batch komponent til vores API. Hvis POST kaldet er succesfuldt vil metoden saveToDBWasSuccesful() blive kaldt som sørger for brugerens view bliver opdateret, så der nu vil stå, hvis en afvejning allerede er fortaget af en råvare.

finishAfvejning

Når alle råvare i en recept er blevet afvejet, og produktbatchen dermed er komplet, vil laboranten kunne trykke på "Afslut Produktbatch"og metoden finishAfvejning() vil blive kaldt.

```
function finishAfvejning() {
```

```
2       if (!(finishedRaaIDs.length ===recept.receptKomponenter.length)){
3           alert("Der er stadig raavarer som mangler at blive afvejet i denne
        produktbatch.");
4           return;
5       }
6       if (produktBatch.status==1){
7           updatePbStatus(2);
8       } else if (produktBatch.status==2){
9           alert("Produktbatchen er allerede afsluttet.");
10      } else {
11          alert("Fejl. Produktbatchens nuvaerende status er ikke \"Under
        produktion\" og det kan derfor ikke afsluttes");
12          return;
13      }
14  }
```

Linje 2 i finishAfvejning tjekker om alle komponenter til produktbathcen nu også faktisk er afvejet. Er de ikke det vil brugeren få en fejlbesked. I linje 7 opdatere vi produktbatch statussen til 2 hvilket vil betyder at produktbatchen nu er afsluttet. Der bliver også tjekket for om produktbatchen enten allerde er afsluttet eller om den ikke er sat til at være under produktion endnu.

10.2 Styling med CSS[29]

Al styling er samlet i en enkelt CSS-fil (main.css), og følger en flad struktur, hvor de enkelte elementer har en tilhørerende 'class'. Nogle enkelte HTML elementer er stylet globalt, og ved en fremtidig skalering af kodebasen, vil vi gå fra globalt stylede HTML-elementer og udelukkende benytte classes, for at simplificere konventionen.

Navnekonventionen, i denne prototype, følger ikke nogen moderne navnekonventioner, men med det relativt lave antal individuelle sider, har det ikke været et konkret behov. Udfordringen med det nuværende fravær af en konsekvent navnekonvention for CSS'en er, at der forekommer dubletter af egenskaber (properties) samt inkonsekvent layout, hvor visse elementer ikke følger samme design overalt på siden. De udfordringer og layout problemer, vil stige i omfang nærmest eksponentielt med en generelt større kodebase, og derfor skal man helst benytte en reel navnekonvention etableret inden udviklingen begynder.

Som en overvejelse af hvordan det kunne gøres bedre fra start, anbefaler vi at man indledningsvist afgør hvilken navnekonvention og struktur al styling skal følge. Denne beslutning imellem udviklere håndholdes af 'code reviews' på tværs af holdet, og én udvikler (frontend) får delegeret et overordnet ansvar, om at vedligeholde al CSS og styling i henhold til en metodologi.

[29] Forfatter: Ida

11 Konfiguration

11.1 Maven[30]

Som et værktøj til at styre samt forstå projektet og dets opbygning har vi gjort brug af Maven. Maven anvender en Project Object Model også kaldet en POM, som er et xml-fil. Denne fil indeholder de samtlige afhængigheder samt de relevante biblioteker, der er essentielle til at kunne compile det færdigtudviklede softwareprogram. De biblioteker der anvendes i det nærværende softwareprojekt sørger maven for at blive inkluderet og opdateret med de rigtige versioner. Dette skyldes de bagvedliggende server der sørger for at vi hele tiden får de nyeste opdateringer online. Som et krav til dette projekt anvender vi java-versionen 1.8 til sourcekoden, ligesom vi også anvender version 1.8 for target Virtuel Machine som koden kompileres til. De andre softwaresystemer som vi anvender i det nærværende projekt er angivet vha. dependency xml.tags. For eksempel anvendes der af mySQL connector, jersey.hk2 og mange flere. Disse afhængigheder og biblioteker er at finde i POM.xml filen i mavenprojektet i IntelliJ.

11.2 Git[31]

Til versionskontrol samt registrereing af ændringer i softwarekildekoden i dette projekt har vi anvendt af Github. Dette muliggjorde parallelprogrammering og på den måde har vi kunne arbejde flere samtidig på kildekoden. Derudover har det været et brugbart værktøj til at opdage ændringer i kildekoden der har forårsaget problemer og på den måde forebygge dette ved at vende valgte kildekoder til en tidligere tilstand. Linket til gruppens git repository er vedhæftede nedenunder.

GitHub: `https://github.com/Software2020Hold21/21_CDIOFinal/`

11.3 Systemkrav[32]

I det følgende gives en oversigt over systemkrav, der skal være tilgængelige for at kunne kompilere softwareprogrammet.

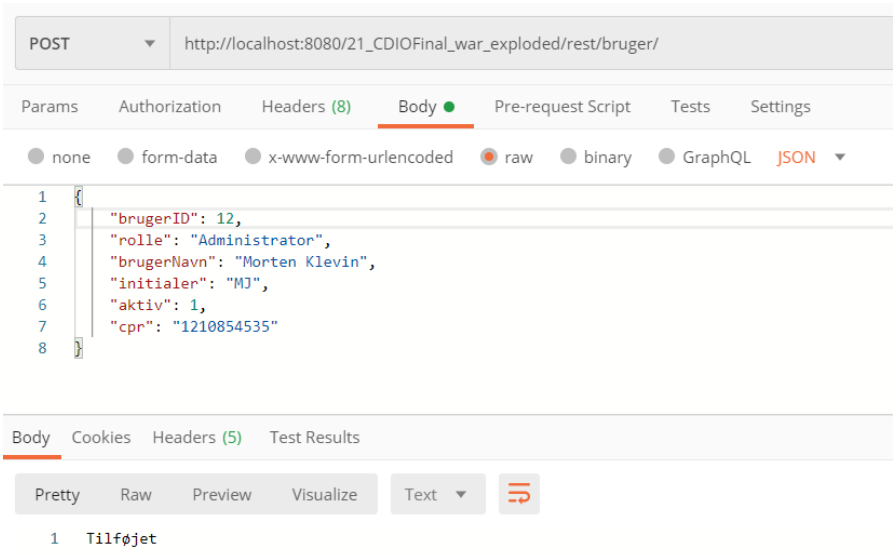
Opensource softwaren til udvikling af computersoftware - intelliJ IDEA Ultimate 2018.1 - skal være installeret i din enhed. Det er ligegyldigt hvilken version af intelliJ IDEA der skal være installeret, men det er dog et krav at Java 1.8 er hentet ind i enheden, før at man kan køre vores færdigtudviklede softwareprogram. Internetforbindelse kræves for at hente opdateringer og downloade samt gøre brug af visse funktioner. Downloade og installering af Apache Tomcat 8.5.11, kræver en internetforbindelse, ligesom det også kræves når man skal downloade intelliJ IDEA og ikke mindst køre vores hjemmeside.

Udover de krav, der er nødvendige for at kunne køre vores program, har især én funktion - print funktionen - yderligere krav. Der skal nemlig anvendes en printer for at kunne udprinte data fra hjemmesiden. Når man trykker på knappen "print siden"på en af html-siderne i hjemmesiden, kan man gemme den pågældende side som pdf fil. Først herefter kan man udprinte data vha. en printer. De fleste printere ude på DTU understøtter Wi-Fi Direct Printing, som gøre det muligt at udksrive trådløs. Men det kræver dog, at den enhed man vil udskrive fra har en Wi-Fi adapter, der understøtter Wi-Fi Direct Printing.

[30] Forfatter: Anthony
[31] Forfatter: Anthony
[32] Forfatter: Anthony

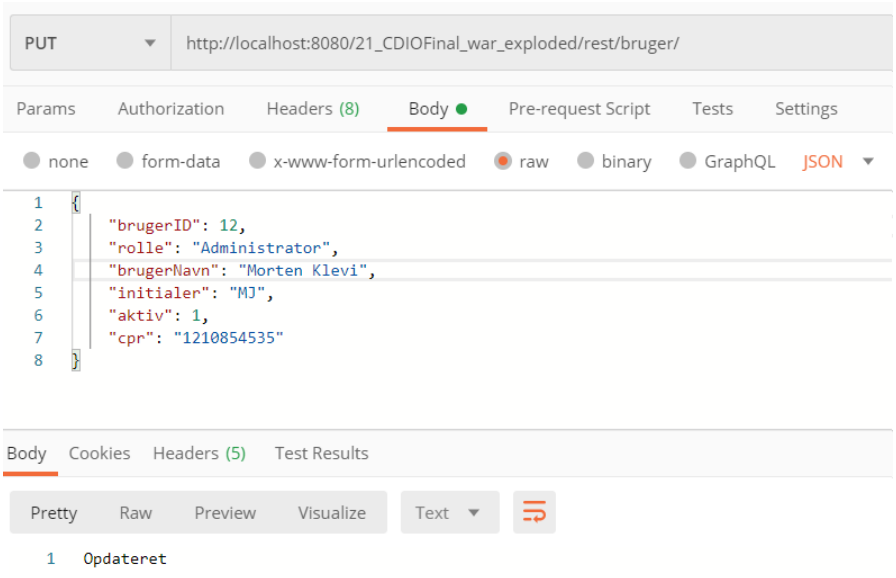
12 Test[33]

Vi har benyttet Postman, til at teste vores API-kald. Ved ethvert HTTP request vil vores server indledningsvist lave et GET-request, og ved succes, vil den returnere data formateret som JSON til klienten, som renderes og vises til brugeren. På figur 12 nedenfor, vises et eksempel på et vellykket POST-kald til API'en med stien 'rest/bruger', som har til formål at oprette en bruger.



Figur 12: POST-kald med postman

På figur 13 ses et tilsvarende put kald som bruges til at ændre i en bruger.



Figur 13: PUT-kald med postman

[33]Forfatter: Marie

12.1 JUnit-Test[34]

Vi har lavet i alt 4 Unit Tests, som ligger i mappen BrugerDAOTest, og vi har lavet én for hver API-kald:

- getBruger()
- getBrugerList()
- createBruger()
- updateBruger()

Vores Unit Tests efterprøver simple semi-randomiserede data cases, hvor eksempelvis brugerId randomiseres, imens andre værdier (eks. rolle: "laborant") forbliver statiske. Vores Unit Tests kunne med fordel bygges ind i vores build og deploy flow, så de bliver kørt og efterprøvet løbende og automatisk.

Unittests har ikke være prioriteret særligt højt i dette projekt, da API-kaldene løbende er blevet testet under udviklingen alligevel, og da programmet ikke har noget "business logic-lag.

12.2 Test af systemet[35]

Der er blevet lavet flere systemtests til at teste funkionaliteten af det færdige program. Et par af disse er blevet dokumenteret og vist i de følgende tabeller.

Testcase	TC1: Laborant log-in
Scenarie	Brugeren logger ind
Krav	K?: "Laboranten skal kunne indtaste sit ID, beskræfter det og bliver diregeret direkte til laborant siden"
Preconditions	Der eksistere en bruger-profil med laborant-rolle.
Postconditions	Laboranten er på laborant-siden.
Testprocedure	Laboranten indtaster og godkender deres identifikation. Derefter kan se den side hvor afvejning foregår.
Testresultat	Success
Testet af	Marie
Dato	23/06/2020
Testmiljø	Java SDK 1.8 221 IntelliJ IDEA 2020.1.2 (Ultimate Edition)

Testcase	TC2: Fejlede bruger log-in
Scenarie	Brugeren skal logge ind
Krav	
Preconditions	Hjemmesiden er indlæst
Postconditions	Brugeren har fået en fejlmeddelelse
Testprocedure	Taster et brugerID ind som ikke findes i databasen. Man får fejlmeddelelsen: "Kunne ikke finde bruger med det ID"
Testresultat	Success
Testet af	Ida
Dato	23/06/2020
Testmiljø	Java SDK 1.8 221 IntelliJ IDEA 2019.2.2 (Ultimate Edition) Build #IU-192.6603.28, built on September 6, 2019 Windows 10 Pro version 1909

[34]Forfatter: Ida
[35]Forfatter: Ida

Der er ikke blevet brugt ret meget energi på at dokumentere systemtests af programmet. Detter er blevet nedprioriteret til fordel for en større fokus på at implementere alle ønskede funktioner. Havde der været mere tid, kunne det have været godt at lave nogle flere brugertests og systemtests.

13 Konklusion

13.1 Produktet

Fra kravlisten, er alle krav af prioritet A og B nået at blive implementeret. Kun enkelte af de funktionelle C-krav er ikke blevet nået, eksempelvis den sidste brugeradministrator ikke skal kunne deaktivere sig selv. Disse krav er til gengæld ikke af nogen større betydning, og gruppen anser alle vigtige krav som værende opfyldt.

På trods af at alle de vigtige formulderede krav er nået, er der stadig plads til forbedring i projektet. Havde der været mere tid til at udvikle, havde det helt klart været en prioritet at gøre mere ud af fejlhåndteringen i backend-delen af programmet. Konkret må det siges at være lidt ærgerligt kun at benytte sig af en Exception i Data Access -laget, DALException. Skulle man videreudvikle programmet, eller lave noget lignende igen, ville det være langt bedre at benytte sig af mere specialiserede Exceptions, der beskriver nøjagtig hvilken fejl der er opstået. Dette kom gruppen udenom ved at vedhæfte ekstra fejlbeskeder til DALExceptions, hvilket lader til at have været en fin midlertidig løsning.

Desuden skulle man, hvis der havde været tid til det, naturligvis også have testet programmet yderligere igennem. Gruppen anser dog programmet for at være ret grundigt gennemtestet for alle væsentlige funktioner, omend ikke ret mange af disse tests er blevet dokumenteret.

Overordnet har det været et vellykket projekt. Produktet ser ud til at virke på en tilfredsstillende måde og kører stabilt uden væsentlige fejl. Det lever op til alle vigtige krav, og gruppen er tilfreds med arbejdet.

13.2 Forløbet

Det har naturligvis givet nogle udfordringer for gruppearbejdet, at alting har skulle foregå online. Efter omstændighederne, må det siges at være gået godt med disse udfordringer alligevel. Det online gruppearbejde samt den online undervisning kan dog ikke siges at fungere lige så godt som det fysiske, og det har selvfølgelig også har en betydning for gruppens præstation.

Det har været udmærket at køre databasen over SQL gennem Amazon. Det har gjort arbejdet lettere på mange måder, omend det også har været en ekstra udfordring at benytte SQL. Gruppen har dog fået meget ud af at arbejde med SQL i praksis.

14 Bilag

14.1 Bilag 1: SQL Script

14.1.1 Tabeldefinitioner

```
   #DROP DATABASE IF EXISTS MedicinalDb;
CREATE DATABASE MedicinalDb;
USE MedicinalDb;

create table Recepter
(receptId decimal(8,0),
receptNavn Varchar(20),
primary key(receptId)
);

create table Raavarer
(raavareId decimal(8,0),
raavareNavn Varchar(20),
leverandoer Varchar(20),
Primary key(raavareId)
);

create table ReceptKomp
(nonNetto Decimal(7,4),
tolerance Decimal(3,1),
receptId decimal(8,0),
raavareId decimal(8,0),
Primary key(raavareId,receptId),
Foreign key(receptId) references Recepter(receptId) on delete cascade,
Foreign key(raavareId) references Raavarer(raavareId) on delete cascade
);

create table Brugere
(brugerId decimal(3,0),
brugerNavn Varchar(20),
ini Varchar(4),
cpr char(10),
rolle ENUM('Administrator','Farmaceut','Produktionsleder','Laborant'),
aktiv TINYINT(1),
Primary key(brugerId)
);

create table RaavareBatches
(rbId decimal(8,0),
maengde decimal(7,4),
raavareId decimal(8,0),
primary key(rbId),
foreign key(raavareId) references Raavarer(raavareId) on delete cascade
);

create table ProduktBatches
(pbId decimal(8,0),
statuss decimal(1,0),
receptId decimal(8,0),
primary key(pbId),
foreign key(receptId) references Recepter(receptId) on delete cascade
);

create table ProduktBatchKomp
(tara decimal(7,4),
netto decimal(7,4),
brugerId decimal(3,0),
pbId decimal(8,0),
rbId decimal(8,0),
primary key(pbId,rbId),
foreign key(pbId) references ProduktBatches(pbId) on delete cascade,
foreign key(rbId) references RaavareBatches(rbId) on delete cascade,
foreign key(brugerId) references Brugere(brugerId) on delete cascade
```

```
);
```

14.1.2 Testdata

```
 Use   MedicinalDb;
#-----------------------------------------------Test
    data----------------------------------#
insert into Brugere Values('1','Morten
    Jeppesen','MJ','1210854535','Administrator',1);
insert into Brugere Values('2','Julie S rensen','JS','0209903217','Farmaceut',1);
insert into Brugere values('3', 'Annie Hansen', 'AH','2807751213',
    'Produktionsleder',1) ;
insert into Brugere values('4', 'Karl Max', 'KM','2302987832', 'Laborant',1) ;
insert into Brugere Values('5','Jacob
    Jensen','JJ','1210854535','Administrator',1);
insert into Brugere Values('6','Torben Test','TT','1903702288','Administrator',1);
insert into Brugere Values('7','Anna Witt','AW','2310004558','Administrator',1);
insert into Brugere Values('8','Karin
    Komputer','KK','1710855735','Administrator',1);

Insert into Recepter values('1234','Paracettamol');
Insert into Recepter values('10203040','Ibuprofen');
Insert into Recepter values('52689435','Vitaminpiller');
Insert into Recepter values('1','Kosttilskud A');
Insert into Recepter values('2','Kosttilskud B');
Insert into Recepter values('3','Kosttilskud C');

#NB: Recepterne skal eksistere, og man skal benytte et eksisterende receptID p
    3 plads,
Insert into ProduktBatches values('4321','0','1234');
Insert into ProduktBatches values('40302010','0','10203040');
Insert into ProduktBatches values('12589647','0','1');
Insert into ProduktBatches values('1','0','1');
Insert into ProduktBatches values('2','0','2');

Insert into Raavarer values('5678','Aloe Vera', 'Special-planter');
Insert into Raavarer values('20304050','Natur kalk', 'Vester kalkbrud');
Insert into Raavarer values('1','Salt', '   stersens    Salt');
Insert into Raavarer values('2','Sukker', 'Plantage 35C');
Insert into Raavarer values('3','Natron', 'Novo Nordisk');
Insert into Raavarer values('4','Konserveringsmiddel', 'Novo Nordisk');


#NB:kr ver eksisterende r varer. R vareID p   3 plads
Insert into RaavareBatches values('1234','605,85','5678');
Insert into RaavareBatches values('70605040','5,7','20304050');
Insert into RaavareBatches values('1','150','5678');
Insert into RaavareBatches values('2','154,7','1');
Insert into RaavareBatches values('3','239,4','1');
Insert into RaavareBatches values('4','435,85','2');
Insert into RaavareBatches values('5','35,7','3');
Insert into RaavareBatches values('6','210,4','4');
Insert into RaavareBatches values('7','79,4','1');
Insert into RaavareBatches values('8','115,85','2');
Insert into RaavareBatches values('9','325,7','3');
Insert into RaavareBatches values('10','110,4','4');

#NB: kr ver recept og r varer der eksisterer
#Table(nonNetto, tolerance, receptID, raavareId)
Insert into ReceptKomp values('20,5689','0,8','1234','5678');
Insert into ReceptKomp values('80,0025','10,0','10203040','20304050');
Insert into ReceptKomp values('14','20,1','1','1');
Insert into ReceptKomp values('3','10,1','1','2');
Insert into ReceptKomp values('4','6','1','3');
Insert into ReceptKomp values('0.005','5,1','1','4');

#NB: kr ver r varerBatch, ProduktBatch og brugere, Table: (tara, netto,
    brugerID, pbId, rbId)
Insert into ProduktBatchKomp values('46,2','86,56','3','4321','1234');
Insert into ProduktBatchKomp values('6,4','7,06','4','40302010','70605040');
```

```
60 Insert into ProduktBatchKomp values('96,3','27,0','4','12589647','1');
```