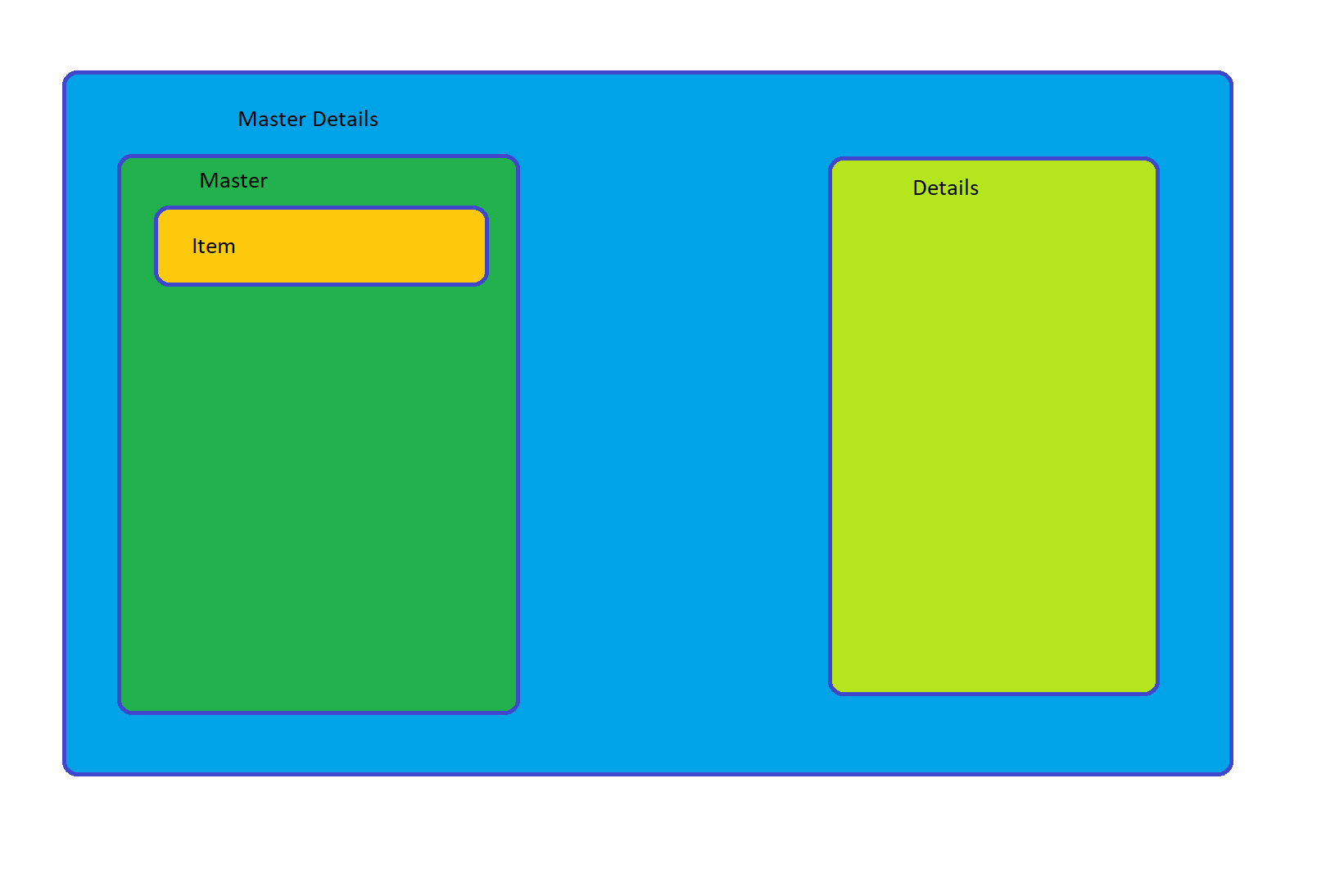
Vores udfordring med koden har ikke været domæne koden, men viewmodel koden. Da vores domæne logik er relativ simpel har vores udfordring været at få det vist i vores view på den måde vi ønsker.

Vi har hovedsageligt gjort brug af en master details model, til visning af vores data, som illusteret på figur 1.

Figur

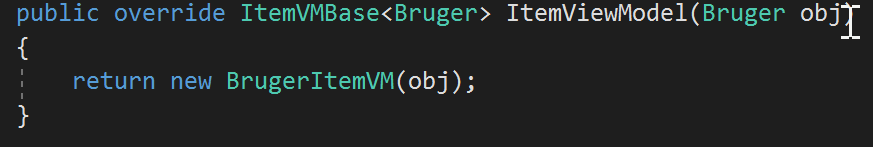
Dette bliver gjort ved hjælp af 5 klasser: Details-Klasse, Master-Klasse, Item-Klasse, Factory-Klasse og en Master-Details klasse.

Vi har forsøgt at lave vores program så generalt som muligt, inden for vores evner. Det gør sig gældende i at vi har lavet de 5 før nævnte klasser generic, dem kalder vi baseklasser, så alle klasser som skal repræsenteres som et master details view kan parametiseres i baseklasserne.

Master details forholdet bliver brugt i næsten hvert af vores views og derfor er der generet en del kode med de 5 base klasser så vi vil gennemgå dem kort herunder:

**FactoryVmBase**

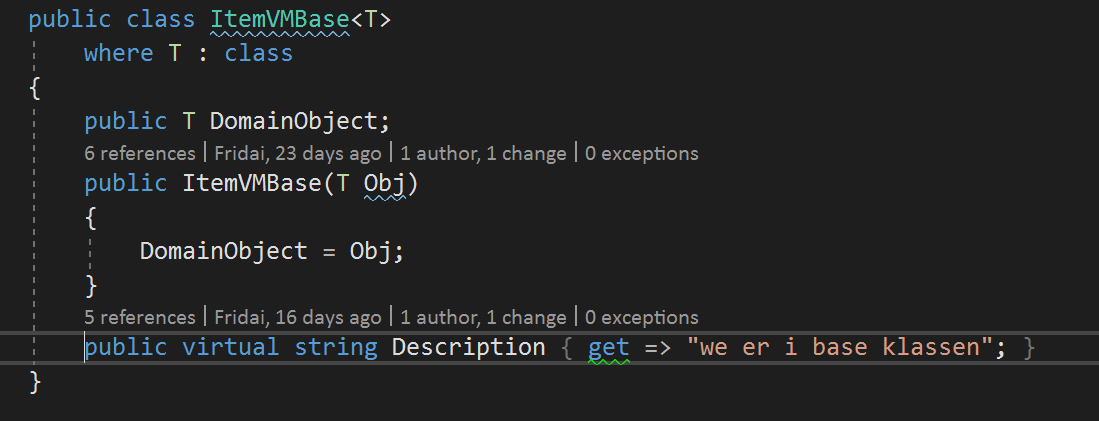
Factory klassen er har til ansvar at tage et model objekt, så som bruger og laver det til et viewmodel objekt, så som BrugerItemVm.



Figur

**ItemViewModelBase**

ItemViewModelBase klassen har til ansvar at skulle være logik til de enkelte objekter der bliver vist i master delen af master details, eller sagt på en anden måde er de individuelle objekter du kan se og trykke på i listviewet.



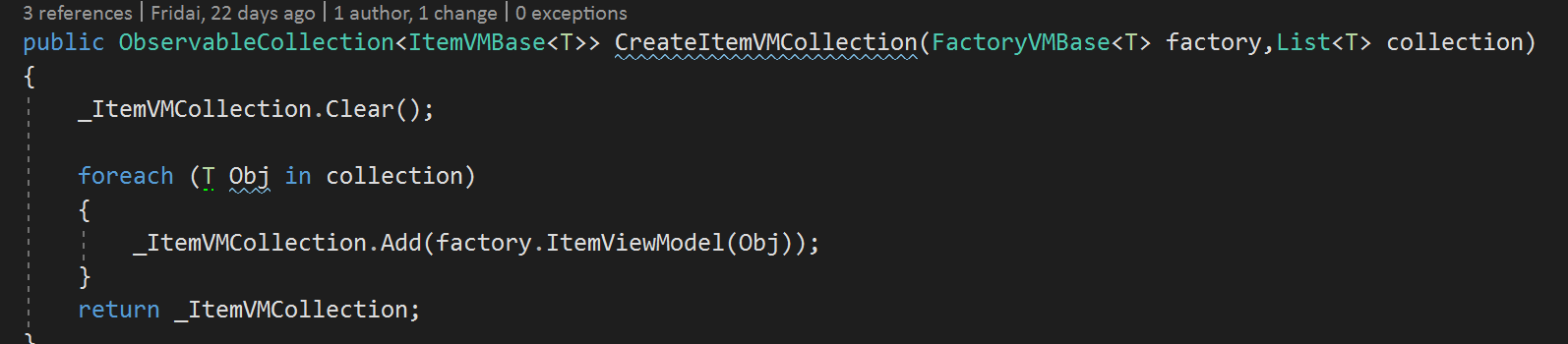
Figur

Klassen har en enkelt propertie, som er gjort virtuel så i den paramatiseret klasse, kan vi overwrite den med den aktuelle beskrivelse.

**MasterVmBase**

Masterviewmodelbasen har til ansvar at tage en normal liste af objekter og lave den om til et observeable liste af viewmodel objekter, vi er nød til at bruge observeablelist da en normal liste ikke kan bruges til visning i et listview, i XAML koden.

Det gøres ved at give metoden en factoryVmBase med af typen som er relevant, samt den list, man ønsker lavet om til en observeablelist.



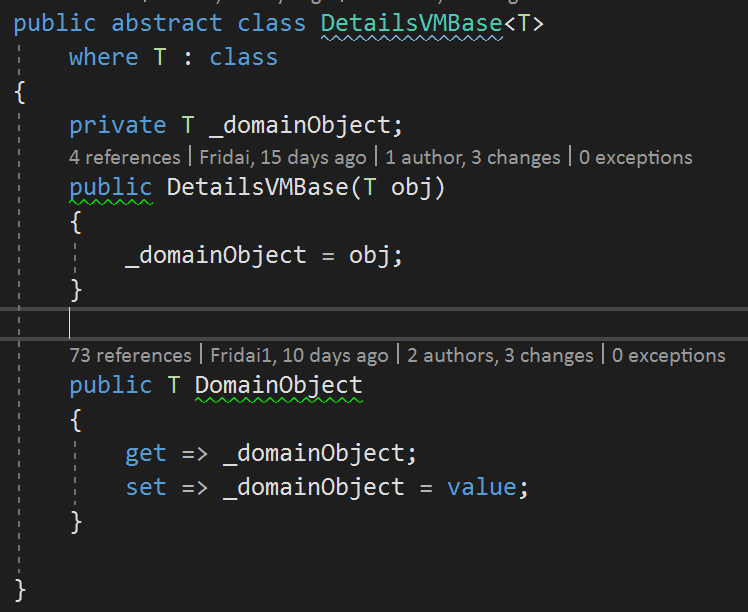
Figur

Vi starter med at .Clear, kollektionen da vi godt kan kalde på den samme metode flere gange og ikke ønsker kopier af det samme objekt i vores liste mere en 1 gang.

Derefter laver vi et simpelt foreeach løkke som tilføjer alle objekter fra den medgivet liste til den nye observeablekolletion, med metoden fra den medgivet factory, med det medgivet objekt som parameter.

**DetailsVmBase**

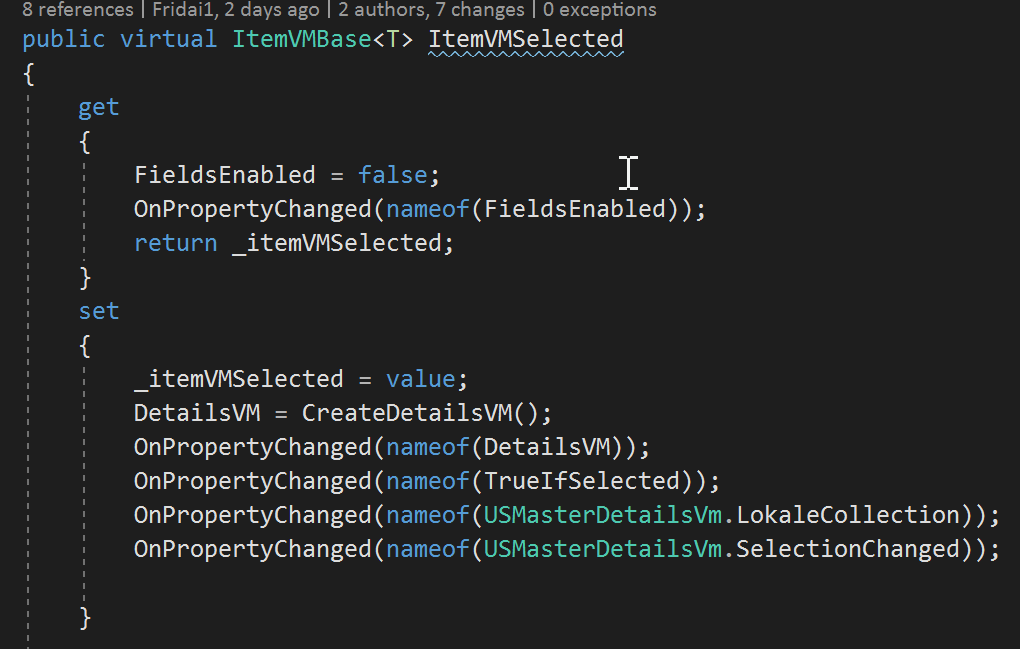
DetailsViewModelBase, har til ansvar at fungere som en mellemled mellem viewet og en model klasse, så der kan skrives og hentes fra det pågældende objekt.



Figur

**MasterDetailsVmBase**

Masterdetailsviewmodelbase klassen er hvor det hele bliver samlet og instanceret, samt generalt er klassen ansvarlig for alt hvad der har med master details forholdet at gøre, så som knapper og relevante metoder/properties.

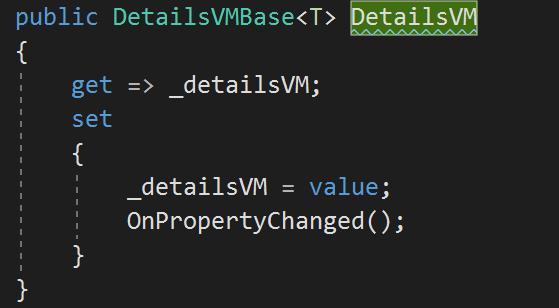
For at viewet kan fungere er vi nød til at havde nogle Properties som kan binde de før omtalte klasser sammen, vi starter med ItemVMSelected

Figur

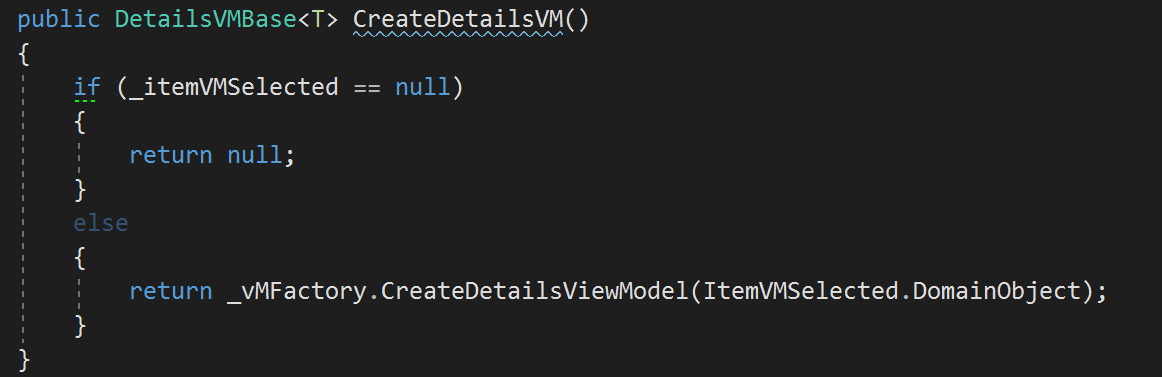
På figur 6 kan det ses at vores propertie har en Get og en Set. Når vi ber om data fra propertien, bliver en bool sat til false, denne bool er bindet til viewet og er ansvarlig for at gører felter enables eller disabled, vi ønsker ikke at brugeren kan redigere i objektet før de trykker på en rediger knap.

OnpropertyChanged bliver kaldt for at gører FieldsEnabled properien opmærksom på at den er blevet ændret.

I SET, sætter vi Instancefieldet  lig med den valgte itemviewmodel i listviewet. Der efter sætter vi DetailsVM propertien lig med metoden CreateDetailsVM().



Figur



Figur

Så det valgte viewmodel objekt i listviewet bliver brugt til at oprette et details view ved hjælp af Factory, som set på figur 8. DetailsVM propertien bliver så bindet til UI, hvor man så er i stand til at gå ned og vælge den attribute på det valgte objekt. .

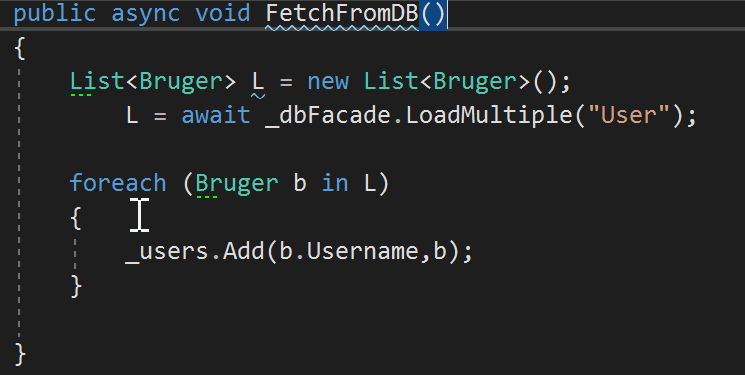
**OverView viewmodel**

Da der i et UI vindue også foregår mere end bare et master details forhold så som Login, har vi valgt at instancere alle relavente klasser til det enkelte view i en overview klasse, for eksempel BrugerOverview.

**LoginVM**

LoginVm klassen er ansvarlig for at styrer hvem der bliver lukket ind i systemet og gemme den bruger som bliver lukket ind i en static attribut. Vi bruger en static attribut så vi, i hver klasse kan sætte \_loggedInBruger  attributen lig den statiske attribute .

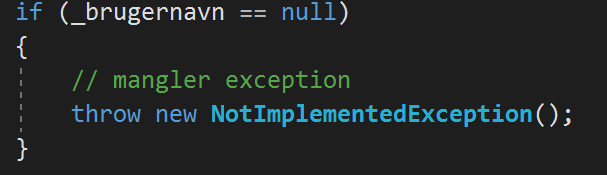
Når LoginVm bliver instanceret kører den en metode i constructoren FetchFromDB().



Som laver en ny liste og gør den lig med en liste af bruger objekter fra databasen, derefter bliver de bruger objekter tilføjet til en dictionary, med brugerens brugernavn som key og bruger objektet som value.

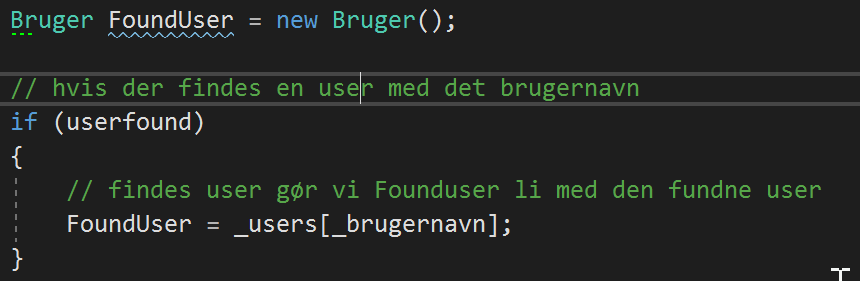
Grunde til at vi benytter en dictionary er primært pågrund af 2 årsager:

* Når der kommer mange brugere og der skal søges efter en bruger er en dictionary meget hurtigere.
* Ved at bruge en dictionary kan vi lettere udfører opgaven at søge efter de 2 atributes vi ønsker og sætte den endelige bruger lig med den statiske attribut givet oplysningerne er korrekte.

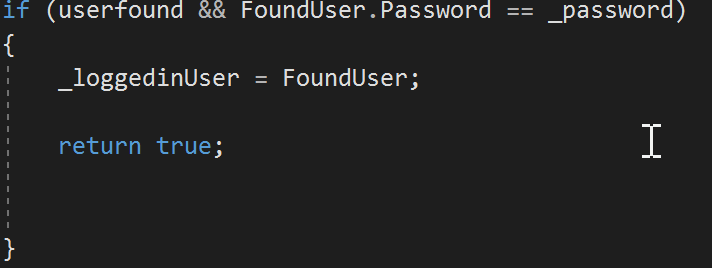
Selve login metoden er af typen bool. Den starter med et sanity check 

Hvorefter hvis brugernavn != null, 

Ser vi om det username som er indtastet også findes i vores dictionary, vi søger efter key.

Hvis brugeren findes gør vi den fundne brugere lig med en attribute af typen bruger 

Til sidst checker vi for om den fundne brugers password stemmer overens med, det password som er blevet indtastet, er det korrekt gør vi den statiske attribute lig med den fundne bruger, og returner true.



Hvis det er ikke er gældende, sætter vi brugernavn og password til null og returner false.

**Problemet:**

Problemet vi løb ind i med dette var at få det bindet til en knap og få den til at kører metoden før den sendte brugeren videre. I første omgang checkede den koden men om den så var true eller false sendte den alligevel brugeren videre, ved hjælp af Click eventen i XAML .

**Løsningen:**

Løsningen blev at kører commandoen fra codebehind.