# Design, implementering og test

## Fridge app

### Design

### Implementering

### Test

## Web app

### Design

Beskrivelse af designprocessen. Hvilke designløsninger man har overvejet samt de valg man har gjort og begrundelsen herfor.

Asp.net frem for KUN html -> funktionalitet

Valget af ASP.NET er basseret på den funktionalitet som frameworked giver frem for udelukkende at lave en side der bruger HTML. Vha. ASP.Net er der mulighed for at manipulere med det data som brugeren indtaster og det data der ligger i databasen, i C#.

ASP.Net MVC (Razor) vs. Webform

Når der laves en webapplikation i ASP.Net skal der tages et valg om hvilke frameworks og mønstre man ønsker at bruge til at udvikle applikationen. Da der skulle besluttes hvordan vi ville udvikle applikationen, stod vi med to metoder: ASP.Net Web Form og ASP.Net MVC. WebForm er benytter sig begge af html, men ifølge forskellige website (bl.a (Sukesh, 2015) og (Hambrick, 2013)), har udvikleren mere kontrol over renderingen af den html kode som brugeren ender med i browseren, hvis der benyttes MVC i forhold i WebForm.

De ASP.Net MVC vi benytter os af er cshtml filer (Razor). Det er valgt på baggrund af hvor nemt det er at bruge. For at lave et View skal man kun kunne html, og hvis der skal tilføjes funktionalitet kan man vælge at kalde en funktion i en controller eller skrive c# kode direkte Viewet vha. ’@’-tagget. WebForm bruger Drag-N-Drop design metoden i stedet for html kodning. Det gør det mere visuelt at designe en applikation, men dette har ikke været en nødvendighed for os.

MVC -> Ikke mange overvejelser? Det var det vi lærte om?

Pro:

– Scale better

• Suitable for large web sites

• Suitable for high traffic sites

– Supports unit testing

MSSQL vs. MySQL -> nemmere synkronisering med allerede oprettet db

Til at lagre applikationens data er MSSQL valgt over MySQL. Grunden til dette er muligheden for synkronisering mellem den online database og den lokale. Det er muligt at synkronisere i mellem en MySQL og en MSSQL, men da Windows applikationen allerede benytter sig af MSSQL, var det nemmere at lave tilsvarende til webapplikationen. Se mere omkring dette i ’Database’-afsnittet.

### Implementering

Beskrivelse af implementeringsprocessen og de færdige løsninger. Her skal man kun medtage kode og kredsløbsdiagram udsnit for særlig interessante udvalgte dele af projektet.

Til oprettelsen af web applikations projektet blev Visual Studio brugt. VS hjalp med at oprette et projekt der benytter sig af MVC-mønstret, og bruger Razor engine. Ydermere sættes det nemt op til Azure. Mappestrukturen bliver herved givet af VS. Mere om dette ses i projektdokumentation **side XX**.

Ikke den bedste implementeringsprocess pga. uddelegering af View og Controller -> manglende erfaring?

Evt. kodestump hvordan funktioner kaldes fra View.

Html Helpers

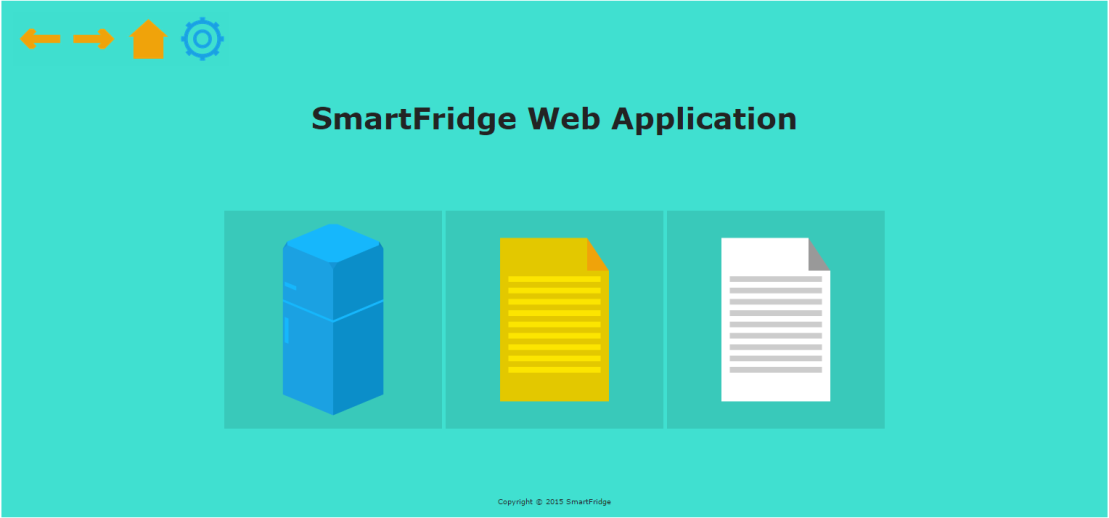
Når data skal sendes fra et view til en controller benyttes html.helpers. Med disse hjælpere er det muligt at kalde en funktion i en controller og sende information med. F. eks. Information om et nyt oprettet item. Ydermere findes der knapper og tekstbokse, som bliver renderet på viewet som html kode. Disse knapper hjælper også med at eksekvere kode i controlleren.

Razor syntax til en html.helper:

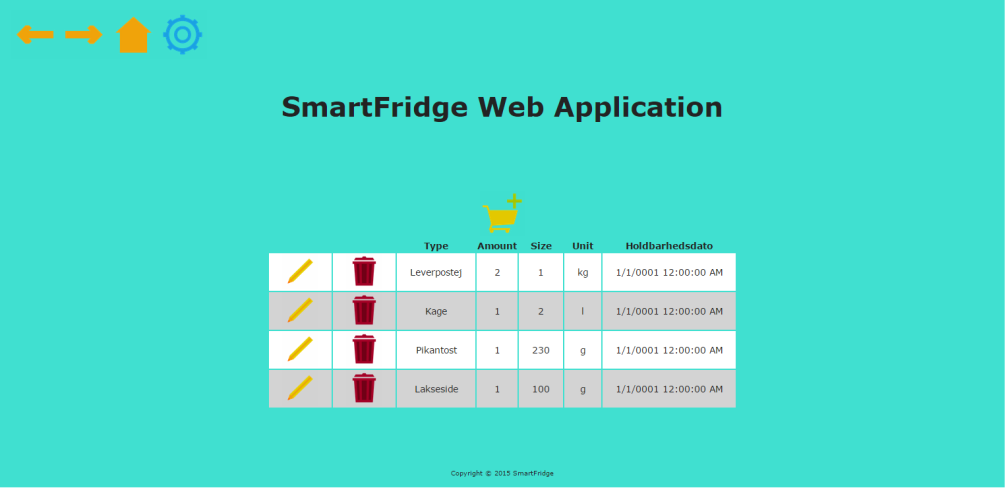
@Html.ActionLink("About this Website", "About") (Reference(w3schools, 2015))

#### View

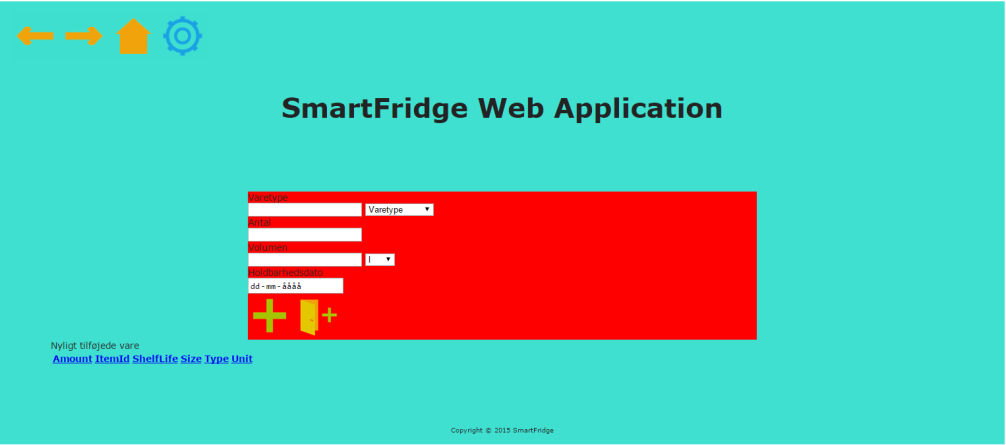
Grundet tidspres er der ikke blevet fokuseret så meget på view delen af web applikationen. Til trods for dette, er de kommet til nogenlunde at ligne det designede. Men der har været fokus på funktionalitet over udseende.

På **Figur 1** kan ses Index, der svarer til hovedmenuen fra Fridge-app, herfra kan en eksisterende liste vælges.

**Figur 1** Index

Når en liste er valgt vil blive præsenteret for et view, der minder om det, som kan ses på **Figur 2**. Dette view viser inholdet af den valgte liste, og giver mulighed for at navigere videre til enten tilføj vare eller rediger vare. Vælges tilføj vare bliver man ført videre til vinduet på **Figur 3**, hvor fra man kan tilføje varer til den nuværende liste. Vælges derimod rediger vare, bliver man præsenteret for vinduet på **Figur 4**, hvor tekstboksene vil være fyldt med den nuværende vareinformation. Foretages der ændringer gemmes disse når man trykker på fluetbenet, hvorefter man returneres til den liste, man var på før. Trykker man på krydset returneres man bare til listen og vareinformationen er som den var før.

**Figur 2** Køleskabs listen.



**Figur 3** Tilføj vare vinduet.



**Figur 4** Rediger vare vinduet.

#### Controllers

Samlet set udgør controllerne det, der svarer til business logic layer fra Fridge-app, og de varetager altså kommunikationen med den eksterne database. Hver controller har sit ansvar. LisView står for at indlæse og præsentere alle varer på en liste, og giver mulighed for at slette en udvalgt vare. Den står også for at omdirigere til både Additem viewed og EditItem viewed. Inden den omdirigere til EditItem, finder den først den varer, der skal sendes med til denne før der omdirigeres. AddItem controlleren har til ansvar at tilføje varer til databasen, hvis den finder en tilsvarende vare på den nuværende liste sørge den for blot at tælle antallet af eksisterende vare op i stedet for at indsætte en duplet. EditItem controlleren står for redigering og opdatering af en valgt vare. EditItem sørger også for at en vare ikke kan opdateres så den bliver en duplet på listen, dette håndteres ligesom i AddItem.

#### Cache

Til web applikationen er der blevet implementeret en cache. Denne cache indeholder alle de ting, som er fælles for alle controllers. Klassen består kun af properties, der alle er static. Begrundelsen for dette er netop at det er fælles data, som alle controllers skal arbejde på. Det er også i cachen at facaden til data access layer ligger, dette er fordi der på alle tidspunkter kun må være en facade.

### Test

Da webapplikationen er bygget op omkring MVC-mønstret skal der kun testes på controller og model klasserne i projektet. Det er nemlig der funktionaliteten og logikken i applikationen ligger. Pga. den lave kobling, som dette mønstre giver, er unit tests nemme at lave. Udfordringen i dette projekt har været at koble det logik, som er lavet til applikationen, til webapplikationen. For at sikre os kontinuitet både i applikationen webapplikationen, er logikken blevet genbrugt og de tests der derfor blev lavet til applikationen kan bruges til webapplikationen. Nogen dog tilpasset, da de ikke bruger samme database.[[1]](#footnote-1)

Et af problemerne ved den måde, som funktionerne i applikationen er opbygget på, er at mange af dem returnere et View eller et ’RedirectToAction’ (Figur 1) efter et item har gennemgået anden logik i samme funktion. Et eksempel på dette kunne være i EditItemController hvor funktionen UpdateItem er implementeret. Denne funktion tager imod en FormCollection hvori de nye værdier der skal ændres i det gamle item ligger i. logikken i funktionen ændre det gamle item til de nye værdier, men returnere det view, som skal vises efter itemet er ændret. Dvs. det ikke er muligt bare at sammenligne returværdien fra funktionen med det der forventes at det pågældende item er ændret til. I stedet skal der testes på om der eksistere et item, i listen med items, der har de værdier der skulle ændres. Dette går igen med alle de funktioner der har ’ActionResult’ som retur parameter.



Figur 1 - Udsnit af hvad der returneres fra UpdateItem-funktionen i EditItemController

1. Webapplikationen benytter sig af MySQL, hvorimod applikationen benytter sig af MS SQL. [↑](#footnote-ref-1)