# Design og implementering

## Web app

Figur - WebApp mappestruktur. MVC pattern

### Design

Mappestrukturen (Figur 1) i web applikations projektet er generet af Visual Studio under oprettelsen af projektet. Den følger MVC mønstret og har derfor mapperne: Controllers, Views og Models. Models mappen er i dette projekt gjort overflødigt da de klasser der burde ligge i denne mappe ligger i et class library ved navn: ”SmartFride\_WebModels”, som har taget udgangspunkt i modellaget fra den anden applikation.

App\_Data mappen bruges ikke i dette projekt da al data der skal gemmes i denne applikation gemmes i en database.

App\_Start indeholder klasser der henter scripts og bestemmer hvilken side der skal vises under opstart af applikationen. Disse er autogenerede og ikke ændret i.

Content mappen indeholder de .cs stylesheet, som hvert view bruger for at style siderne.

Images, fonts og Scripts mapperne indeholder henholdsvis de billeder, de scripts og de fonts, som bruges i applikationen.

I Web.config (XML fil) indeholder information om hvilke assamblys applikationen skal hente, hvilke connectionsstrings den skal bruge og meget mere.

Global.asax (cs fil) kører de filer der ligger i App\_Start under opstart af applikationen.

Hver controller har et View tilknyttet, hvor controlleren sørger for funktionalitet og Viewet sørger for at rendere html kode, som browseren kan læse og brugeren herved kan se i sin browser.

Dette projekt består af 4 controllers:

* HomeCotroller
* LisViewController
* AddItemController
* EditItemController

### Design af views

De individuelle views følger stort set samme design som deres modparter i Fridge-app, dog med et par forskelle. Da det er en web applikation, og den derfor vil kører i en browser, er luk knappen overflødig og derfor ikke taget med. En anden følge af at det kører i browser har været at uret fra Fridge-app har virket overflødigt da uanset, hvilken enhed den tilgås fra har let adgang til et ur på anden vis. Udover disse to ting er "rammen" den samme som på Fridge-app og har altså stadig navigations knapperne. Da rammen er den samme vil denne ikke være med på de wireframes, der er i dette afsnit. Endnu en begrundelse for de mindre forskelle, der er på udseendet af de to applikationer, er at web applikationen, også skal kunne bruges på mindre enheder, hvor for mange ting i hvert view kunne besværliggøre brug.

#### C:\Users\Mikkel\AppData\Local\Microsoft\Windows\INetCache\Content.Word\AddItem wireframe.pngAdditem

**Figur 2** Wireframe for Additem

Dette view svarer til "Tilføj varer" fra Fridge-app. Der er stort set ikke ændret på udseendet af dette view. Den eneste forskel er, at listen over nyligt tilføjede varer er placeret under selve tilføjelses menuen i stedet for ved siden af, som det ses ud fra det tilhørende wireframe på **Figur 2**.

#### Edititem

Det er ved edit item, at der er sket den største forandring. I Fridge-app var denne en del af det vindue, hvor man kunne se alle vare på en given liste, på web applikationen har den fået et vindue for sig selv. Grunden til dette var, at der ved brug af mindre enheder ville gå udover brugervenligheden, hvis der var for mange ting på et view. Som det kan ses på **Figur 3**, minder den rigtig meget om både boksen med vareinformation fra Fridge-apps List view, og om Add items view.

#### EditItem wireframeIndex

Figur Wireframe for EditItem

Index svarer til hovedmenuen i Fridge-app, hertil har vi valgt at beholde det design, der var lavet der til.

#### ListView

Som det fremgår af Figur 4 har List viewet været igennem nogle ændringer. Det mest iøjefaldende er nok at boksen med vareinformation er væk. Dette skyldes at al information vedrørende en vare kunne vises i gridded, og at funktionaliteten vedrørende redigering af vareinformation blev flyttet ud i et view for sig. Udover dette er der ikke blevet ændret der store ved viewet udover at der i griddet til vareinformation er kommet en knap, der fører til EditItem viewet, til venstre for slet knappen.

### D:\Dokumenter\IHA IKT\4. Semester\I4PRJ4\Design og implementering\Design\Wireframes\LisView wireframe.pngImplementering

#### Controllers

##### AddItemController

AddItem består af én constructor og 2 funktioner.

Constructoren indlæser items fra databasen vha. Unit Of Work, og tilføjer herefter alle items til ListGuiItems. model bliver sat til newGuiItems, som er de items der skal tilføjes i addNewItem-funktionen. Dette ses på Kodestump 1. Constructoren returnere et View, som sørger for at AddItemViewet bliver vist for brugeren.

Figur Wireframe for List view

|  |
| --- |
| public ActionResult AddItem()  {  var uow = Cache.DalFacade.GetUnitOfWork();  ListGuiItemTypes.Add(new SelectListItem { Text = "Varetype", Value = "Varetype", Selected = true });  foreach (var guiItemTypes in uow.ItemRepo.GetAll())  {  ListGuiItemTypes.Add(new SelectListItem { Text = guiItemTypes.ItemName });  }  model = newGuiItems;  ViewBag.ListNewGuiItems = ListGuiItemTypes;  uow.SaveChanges();  Cache.DalFacade.DisposeUnitOfWork();  return View(model);  } |

Kodestump – AddItemController’s constructor

Funktionen addNewItem tager imod de indtastede data og tilføjer dem til guiItemToAdd. Men først tjekkes der på om den holdbarhedsdato der medsendes er indtastet. Er denne ikke det, sættes det holdbarhedsdatoen til en default værdi. Ellers bliver den konverteret til den rigtige DateTime. Den skal konverteres da den modtager værdien som en string, men for at databasen kan forstå denne værdi skal den være af typen DateTime.

|  |
| --- |
| public ActionResult addNewItem(string Varetype, string Antal, string Volume, string Enhed, string Holdbarhedsdato, string ItemImgClicked)  […]  GUIItem guiItemToAdd = new GUIItem();  guiItemToAdd.ShelfLife = dblistItemDateTime; //AMOUNT READ FROM FIELD  guiItemToAdd.Amount = Convert.ToUInt32(Antal); //Antal READ FROM FIELD  guiItemToAdd.Size = Convert.ToUInt32(Volume); //Volume READ FROM FIELD  guiItemToAdd.Type = Varetype; //Varetype READ FROM FIELD  guiItemToAdd.Unit = Enhed; //unit READ FROM FIELD  foreach (var newGuiItem in newGuiItems)  {  if (newGuiItem.Type.Equals(guiItemToAdd.Type) &&  newGuiItem.Size.Equals(guiItemToAdd.Size) &&  newGuiItem.Unit.Equals(guiItemToAdd.Unit) &&  newGuiItem.ShelfLife.Equals(guiItemToAdd.ShelfLife))  {  newGuiItem.Amount += guiItemToAdd.Amount;  return null;//Viewet skal dog opdateres først  }  }  newGuiItems.Add(guiItemToAdd);  […]  model = newGuiItems;  ViewBag.ListNewGuiItems = ListGuiItemTypes;  return PartialView("~/Views/AddItem/AddItem.cshtml",model); |

Kodestump – udsnit af AdditemController’s addNewItem funktion

På Kodestump 2 ses hvordan der tjekkes om det item der skal tilføjes allerede eksistere. Hvis den allerede gør det, ændres der kun på dette items Amount i stedet for at tilføje et nyt. Findes item ikke i listen bliver det tilføjet. Når det nye item er tilføjet lægges det i den liste som skal vises (model) og viewet returneres. Den nye item er nu tilføjet og kan ses i tabellen.

I addNewItem funktionen tjekkes der også på parameteren ’ItemImgClicked’. Hvis denne er lig ”Exit”, betyder det at knappen ’Tilføj og Exit’ er trykket på. Er dette tilfældet bliver funktionen Exit i AddItemControlleren kaldt.

|  |
| --- |
| public ActionResult Exit()  {  if (newGuiItems.Count == 0) {return null;}  var uow = Cache.DalFacade.GetUnitOfWork();  foreach (var newGuiItem in newGuiItems)  {  //Searching for item in DB  Item dbItem = uow.ItemRepo.Find(l => l.ItemName == newGuiItem.Type);  if (dbItem == null)  {  dbItem = new Item(newGuiItem.Type){StdUnit = newGuiItem.Unit, StdVolume = (int)newGuiItem.Size};  uow.ItemRepo.Add(dbItem);  }  //Searching for ListItem in DB  ListItem dbListItem = uow.ListItemRepo.Find(l =>  l.List.ListName == Cache.CurrentList.ListName &&  l.Item.ItemName == newGuiItem.Type &&  l.Unit == newGuiItem.Unit &&  l.Volume == newGuiItem.Size &&  l.ShelfLife == newGuiItem.ShelfLife);  if (dbListItem != null)  {  dbListItem.Amount += (int)newGuiItem.Amount;  }  else if (dbListItem == null)  {  dbListItem = new ListItem();  dbListItem.ShelfLife = newGuiItem.ShelfLife;  dbListItem.Amount = (int)newGuiItem.Amount;  dbListItem.Volume = (int)newGuiItem.Size;  dbListItem.Unit = newGuiItem.Unit;  dbListItem.ListId = Cache.CurrentList.ListId;  //dbListItem.List = Cache.CurrentList;  dbListItem.ItemId = dbItem.ItemId;  dbListItem.Item = dbItem;    uow.ListItemRepo.Add(dbListItem);  }  }  uow.SaveChanges();  Cache.DalFacade.DisposeUnitOfWork();  newGuiItems = new List<GUIItem>();  model = newGuiItems;  return RedirectToAction("ListView", "LisView"); |

Kodestump – Udsnit af AddItemController’s Exit-funktion

Exit funktionen (på Kodestump 3) tjekker først og fremmeste på om der er tilføjet nogen items. Er der ikke det returneres der null. Er dette ikke tilfældet skal der tjekkes på om hvert item der er tilføjet ligger i databasen. Der ledes først i item listen om det nye item eksistere. Herefter ledes der i ListItem for at se om det skulle lægge der. Findes de ikke bliver de tilføjet til databasen. Hvis det nye item ligger i ListItem, skal det nuværende items Amount tælles op. Efter alle nye item er gennemgået gemmes ændringerne i databasen og listerne med items opdateres. Til sidst returneres LisView Viewet.

##### EditItemController

EditItem controlleren har 3 forskellige funktioner, den første af disse er EditItem funktionen, der til dels kan kaldes for en constructor for EditItem viewet. Her indlæses alt det data, der skal bruges i viewet, såsom varetyper fra cachen og de forskellige enheder, der kan bruges til at beskrive en varer. Udover dette ligges, den valgte vare i ViewData, og varetyperne og enhederne ligges i ViewBag. Det hele kan ses på **Kodestump 4**.

|  |
| --- |
| public ActionResult EditItem(GUIItem oldItem)  {  \_oldItem = oldItem;  \_types = new List<SelectListItem>();  \_types.Add(new SelectListItem { Text = "Varetype", Value = "Varetype", Selected = true });  foreach (var item in Cache.DbItems)  {  \_types.Add(new SelectListItem { Text = item.ItemName,Value=item.ItemName });  }  \_units = new List<SelectListItem>(){new SelectListItem{Text = "l", Value = "l"},  new SelectListItem{Text = "dl", Value = "dl"},  new SelectListItem{Text = "ml", Value = "ml"},  new SelectListItem{Text = "kg", Value = "kg"},  new SelectListItem{Text = "g", Value = "g"},  new SelectListItem{Text = "stk", Value = "stk"}};  ViewData.Add("oldItem", \_oldItem);  ViewBag.types = \_types;  ViewBag.units = \_units;  return View();  } |

**Kodestump 4** Viewets "constructor"

Den anden funktion, der dækker over den primære opgave for Controlleren er UpdateItem. Denne funktion sørger for at de i viewet indtastede data bliver gemt ned i databasen, ved at opdatere enten det tilhørende ListItem, Item eller begge dele. Funktionen består af en serie af tjek som der kan ses et eksempel på, på **Kodestump 5**. tjekkene går alle sammen på at finde ud af hvad, der er blevet ændret og hvordan det skal håndteres. I tilfælde af en vare opdateres på en sådan måde, at det stemmer overens med allerede eksisterende vare på den liste, håndteres dette ved at tælle den originale vares antal op, og slette duplikatten.

|  |
| --- |
| if (listItem.Item.ItemId == \_oldItem.ItemId)  {  if ((listItem.Item.ItemName == \_updatedGUIItem.Type))  {  listItem.Amount = Convert.ToInt32(\_updatedGUIItem.Amount);  listItem.Volume = Convert.ToInt32(\_updatedGUIItem.Size);  listItem.Unit = \_updatedGUIItem.Unit;  listItem.ShelfLife = \_updatedGUIItem.ShelfLife;  uow.ListItemRepo.Update(listItem);  uow.SaveChanges();  Cache.DalFacade.DisposeUnitOfWork();  return RedirectToAction("ListView", "LisView");  }  [...] |

**Kodestump 5** Eksempel på Check i UpdateItem funktionen

Den tredje funktion i controlleren er selectedUnit funktionen. Denne funktion sørger for at det er den valgte vares enhed, der vises i dropdownmenuen i stedet for det øverste element. Som det kan ses på **Kodestump 6**, gøres dette ved at finde enheden i listen over enheder, og sætte dens Selected property til true.

|  |
| --- |
| public static void selectedUnit(GUIItem guiItem)  {  foreach (var unit in \_units)  {  if (unit.Text == guiItem.Unit)  {  unit.Selected = true;  break;  }  }  } |

**Kodestump 6** Den valgte items enhed findes, og sættes som   
den valgte med Selected propertyen.

##### HomeController

HomeController controlleren er den første controller, der kommer i brug. Når siden tilgås kaldes dens Index metode, som returnere Index viewed, det er også i denne metode, hvor cachens data acces layer facade instantieres, cachen er beskrevet længere nede i dette dokument. Udover denne metode har den også SetCurrentList funktionen, der kaldes når brugeren har trykket på en af de tre mulige knapper i Index viewed. Funktionen der kan ses på **Kodestump 7**, gør brug af cachens DAL facade til at finde den liste i databasen, der svarer til den Bruger valgte, og sætte den i cachen, før den returnere det relevante view.

|  |
| --- |
| public ActionResult SetCurrentList(string listToEdit)  {  List currentList = new List();  var uow = Cache.DalFacade.GetUnitOfWork();  currentList = uow.ListRepo.Find(l => l.ListName == listToEdit);  Cache.DalFacade.DisposeUnitOfWork();  Cache.CurrentList = currentList;  return RedirectToAction("ListView","LisView");  } |

**Kodestump 7** Her ses metoden, hvor den nuværende liste sættes i Cachen

##### LisViewController

LisViewController indeholder én constructor, 2 funktioner og en liste med de items der skal vises på ListView’s view. Constructoren henter de items der ligger i Cache klassen ind og tjekker først på om denne liste er tom. På Kodestump 8 tjekkes det både på CurrentListItems og DbItem om id’erne passe samme på hvert item. Hvis ListItem og Item’s id matcher skal navnet fra Item og amount, volumen, unit og shelflife fra ListItem lægges sammen til ét nyt item og lægges over i tempData listen. Efter der er blevet tjekket på alle items i listerne bliver de lagt ind i model som herefter returneres sammen med viewet.

|  |
| --- |
| Cache.CurrentList = Cache.CurrentList;  List<GUIItem> tempData = new List<GUIItem>();  if(Cache.CurrentListItems.Any())  {  foreach (var listItem in Cache.CurrentListItems)  {  foreach (var item in Cache.DbItems)  {  if(item.ItemId == listItem.ItemId)  {  GUIItem temp = new GUIItem(item.ItemName, (uint)listItem.Amount, (uint)listItem.Volume, listItem.Unit){ItemId = item.ItemId, ShelfLife = listItem.ShelfLife};  tempData.Add(temp);  }    }  }  }  model = tempData;  return View(model);  } |

Kodestump – Udsnit af ListViewController’s constructor.

Den første funktion i ListViewController er ToEditItem, hvis opgave er at få sendt det rigtige GUIItem over til EditItemController’en. Da det ikke var muligt at sende Id’et med fra viewet, så tjekker denne funktion på det medsendte GUIItem’s attributter om de stemmer overens med et item på listen. Når dette bliver fundet bliver EditItem funktionen kaldt i EditItemControlleren, og det fundne item sendes med over.

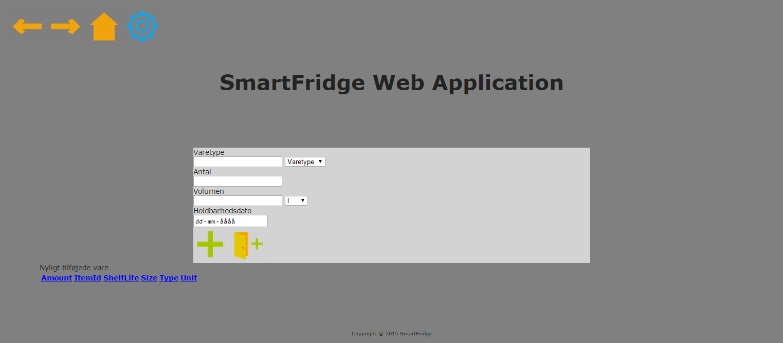
Den anden funktion i ListViewController er DeleteSelectedItem. Ligesom ToEditItem funktionen får den et GUIItem med fra viewet og tjekker på om dette item eksistere i model listen. Når/hvis det medsendte item bliver fundet konverteres IEnumerable model om til en liste så den kan rettes i. Herefter slettes itemet fra listen og den nye liste, uden det slettede item, lægges i model igen. Herefter skal itemet slettes fra databasen, hvilket gøres ved at finde og slette den i Cache’ns CurrentListItems. Til sidst gemmes ændringerne, chache’ns adgang til databasen bliver disposed og constructoren, der returnere LisView viewet, bliver kaldt.

|  |
| --- |
| […]  foreach (var item in model)  {  if (item.Type == itemToDelete.Type && item.Amount == itemToDelete.Amount && item.Size == itemToDelete.Size && item.Unit == itemToDelete.Unit)  {  List<GUIItem> tempList = model.ToList();  tempList.Remove(itemToDelete);  model = tempList;  foreach (var dblistitem in Cache.CurrentListItems)  {  if (dblistitem.ItemId == item.ItemId)  {  uow.ListItemRepo.Delete(uow.ListItemRepo.Find(l =>  l.ItemId == dblistitem.ItemId &&  l.ListId == dblistitem.ListId &&  l.Unit == dblistitem.Unit &&  l.Amount == dblistitem.Amount &&  l.ShelfLife == dblistitem.ShelfLife &&  l.Volume == dblistitem.Volume));  }  }  }  }  uow.SaveChanges();  Cache.DalFacade.DisposeUnitOfWork();  return RedirectToAction("ListView", "LisView");  […] |

Kodestump – Udsnit af DeleteSelectedItem funktionen i LisViewController

#### Views

##### AddItem

Ligesom i SmartFridge-applikationen har AddItem fået sit eget view. På Figur 1 ses en rød indramning der indeholder de fire inputs: Varetype, Antal, Volumen og Holdbarhedsdato. Ydermere indeholder den ’Tilføj’ knappen symboliseret af et plus og ’Tilføj og Exit’ knappen symboliseret af en dør og et plus. Under den røde indramning er tabellen med de items der er tilføjet uden at afslutte.

Figur - AddItem View

Viewet indeholder en IEnumerable med de GuiItems der er tilføjet uden at afslutte. Ligesom de andre views bruger den samme layout (\_Layout.cshtml) som wrapper til viewet. Disse ting ses på Kodestump 10.

|  |
| --- |
| @model IEnumerable<SmartFridge\_WebModels.GUIItem>  @{  ViewBag.Title = "AddItem";  Layout = "~/Views/Shared/\_Layout.cshtml";  } |

Kodestump – Udsnit af additem.cshtml. Header

Alle inputs er inde i div klassn ”NewItem” som ses på Kodestump 11. For alle inputs gælder det at det skal være inde i @using (Html.BeginForm()){}. Dette er gjort for at fortælle programmet hvilken information den skal sende med til addNewItem funktionen. Varetypen har to muligheder for at få et input. Enten kan brugeren vælge at skrive det ind i <Input> objektet eller vælge det på den @html.DropDownList der er ved siden af inputtet.

|  |
| --- |
| <div class="NewItem" style="background-color: red; margin: 0 22em 0 22em">  @using (Html.BeginForm("addNewItem", "AddItem"))  {  <label>Varetype</label>  <br />  <input type="text" name="Varetype" id="VaretypeText" required/>    @Html.DropDownList("ListNewGuiItems", (IEnumerable<SelectListItem>)ViewBag.ListNewGuiItems, new  {  @id = "VaretypeDropDown"  })  <br />  <label>Antal</label>  <br />  <input type="number" name="Antal" min="0" required />  <br />  […]  <input type="image" src="~/Images/Add.png" alt="Submit" class="AddImg" id="addItemImgClicked" name="ItemImgClicked" value="Add" />  <input type="image" src="~/Images/AddAndExit.png" alt="Submit" class="AddImg" name="ItemImgClicked" value="Exit" />  } |

Kodestump – Udsnit af AddItem.cshtml. Inputs

##### EditItem

Udover at det har fået sit eget view minder måden EditItem ser ud på rigtig meget om rediger vare fra Fridge-app. Varens information indlæses i tekstbokse, som kan redigeres i, og Bruger har mulighed for at vælge enten at gemme eller annullere de ændringer der er foretaget. Viewed er ikke blevet helt som designet, hovedsagligt er det fluebenet og krydset, der ikke er kommet til at sidde som på designet. Dette skyldes bøvl med at få to forskellige Html forms til at være på samme linje. Udover dette skulle det egentlig bare have været centreret.

**Figur 6** EditItem view

På **Kodestump 13** ses det allerførste, der sker i viewed. Den vare Bruger valgte i List viewed indlæses fra EditItem viewets ViewData, og der kaldes en funktion, der finder ud af, hvilken enhed, der skal være valgt i enheds dropdown menuen. Alle inputfelterne i viewed er lavet med Html helpers, på **Kodestump 14** kan der ses nogle eksempler på disse. Som det kan ses er der hovedsagligt brugt TextBox helperen, men i eksemplerne fra **Kodestump 14** kan det ses at der er manipuleret lidt med nogle af dem. Det er muligt at ændre på typen af TextBox helperen så den f.eks. kun tager imod tal eller datoer. Overholdes input typen ikke, gives der besked om dette, og man kan ikke gemme ændringerne før de alle sammen er overholdt. Alle Html helperne med vareinformationer i er i en Html form, der sender informationen videre til UpdateItem funktionen i EditItem controlleren.

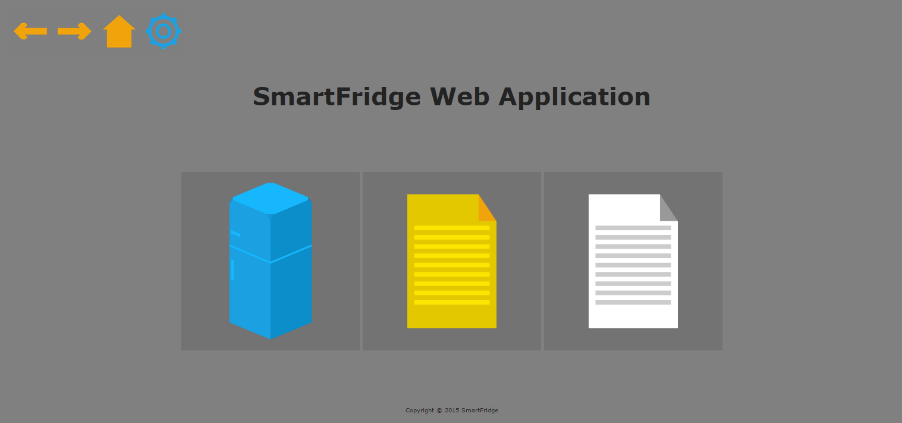
|  |
| --- |
| @{  ViewBag.Title = "EditItem";  Layout = "~/Views/Shared/\_Layout.cshtml";  GUIItem dataSource = ViewData["olditem"] as GUIItem;  EditItemController.selectedUnit(dataSource);  } |

**Kodestump 12** Her ses, hvordan den valgte items data indlæses til brug i viewed

|  |
| --- |
| @Html.TextBox("Type", dataSource.Type, new  {  @id = "VaretypeText"  })  @Html.DropDownList("types", (List<SelectListItem>)ViewBag.types , new  {  @id = "VaretypeDropDown"  })  [...]  @Html.TextBox("Amount", dataSource.Amount, new { @type = "number", @min = 0 })  [...]  @Html.TextBox("Shelflife", dataSource.ShelfLife.ToString("dd-mm-yyyy"), new{@type = "date", @autocomplete = "on"}) |

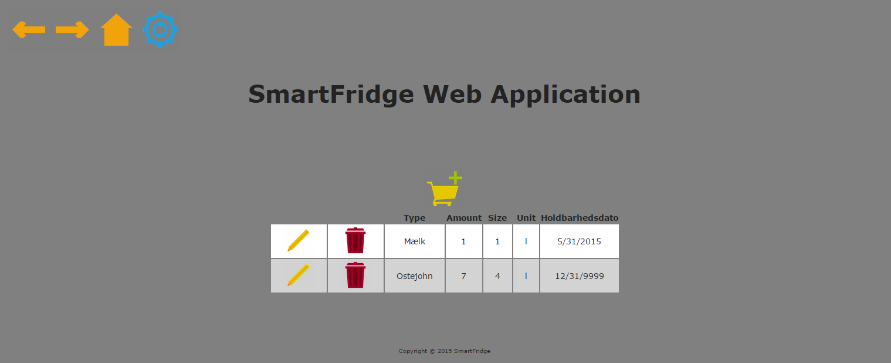
**Kodestump 13** Eksempler på forskellige typer af Html helpers.

##### Index

Index viewed der svarer til hovedmenuen fra Fridge-app er det simpleste af alle web applikationens views og består kun af tre knapper. Som det ses på **Figur 6**, ligner de to også hinanden rigtigt meget.

**Figur 7** Index view web applikationens svar på en hovedmenu

##### ListView

ListView’et indeholder en ’Tilføj Item’-knap til at navigere brugeren hen til AddItemView. Denne knap er symboliseret som en indkøbsvogn og et plus. Ydermere indeholder ListView en tabel med de items der er på den pågældende liste. Hvert item har to knapper ud fra sig. Et ’EditItem’-knap symboliseret af en blyant og en ’DeleteItem’-knap symboliseret af en skraldespand. Disse knapper bliver tilføjet når griddet laves. På Kodestump 15 ses det hvordan knappen er tilføjet til griddet. Cols, som griddet hedder, bliver lavet hver gang viewet bliver renderet. For hvert item i model listen bliver der lavet syv nye columns. De første to collumns er redigér og slet knapperne. Herefter bliver hver attribut tilføjet, som det ses på Kodestump 15 hvor Type bliver tilføjet.

Figur – ListView’s View

Efter cols er blevet oprettet, sørger @grid.GetHtml for at rendere og style det på viewet.

|  |
| --- |
| […]  IEnumerable<WebGridColumn> cols =  new List<WebGridColumn> {  new WebGridColumn{  Format =  @<text>  <a href="@Url.Action("ToEditItem", "LisView", new GUIItem(item.Type, item.Amount, item.Size, item.Unit))"><img src="~/Images/Edit.png" class="RowImg" /></a>  </text>  […]  new WebGridColumn{  Header="Type",  Format = @<text>  <span class="display">  <label id="lblType">@item.Type</label>  </span>  </text>  },  […]  <div align="center">  @grid.GetHtml(columns:cols,  rowStyle:"oddRow", alternatingRowStyle:"evenRow")  </div> |

Kodestump – Udsnit af ListView.cshtml

#### Models

Modellerne brugt i web-appen, er de samme klasser, der blev brugt i Fridge-app med undtagelse af GUIItemList. Der en model for hver type database entitet, der gøres brug af ved al database tilgang, og GUIItem klassen, som er den lokale repræsentation af en vare og derfor bruges i forbindelse med samtlige views. Modellerne er implementeret i et klassebibliotek for sig, da data access layered, der også ligger i et klasse bibliotek for sig, har behov for at kende til disse. Ellers er praksissen med at ligge modeller i et klasse bibliotek normalt kun noget, der gøres brug af i projekter med mange modeller.

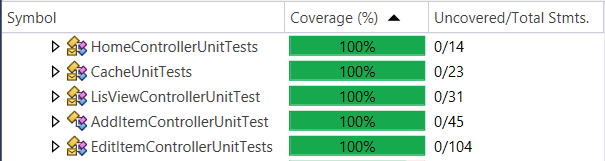
#### Cache

Til wep-appen er der blevet implementeret en cache klasse, der indeholder de relevante ting for de forskellige views. Alle cache klassens attributter er static . Især med facaden til data acces layered er det vigtigt at det er static, da flere instanser af denne kunne resultere i fejl, hvis flere instanser samtidigt forsøger at tilgå databasen. Som det fremgår af **Kodestump 16** CurrentList propertyens set metode, sørger for også at hente alle ListItems, der figurere på den nuværende liste og alle Items ned. Her hentes også samtlige items fra databasen ned. Grunden til at alle items hentes ned er at deres navne bruges som info i dropdown menuer i både EditItem og AddItem viewene, hvor det altså er relevant at få alle varetyper med, og ikke blot dem, der er tilknyttet ListItems på den nuværende liste

|  |
| --- |
| public static List CurrentList { get { return \_currentList; }  set  {  \_currentList = value;  var uow = DalFacade.GetUnitOfWork();  CurrentListItems = new List<ListItem>();  var tempList = uow.ListItemRepo.GetAll().ToList();  if (tempList.Any())  {  foreach (var Listitem in tempList)  {  if (Listitem.ListId == \_currentList.ListId)  {  CurrentListItems.Add(Listitem);  }  }  }  DbItems = uow.ItemRepo.GetAll().ToList();  DalFacade.DisposeUnitOfWork();  }  } |

**Kodestump 15** CurrentList propertyen, hvis set metode også henter alle relevante   
ListItems på listen ned

### Test

Som nævnt i rapporten er en af grundende til valget af MVC mønstret, at det er nemt at teste. Dette har vist sig ikke at være tilfældet, men det er alligevel lykkedes at opnå en coverage på 100% på controllerne og Chache klassen i web applikationen. Figur 9.

Figur 9 – Coverage rapport for controllers og Chache

#### AdditemController test

Til at teste AdditemController klasssen er det oprettet 7 tests.

Setup funktionen sørger for at der bliver oprettet et nye objekter inden hver funktion bliver kørt. Nogle af disse objekter er; item, dalfacade, listitem og flere, som ses på Kodestump 16.

|  |
| --- |
| [SetUp]  public void Setup()  {  var dalfacade = Substitute.For<ISmartFridgeDALFacade>();  Cache.DalFacade = dalfacade;  listitem = new ListItem(1, 1, "l", default(DateTime), currentList, item) { ItemId = item.ItemId };  dalfacade.GetUnitOfWork().ListRepo.GetAll().Returns(new List<List>() {currentList});  dalfacade.GetUnitOfWork().ItemRepo.GetAll().Returns(new List<Item> {item});  dalfacade.GetUnitOfWork().ListItemRepo.GetAll().Returns(new List<ListItem>() {listitem});  } |

Kodestump - setup til additem

Den første test vist på Kodestump 16 tester på AddItem funktionen. Der oprettes et AddItemController objekt ved navn uut. Den kalder funktionen AddItem og konvertere det returnerede ActionResult, fra funktionen, til et ViewResult, som ligges ned i en variabel ved navn actResult. Herefter assertes der på om det returnerede actResult. Problemet ved denne test er at der ligger funktionalitet inde i funktionen, som ikke kan assertes på, da disse attributter er private. I denne test tjekkes der på om navnet på viewet er en tom string, hvilket den altid vil være når en funktion returnere ’return View(model); ’ kan den kun returnere navnet på viewet eller en tom string.

|  |
| --- |
| [Test]  public void AddItem\_TestLoadingCorrectModel\_ReturnCorrectModel()  {  var actResult = uut.AddItem() as ViewResult;  Assert.That(actResult.ViewName, Is.EqualTo(""));  } |

Kodestump – AddItem test

De næste 5 funktioner tester addNewItem og disse tests er designet til at nå al koden i funktionen.

Funktionen, som ses på Kodestump 17, tilføjer et item og ender med at kalde en anden funktion i ListView. Derfor assertes der på om den rigtige funktion er blevet kaldet. Denne test giver derfor et mere sigende resultat sammenlignet med testen på Kodestump 16.

|  |
| --- |
| [Test]  public void addNewItem\_Add1ItemAndExit\_LisViewReturned()  {  RedirectToRouteResult result = uut.addNewItem("TestItem", "1", "1", "l", "", "Exit") as RedirectToRouteResult;  Assert.That(result.RouteValues["action"], Is.EqualTo("ListView"));  } |

Kodestump 18 – AddNewItem Test

#### HomeController test

HomeController var den nemmeste at teste, da den kun består af to metoder. Den første metode, Index, opretter Chache og returnere det View der skal ses på forsiden. Der testes på om dette view returneres.

Den anden funktion skal kalde ListView funktionen i LisView Controlleren og sende en string med den rigtige liste med over. På Kodestump 19 ses der hvordan der tjekkes på at den rigtige funktion bliver kaldt i LisView.

|  |
| --- |
| [Test]  public void TestSetCurrentListReturnsTheRigtView()  {  string currentlist = "TestList";  var actResult = uow.SetCurrentList(currentlist) as RedirectToRouteResult;  Assert.That(actResult.RouteValues["action"], Is.EqualTo("ListView"));  } |

Kodestump - HomeController test

#### LisViewController test

Ligesom i AddItemController og HomeController, blev der tjekket på den funktion der skal returnere viewet, returnere det rigtige view. Ydermere er der tre tests som alle har til formål at teste på om den rigtige metode bliver kaldt når henholdsvis ItemToEdit og DeleteSelectedItem bliver kaldt. De to tests der tjekker på ItemToEdit metoden skal kalde EditItem metoden i EditItem Viewet. Og den test der tjekker på om DeleteSelectedItem bliver kaldt skal returnere samme view, men med en ændret model. Problemet disse tests er de samme som i AddItemController. Der bliver kun testet på slutresultatet og ikke den manipulation af det medsendte item.

#### EditItemController test

De tests der er lavet til til EditItemController har samme problem som beskrevet i AdditemCOntroller og LisviewController. Den største forskel, som set på Kodestump 20 er at for at få en højere coverage % skal nogle ting initialiseres.

|  |
| --- |
| [Test]  public void UpdateItem\_Item\_EverythingButTypeAndNoShelfLifeUpdated()  {  uut.EditItem(new GUIItem("test", 1, 1, "l"));  FormCollection collection = new FormCollection();  collection.Add("Type","test");  collection.Add("Amount", "3");  collection.Add("Volume", "2");  collection.Add("units", "kg");  collection.Add("date", "");  var actResult = uut.UpdateItem(collection) as RedirectToRouteResult;  Assert.That(actResult.RouteValues["action"], Is.EqualTo("ListView"));  } |

Kodestump – Test til EditItemController

#### Cache test

ChacheUnitTests klassen indeholder tre tests og en setup.

Disse tre test har alle til formål at tjekke på om den data der ligger i Chache. På Kodestump 21 ses det at Chacen’s currentList bliver sat til currentList, som indeholder data hentet vha. dalfacade i Setup funktionen. Herefter tjekkes der på om det rigtige listitem ligger i denne liste. Listitem initialiseres i Setup funktionen.

|  |
| --- |
| [Test]  public void CurrentListSet\_ContainsListItem()  {  Cache.CurrentList = currentList;  Assert.That(Cache.CurrentListItems.Contains(listitem));  } |

Kodestump – Chache test

De andre er opbygget på samme simple måde, men der assertes på hvad der ligger i de forskellige lister i klassen.