#### Business Logic Layer

##### Constructor

I constructoren, der kan ses på oprettes aller først den AdoNetContext som skal bruges til at oprette forbindelse til den lokale database med. Herefter oprettes de tre repositories, som skal bruges til at tilgå databasens forskelllige tabeller.

|  |
| --- |
| public BLL()  {  Context = new AdoNetContext(\_connectionFactory);  \_itemRepository = new ItemRepository(Context);  \_listRepository = new ListRepository(Context);  \_listItemRepository = new ListItemRepository(Context);  } |

**Kodestump 1** Business Logic Layer constructoren

##### WatchItems

WatchItems er en property, der bruges til at hente en liste med alle de relevante GUIItems for en given liste til GUIen. Det første den gør er at indlæse hele databasens indhold med LoadFromDB(); funktionen, der er beskrevet senere i dette afsnit. Herefter finder den ud af om listen den skal fylde i eksisterede på i databasen. Hvis ikke listen eksisterede i forvejen oprettes den, dette kan ses på **Kodestump 2**.

|  |
| --- |
| List<string> temp = new List<string>();  foreach (var list in Lists)  {  temp.Add(list.Name);  }  if (!temp.Contains(CurrentList))  {  CreateList();  } |

**Kodestump 2** Lists er en property,   
 der indeholder listerne.

Efter det findes alle de ListItems i databasen, der skal figurerer på den nuværende liste. For hvert ListItem der bliver fundet oprettes et tilsvarende GUIItem, hvis type er givet af den Item, det pågældende ListItem er koblet sammen med. Hele processen kan ses i **Kodestump 3**. Alle disse GUIItems ligges i en liste, som til sidst bliver returneret.

|  |
| --- |
| foreach (var dbListItem in \_dblistItems)  {  if (dbListItem.List.ListName == CurrentList)  {  foreach (var dbItem in \_dbItems)  {  if (dbListItem.Item.ItemId == dbItem.ItemId)  {  GUIItem guiItem = new GUIItem();  guiItem.Type = dbItem.ItemName;  guiItem.Amount = (uint)dbListItem.Amount;  guiItem.Unit = dbListItem.Unit;  guiItem.Size = (uint)dbListItem.Volume;  guiItem.ShelfLife = dbListItem.ShelfLife;  guiItems.Add(guiItem);  }  }  }  } |

**Kodestump 3** Her ses, hvordan alle ListItems, der skal figurerer på   
 den nuværende liste, kobles sammen med deres respektive item i   
 et GUIItem.

##### LoadFromDB

LoadFromDB indlæser alt fra databasen, og kommet indholdet i lokale objekter. Som det kan ses på **Kodestump 4**, gøres dette ved at kalde funktionen GetAll(); på samtlige repositories. Da denne funktion returnere en IEnumerable kaldes ToList(); for at lave dem til lister. Når alt ligger lokalt kaldes Mapper(); funktionen, der sørger for at de lokale referencer er som de skal være. Listerne fra databasen bruges til at oprette lokale GUIItemList objekter, der til forskel fra en List fra databasen har en indbygget liste af GUIItems.

|  |
| --- |
| List<List> lists = new List<List>();  using (var uow = Context.CreateUnitOfWork())  {  lists = \_listRepository.GetAll().ToList();  \_dbItems = \_itemRepository.GetAll().ToList();  \_dblistItems = \_listItemRepository.GetAll().ToList();  \_listItemRepository.Mapper(\_dbItems, lists, \_dblistItems);  } |

**Kodestump 4 Her indlæses databasens inhold.**

Når alt ligger lokalt og alle GUIItemListerne er oprettet sammenkobles alle ListItems med deres respektive Item, i et GUIItem. Alle disse GUIItems ligges i deres tilhørende lokale GUIItemList, som det kan ses på **Kodestump 5**

|  |
| --- |
| foreach (var list in Lists)  {  foreach (var listItem in \_dblistItems)  {  if (listItem.List.ListId == list.ID)  foreach (var dbItem in \_dbItems)  {  if (listItem.Item.ItemId == dbItem.ItemId)  {  GUIItem guiItem = new GUIItem()  {  Amount = (uint)listItem.Amount,  Type = dbItem.ItemName,  Size = (uint)listItem.Volume,  Unit = listItem.Unit,  ID = listItem.ItemId,  ShelfLife = listItem.ShelfLife  };  list.ItemList.Add(guiItem);  break;  }  }  }  } |

**Kodestump 5 Her ligge alle ListItems og Items sammen til GUIItems,  
 der fordeles ud på deres respektive lokale lister.**

##### CheckShelfLife

CheckShelfLife funktionen går alle items på en specifik liste igennem for at finde ud af, hvorvidt nogle af disse har overskredet deres sidste holdbarhedsdato. Den gør dette ved at sammenligne hvert GUIItems ShelfLife property, med dagens dato. For hvert GUIItem, der er blevet for gammel, tilføjes en notifikation til en liste, der til sidst returneres som, det kan ses på **Kodestump 6**.

|  |
| --- |
| var list = new List<Notification>();  foreach (var item in items.ItemList)  {  if (item.ShelfLife != null)  {  if (item.ShelfLife.Date <= DateTime.Now)  {  string message = item.Type + " blev for gammel d. " + DateTime.Now.Date;  list.Add(new Notification(message, DateTime.Now, item.ID));  }  }  }  return list; |

**Kodestump 6** Her kan ses hvordan en notifikation med information omkring hvilken vare,  
 der blevet for gammel, tilføjes for hver vare hvis holdbarhedsdato er overskreden.

##### AddItemsToTable

AddItemsToTable undersøger først, hvorvidt den liste funktionen blev fra eksistere, hvis den ikke før det bruges CreateList. Når listen eksistere findes den rigtige lokale liste dens id skal bruges når der skal tilføjes ListItems til databasen. Herefter løbes alle de GUIItems, der skal tilføjes igennem for at finde ud af om der er nogle af dem, som er af ikke i forvejen eksisterende type. Dette kan ses på **Kodestump 7**, her bruges IsNewItem til at finde ud af, hvorvidt et GUIItem er af en ny type eller ej. Bliver der tilføjet nye Items, kaldes LoadFromDB så de nye Items også ligger lokalt.

|  |
| --- |
| bool newItemAdded = false;  using (var uow = Context.CreateUnitOfWork())  {  foreach (var newGuiItem in newGuiItems)  {  if (IsNewItem(newGuiItem))  {  Item dbItem = new Item()  {  ItemName = newGuiItem.Type,  StdUnit = newGuiItem.Unit,  StdVolume = (int)newGuiItem.Size  };  \_itemRepository.Insert(dbItem);  newItemAdded = true;  }  }  uow.SaveChanges();  }  if (newItemAdded)  LoadFromDB(); |

**Kodestump 7 Her tilføjes alle nye typer af Items,   
 hvorefter der loades fra databasen igen.**

Når de nyeste opdateringer er hentet ned, tjekkes der på hver af de nye GUIItems om den kaldende liste allerede har et GUIItem af samme type. Findes der ikke et match, laves der et ListItem ud fra det nye GUIItems attributter, og det tilføjes til databasen. Eksistere der allerede et GUIItem af samme type sammenlignes der på deres størrelse/volume, enhed og udløbsdato, er der blot en af disse, som ikke stemmer overens med det i forvejen eksisterende tilføjes der et nyt ListItem. Stemmer alle parametrene overens laves der et nyt ListItem, hvor antallet er summen af det gamle og det nye item. Herefter slettes det gamle ListItem, og det nye indsættes i databasen, hvilket kan ses på **Kodestump 8**.

|  |
| --- |
| [...]  if (dbItem.ItemName == newGuiItem.Type)  {  foreach (var dbListItem in \_dblistItems)  {  if (dbItem.ItemId == dbListItem.Item.ItemId &&  dbListItem.List.ListId == currentGuiItemList.ID &&  dbListItem.Unit == newGuiItem.Unit &&  dbListItem.Volume == newGuiItem.Size)  {  int currentAmount = dbListItem.Amount;  ListItem updatedListItem = new ListItem(((int)newGuiItem.Amount +  currentAmount),  (int)newGuiItem.Size,  newGuiItem.Unit,  dbListItem.List,  dbItem,  newGuiItem.ShelfLife);  \_listItemRepository.Delete(dbListItem);  \_listItemRepository.Insert(updatedListItem);  }  [...] |

**Kodestump 8 Her kan ses hvordan det håndteres når der bliver tilføjet et allerede eksisterende  
 GUIItem**

Når alle nye GUIItems er tilføjet til databasen på den ene eller den anden måde kaldes SaveChanges(); så ændringerne gemmes i databasen, og LoadFromDB(); kaldes for at få alle disse ændringer gemt lokalt også. Når det nyeste ligger lokalt kaldes Mapper(); der sikrer at alle referencer er som de skal være. Til sidst kaldes STDToShopListControl, som er beskrevet længere nede i dette afsnit.

##### DeleteItem

DeleteItem finder først den rigtige lokale liste, da listens id skal bruges til at finde den ListItem, der skal slettes fra databasen. Bagefter som vist på **Kodestump 9** findes og slettes det fundne ListItem. Når det er gjort laves et kald til uow.SaveChanges, så ændringen også sker i databasen.

|  |
| --- |
| using (var uow = Context.CreateUnitOfWork())  {  foreach (var dbListItem in \_dblistItems)  {  if (dbListItem.Item.ItemName == GUIitemToDelete.Type  && dbListItem.Amount == GUIitemToDelete.Amount  && dbListItem.Unit == GUIitemToDelete.Unit  && (uint)dbListItem.Volume == GUIitemToDelete.Size  && dbListItem.ListId == currentGuiItemList.ID)  {  \_listItemRepository.Delete(dbListItem);  break;  }  }  uow.SaveChanges();  } |

**Kodestump 9 Her kan ses hvordan et ListItem slettes fra databasen.**

##### ChangeItem

ChangeItem starte ligesom DeleteItem med først at finde den rigtige lokale liste da dennes id skal bruges til at finde det rigtige ListItem. Herefter tjekker den på om det kun er det fundne ListItem, der skal opdateres eller om den tilhørende Item også skal opdateres. Hvis Itemet skal opdateres gøres det som det kan ses på **Kodestump 10**.

|  |
| --- |
| [...]  if (oldItem.Type != newItem.Type)  {  foreach (var dbItem in \_dbItems)  {  if (dbItem.ItemName == oldItem.Type)  {  dbItem.ItemName = newItem.Type;  \_itemRepository.Update(dbItem);  }  }  }  [...] |

**Kodestump 10 Her opdateres et ListItems   
 tilhørende item.**

Når det rigtige ListItem er fundet, bliver dette opdateret ved at det allerede eksisterende ListItem slettes, og der indsættes et nyt med de opdaterede værdier som det ses på **Kodestump 11**.

|  |
| --- |
| [...]  ListItem updatedListItem = new ListItem((int)newItem.Amount,  (int)newItem.Size,  newItem.Unit,  dbListItem.List,  dbListItem.Item,  newItem.ShelfLife);  \_listItemRepository.Delete(dbListItem);  \_listItemRepository.Insert(updatedListItem);  uow.SaveChanges();  break;  [...] |

**Kodestump 11 Her "opdateres" et ListItem ved først at slette det gamle  
 ListItem, og derefter indsætte et nyt med opdaterede værdier.**

##### STDToShopListControl

STDToShopListControl tjekker først på om den er blevet kaldt fra "Standard-beholdning" listen, da det kun er ved kalde derfra den skal gøre noget. Derfor returnere den bare, hvis kaldet kom fra en af de to andre liste. På **Kodestump 12** kan det ses, hvordan den opretter to lister af ListItems med varer der henholdsvis er i køleskabet, eller som Bruger har valgt altid skal være i køleskabet.

|  |
| --- |
| foreach (var dbListItem in \_dblistItems)  {  if (dbListItem.List.ListName == "Køleskab")  {  har.Add(dbListItem);  }  else if (dbListItem.List.ListName == "Standard-beholdning")  {  skalAltidHave.Add(dbListItem);  }  } |

**Kodestump 12 Her kan ses, hvorledes der oprettes en liste over ting, der er  
 i køleskabet, og ting der skal være i køleskabet.**

Listen over ting, der altid skal være i køleskabet kopieres over i en ny liste ved navn "mangler", og der kigges nu efter overenstemmelser mellem de to tidligere lister. Findes der et ListItem i køleskabet, hvis amount er større end eller lig med det fra standard-beholdningen, fjernes det fra mangler listen, ellers, hvis antallet er under bliver differensen mellem disse, antallet på "mangler" listen. På **Kodestump 13** kan ses, hvordan dette er implementeret. Her efter løbes "mangler" listen igennem og samtlige ListItems kobles sammen deres tilhørende Item i et GUIItem, som tilføjes til en ny liste. Denne liste tilføjes til "Inkøbsliste" med AddItemsToTable funktionen.

|  |
| --- |
| List<ListItem> mangler = new List<ListItem>(skalAltidHave);  foreach (var STDListItem in skalAltidHave)  {  foreach (var ownedListItem in har)  {  if (ownedListItem.Item.ItemName == STDListItem.Item.ItemName &&  ownedListItem.Unit == STDListItem.Unit &&  ownedListItem.Volume == STDListItem.Volume)  {  if (ownedListItem.Amount >= STDListItem.Amount)  {  mangler.Remove(STDListItem);  }  if (ownedListItem.Amount <= STDListItem.Amount)  {  STDListItem.Amount -= ownedListItem.Amount;  }  }  }  } |

**Kodestump 13 Her fjernes alle vare som allerede er i køleskabet i et korrekt antal fra  
 listen over varer, der mangler. Varer, hvor antallet er forkert, bliver stående med det  
 manglende antal.**