

Projektuppgift

DT071G – Programmering i C# .NET

MyPlanner

Todo "Att-göra-post"

Hanin Farhan



Mittuniversitetet
MID SWEDEN UNIVERSITY

MITTUNIVERSITETET
Avdelningen för informationssystem och -teknologi

Författare: Hanin Farhan, hafa2300@student.miun.se
Utbildningsprogram: Webbutveckling, 120 hp
Huvudområde: Datateknik
Termin 1, år: 2 2024

Sammanfattning

Projektet går ut på att skapa en "Att-göra-post" webbapplikation. Målet är att visa grundläggande kunskaper inom C# och tillämpa i en interaktiv applikation. Projektet är uppdelad i två delar, databasen SQLite för lagring av To-do posts och en serverbaserad Blazor app, som kopplas till databasen och skapar det interaktiva användargränssnittet. Applikationen körs lokalt och användaren har möjlighet att lägga till, uppdatera, radera och markera To-do posts.

Nyckelord: C#, SQLite, "Att-göra" To-do posts, Blazor

1 Innehållsförteckning

| | |
|---|------------|
| Sammanfattning | iii |
| 1 Innehållsförteckning..... | iv |
| 2 Terminologi | vi |
| 2.1 Akronymmer..... | vi |
| 3 Introduktion..... | 1 |
| 3.1 Bakgrund och problemmotivering..... | 1 |
| 3.2 Detaljerad problemformulering..... | 1 |
| 3.3 Avgränsningar | 1 |
| 4 Teori..... | 2 |
| 4.1 C# | 2 |
| 4.1.1 Objekt orientering..... | 2 |
| 4.1.2 Method | 2 |
| 4.1.3 Class..... | 3 |
| 4.1.4 Constructor..... | 3 |
| 4.1.5 Get; Set; | 3 |
| 4.1.6 List..... | 3 |
| 4.2 SQLite..... | 3 |
| 4.3 Entity Framework Core | 4 |
| 4.3.1 Initial migration..... | 4 |
| 4.3.2 CRUD operationer..... | 4 |
| 4.4 Blazor..... | 5 |
| 4.5 Razor | 5 |
| 4.5.1 Bindings..... | 5 |
| 5 Metod..... | 6 |
| 6 Konstruktion..... | 7 |
| 6.1 ER-Diagram | 7 |
| 6.2 Flödesschema över applikationen..... | 8 |
| 6.3 Todo klass | 8 |
| 6.4 TodoContext klass..... | 10 |
| 6.5 Initial migration –Todos tabell SQLite..... | 11 |
| 6.6 Blazor..... | 11 |
| 6.6.1 Felhantering..... | 13 |
| 6.6.2 Styling..... | 14 |
| 7 Resultat..... | 15 |

MyPlanner – Todo lista

Hanin Farhan

2024-10-29

| | |
|---------------------------------------|-----------|
| 8 Slutsatser..... | 16 |
| Källförteckning | 17 |
| Bilaga A: ER-diagram..... | 19 |
| Bilaga B: Flödesscheman | 20 |
| Bilaga C: Wireframes | 23 |
| Bilaga D: Grafisk profil | 24 |
| Bilaga E: Databas SQLite | 25 |

2 Terminologi

2.1 Akronymer

| | |
|-------------|--------------------------------------|
| C# | Objektorienterad programmeringsspråk |
| Method | Funktion |
| Class | Klass |
| Constructor | Konstruktör |
| To-do | Att-göra-post |
| get; set; | Egenskaper att hämta och ställa in |
| SQLite | Relationsbaserad databas |
| CRUD | Skapa, Läs, Uppdatera, Ta bort |
| Blazor | C# ramverk |

3 Introduktion

3.1 Bakgrund och problemmotivering

Målet med projektet är att skapa en valfri applikation med språket C#. Projektet är en Blazor applikation som heter MyPlanner och går ut på att skapa en att göra lista (To-do). Med hjälp av databaser SQLite är det möjligt att lägga till ny todo post, uppdatera, radera och markera existerande To-do poster. Förutom C# funktionaliteter kommer webbsidan även innehålla CSS styling och inbyggda verktyg från Blazor.

3.2 Detaljerad problemformulering

Krav punkter som ska utföras enligt följande:

- Wireframe/Designskisser för webbsidans utseende och layout.
- Passande favicon och logotyp
- ER-diagram över vad som ska lagras i To-do listan
- Skapa Flödesschema
- Skapa SQLite databas
- Skapa nytt Blazor projekt
- Koppla Blazor till databasen
- Utföra CRUD operationer i Blazor
- Skapa html struktur för sidan med formulär, knappar och To-do poster.
- Binda html element med C# funktionaliteter
- Lägga till felhantering för tomma input
- Styla element med CSS enligt designskisserna

3.3 Avgränsningar

Eftersom fokuset i kursen är språket C# kommer rapporten inte ta upp hur html och CSS fungerar. Dock kommer rapporten ta upp hur html elementen binds ihop med C# för att skapa en interaktiv sida med funktionaliteter.

4 Teori

4.1 C#

C# är ett programmeringsspråk som är en del av ramverket .NET och är utvecklat av Microsoft. .NET ramverk består av ett stort klassbibliotek som innehåller funktioner som kan anropas och användas i C#. Därmed innehåller biblioteket inbyggda verktyg som förenklar kodarbetet C#. [1]

Precis som andra språk innehåller C# även datatyper där de vanligaste är bland annat integer, string, boolean och char. C# använder även logiska operatorer för att sätta villkor såsom && som betyder **och**, || betyder **eller** och ! betyder **inte**. Användning av operatorerna tillsammans med if/else satser kör olika kodblock om villkoret är sant eller falskt och vi får därmed olika resultat beroende på exekverat kod. [1]

4.1.1 Objekt orientering

C# är ett objektorienterat programmeringsspråk, dvs det är möjligt att definiera klasser, kapsla in funktionaliteter i ett objekt samt ärva funktionaliteter och egenskaper. På så sätt kan koden struktureras och brytas ner till mindre återanvändbara delar att utnyttja i C# applikationen. [2]

4.1.2 Method

C# använder **Method eller metoder** och fungerar precis som funktioner i andra språk. Den definieras och innehåller ett kodblock att utföra olika åtgärder. Metod kan även returnera ett värde som argument om värdet ska användas vidare, det görs med hjälp av att en parameter skickas med. [1]

Metoder och variabler i en klass kan nås med olika åtkomst som public, protected, private och internal. Public kan nås av alla medan private endast kan nås av egna objektet. [3]

Nedan visar exempel på method struktur som inte returnerar värde:

```
public async Task ShowTodoList ()  
{  
    //Kodblock  
}
```

Public beskriver att den är publik och tillgänglig att anropa utanför sin klass. Async beskriver att funktionen är asynkron och antagligen hämtar data som tar tid. Task är ett tomt asynkron värde och returnerar inget. Innanför måsvingarna placeras kod som ska exekveras. [4]

Method anropas genom att skriva funktionens namn följt av parentes som exempelvis: ShowTodoList(); [1]

4.1.3 Class

Klasser används som en mall för att skapa ett eller flera objekt. Den innehåller medlemsvariabler, methods och egenskaper. Objekt skapas som ny instans i klassen, de skapas som kopior i klassen. En klass beskriver objekttypen medan objekt är en instans av klassen. Fördelen med att använda klasser är att koden kan återanvändas. Klassen kan ärvas med data och egenskaper och användas i andra klasser. Dessutom blir koden även lättare att underhålla då ändringar endast behöver göras på en plats. [5]

4.1.4 Constructor

Constructor eller Konstruktör är en metod som kallas när en ny instans skapas i klassen. Konstruktorn har samma namn som klassen och har inget returvärde. Den säkerställer att startvärden är tilldelade till klassens variabler när en ny initiering startas. [5]

4.1.5 Get; Set;

För att komma åt variabler och deras egenskaper som är privata i en klass används get; och set;. Dem fungerar som ett slags inkapsling och kontroll över hur variablerna kan läsas och skrivas ut utanför den privata klassen. Get används för att returnera variabelns värde och set; för att skriva ett nytt värde med validering innan den tilldelas till variabeln. På så sätt förblir de ursprungliga variabler skyddade och säkra. [3]

Följande exempel visar hur vi kommer åt en privat sträng:

```
private string title;
```

```
public string? Title { get { return title; } set { title = value; } }
```

get returnerar strängen title och title sätts till ett värde som anges som input. Nu kan nya värden sättas på title utan att komma åt den ursprungliga title.

4.1.6 List

List <T> är en generisk klass och lagrar objekt i en lista lite likt syfte med array. Men List<T> gör det möjligt att lagra objekt i valfri typ <T>. List<T> rekommenderas när man vill dynamisk kunna lägga till och radera data från listan och till skillnad från array så behöver man inte definiera storlek på listan från början. [6, s.146, s.249]

4.2 SQLite

SQLite är en liten men snabb databas som bygger på ett relationsbaserat hanteringssystem. Det är den mest använda databasmotorn i världen. SQLite lagrar hela databasen i en fil och är därmed lämplig att använda för inbyggda system, mobilapplikationer och mindre applikationer. [6, s. 516-517]

4.3 Entity Framework Core

Entity Framework Core är ett slags verktyg och ramverk som lagrar .NET-objekt i olika typer av databaser utan behov av att skriva mycket kod eller någon databas-kod. Den förenklar arbetet, kontrollerar åtkomst till databaser och gör det möjligt att lagra data i databas smidigare utan att skriva SQL frågor i C# koden då den genererar SQL i bakgrunden. Den sköter därmed kopplingen, kommandon och interaktionen med databasen. För att kunna använda verktyget behöver man lägga till följande paket i projektet [7]:

```
dotnet add package Microsoft.EntityFrameworkCore
```

4.3.1 Initial migration

Initiala migration syftar på att skapa en databas för första gången. Entity Framework Core kommer att skapa en struktur för databasen, generera den SQL kod som behövs för att skapa den och sen köra den. Följande kommandon behövs för att skapa databas, lägga till data och uppdatera senaste förändringar i C# [8]:

```
dotnet ef migrations add InitialCreate
```

```
//Migrering dvs uppsättningar av förändringar som skapar och kontrollerar ändringar i strukturen och sätter upp databasschemat enligt angiven modellklass. [8]
```

```
dotnet ef migrations add SeedData
```

```
//Migrering, som lägger till data i databsen [8]
```

```
dotnet ef database update
```

```
//Tillämpar, uppdaterar och skapar databas med tabell utifrån modellklassen och dess ändringar. [8]
```

4.3.2 CRUD operationer

För att kunna dynamiskt visa existerande data, lägga till, uppdatera och radera data från databasen, används CRUD operationer. Det är en förkortning på Create, Read, Update och Delete. Med hjälp av Entity Framework blir det möjligt att kalla på inbyggda metoder för att utföra operationerna. Exempel på inbyggda operationer: **.Add()** för att lägga till data, **.Update()** för att uppdatera data och **.Remove()** för att ta bort data. [9]

4.4 Blazor

Blazor är ett .NET frontend ramverk som använder C# som språk. Det är komponentbaserad där varje komponent byggs upp och innehåller funktionaliteter. Med hjälp av Blazor är möjligt att bygga upp ett fullständig UI och interaktion som stödjer både HTML och CSS. Blazor applikation kan skapas som klientapplikation eller även inkludera server-sida för att bygga upp ett full-stack applikation. [10]

4.5 Razor

Komponenter skrivs med ändelsen .razor i Blazor och är syntax som kombinerar HTML och C# kod. Det är även möjligt att separera C# kod från HTML och lägga till en separat CSS fil som isolerar och avgränsar till razor komponent. [10]

4.5.1 Bindings

Razor komponenter tillför databindnings funktionalitet och binder gränssnittets HTML element med C# kod. På så sätt kan komponenter reagera på ändringar, lagra information och uppdatera funktionaliteter. Det underlättar utvecklingen av att göra webbapplikationen interaktiv för användaren. [11]

5 Metod

Följande verktyg och metoder kommer användas för att lösa tidigare nämnda kravpunkter:

- ER-Diagram för todo poster skapas i Draw.io som mall för ny databas tabell i SQLite.
- Wireframe/designskiss skapas i programmet Figma och visar layout, tema och gränssnitt för webbsidan MyPlanner.
- Logotypen och faviconen skapas i illustrator och exporteras i lämpliga format.
- Databasen skapas i SQLite med hjälp av DB Browser desktop app.
- Flödesschemat skapas i Microsoft Visio.
- Blazor projektet skapas i Visual Studio 2022.
- Konfiguration och hämtning av data mellan Blazor och SQLite hanteras med hjälp av Entity Framework Core.
- Koppling mellan databasen och blazor görs med hjälp av klassen DbContext och modellklass.
- Varje ny todo post definieras med hjälp av get; och set; i klassen Todo.
- CRUD operationerna utförs i en Home.razor.cs fil kopplad till Home.razor och visar webbsidans innehåll som användaren ser på sidan.
- Formulär, knappar, todo poster och sidans layout skapas i filen Home.razor och innehåller bland annat sidans html innehåll.
- Home.razor.cs kod kopplas till Home.razor med hjälp av bindings, if/else-satser och foreach.
- Felhantering i Home.razor ska säkerställa att tomma fält, som kan orsaka null-värden, inte skickas vidare till databasen.
- Styling av Home.razor kommer ske i Home.razor.css fil för att isolera stylumen. Layouten, färg och tema utgår ifrån designskissen.

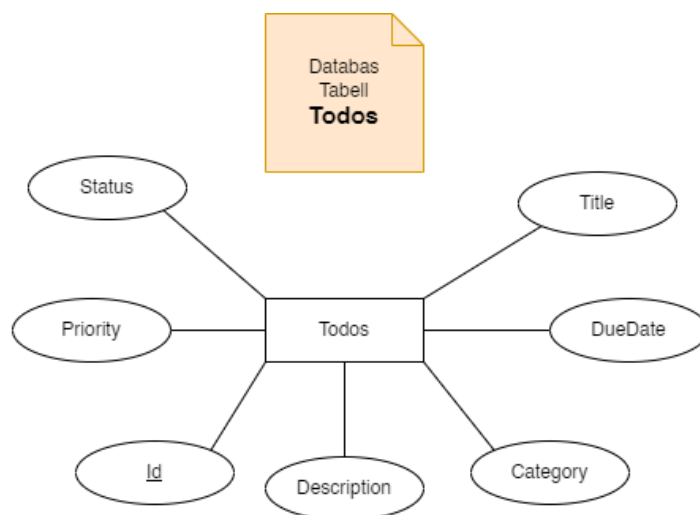
6 Konstruktion

Konstruktionen kommer gå igenom följande huvudpunkter för applikationen:

- ER-diagram
- Flödesschema
- To-do lagring
- Databasanslutning
- CRUD operationer
- Blazor

6.1 ER-Diagram

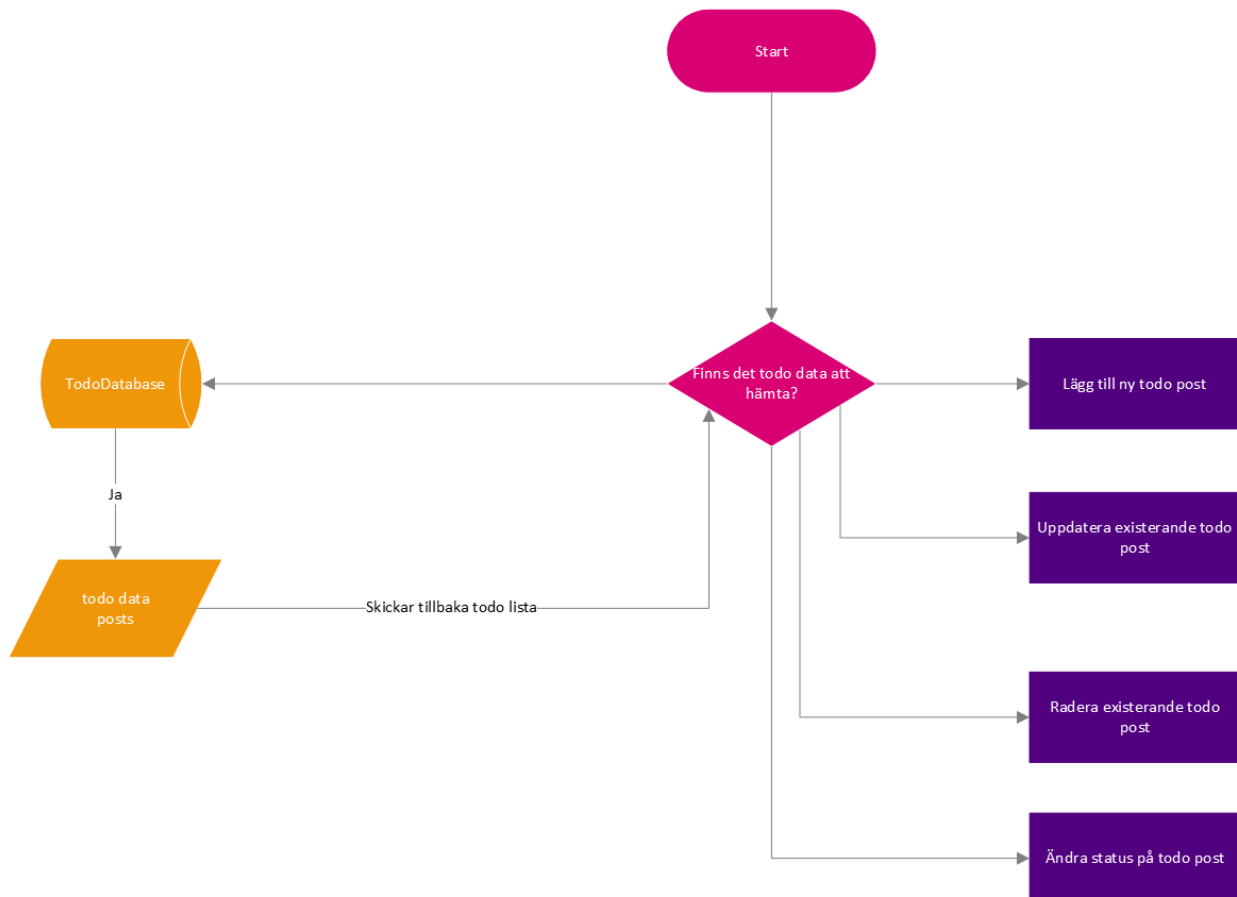
Första steget i utveckling av applikationen var att skapa ER-diagram för tabellen Todos. Den innehåller all data som behöver lagras för varje To-do post, varje To-do får ett unikt id, se figur 1. De olika entiteter lagras i olika datatyper, se bilaga A. Vissa datatyper översätts till motsvarande i SQLite: String blir Text, Status blir Integer och DueDate blir Text.



Figur 1 – Er-diagram över Todos

6.2 Flödesschema över applikationen

För att få en överblick över funktionaliteter som behövs så utformades ett mer generellt flödesschema över hela applikation som visar hur applikationen ska fungera från start för användaren, se figur 2. Från start hämtas alla



Figur 2 – Flödesschema över applikationen från start

To-do poster som finns i databasen. Utifrån användarens 4 möjliga val som inkluderar att lägga till ny To-do, uppdatera, radera och ändra status på To-do post exekveras olika funktionaliteter, se figur 2.

Varje val motsvarar varsitt flödesschema och visar logiken och hur funktionen är tänkt att fungera, se alla figurer i bilaga B.

Efter att ha byggt upp logiken bakom funktionaliteterna skapades ett nytt server Blazor projekt som både hanterade databasanslutning, lagring och uppbyggnad av ett användargränssnitt (UI).

6.3 Todo klass

Därefter skapades en Todo.cs klass med alla variabler som behövdes inför lagring. Varje variabel blev satt till private och med hjälp av get; och set; kunde jag senare komma åt dem utanför klassen, se figur 3.

```
//Definierar struktur på todo data
private int id;
private string title = string.Empty;
private string description = string.Empty;
private string category = string.Empty;
private int priority;
private bool status;
private DateTime dueDate;
//Definierar get och set på alla variabler
public int Id { get { return id; } set { id = value; } }
public string? Title { get { return title; } set { title = value; } }
public string? Description { get { return description; } set { description = value; } }
public string? Category { get { return category; } set { category = value; } }
public int Priority { get { return priority; } set { priority = value; } }
public bool Status { get { return status; } set { status = value; } }
public DateTime DueDate { get { return dueDate; } set { dueDate = value; } }
```

Figur 3 – Deklarerade/initierade alla variabler mha get; set;

En konstruktör för Todo klassen användes för att säkerställa att vissa värden på variablerna skickas med som standard varje gång en ny initiering skapas, se figur 4. Varje gång en ny To-do post skapas ska statusen vara satt till falskt dvs icke avklarad.

```
public Todo()
{ //Avrundar tiden till närmaste minut
    DateTime timeNow = DateTime.Now;
    DueDate = new DateTime(timeNow.Year, timeNow.Month, timeNow.Day,
timeNow.Hour, timeNow.Minute, 0);

    //Status ska alltid vara satt till false(=0), dvs inte avklarad
    när ny todo läggs till
    Status = false;
    Category = "";}
```

Figur 4 – Standard värden som alltid skickas med.

6.4 *TodoContext* klass

Självpaste databasanslutning sker i en ny klass som heter *TodoContext.cs* för att skapa struktur och bryta ner koden samt bevara objektorienterad metodik. Varje klass i applikationen har specifika uppgifter, funktionaliteter och ansvarsområden. I *TodoContext* skapas både konfiguration och uppsättning av databasschemat mha Entity Framework.

Paketet för Entity Framework läggs till i projektet:

```
dotnet add package Microsoft.EntityFrameworkCore
```

och används i klassen *TodoContext*:

```
using Microsoft.EntityFrameworkCore;
```

Databasen konfigureras och kopplas till projektet, se figur 5. Databasfilen skapas i förväg med namnet *ToDoDatabase.db*. Sökvägen till databasfilen definieras i *appsettings.json*, se figur 6

```
//Konfigurerar typ av databas
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    //Entity framework core använder metod UseSqlite för att hämta ToDoDB i appsettings.json
    optionsBuilder.UseSqlite(Configuration.GetConnectionString("ToDoDB"));
}
```

Figur 5 – Konfiguration av databas mha Entity Framework

```
"ConnectionStrings": {
  "ToDoDB": "Data Source=Components\\Data\\ToDoDatabase.db"
}
```

Figur 6 – Sökväg till databasen.

Därefter skapas modellen för att bygga upp tabellen i SQLite. *Todo* objekt skickas med in som testdata för att se att det är möjligt att skicka in data i tabellen, se figur 7.

```
//Skapar upp modellen som kommer bygga upp tabellen Todos
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Todo>()
```



```
.ToTable("Todos");  
modelBuilder.Entity<Todo>()  
    .HasData(  
        //Testdata till databasen  
        new Todo  
        { //Test data Todo objekt})  
    }
```

Figur 7 – Modell som skapar upp tabellen i ToDoDatabase.db

6.5 Initial migration –Todos tabell SQLite

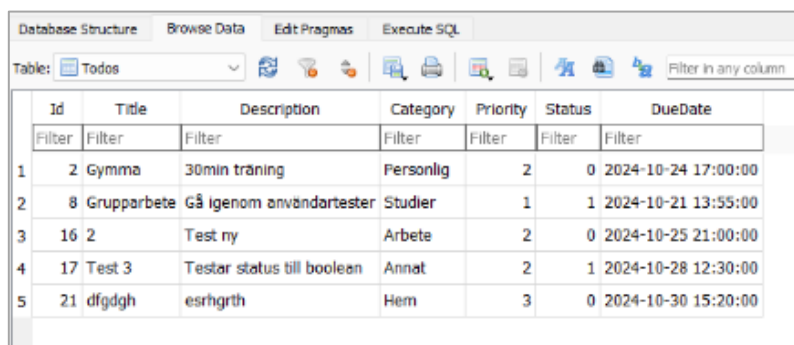
När konfigurationen, koppling och testdata är redo att lagras skapas en Initial migration för att skapa databasen med hjälp av följande kommandon:

```
dotnet ef migrations add InitialCreate
```

```
dotnet ef migrations add SeedData
```

```
dotnet ef database update
```

I desktop app DB Browser kan vi öppna databasfilen och se att testdata har lagrats, se figur 8. Tabellen har strukturerats upp med tidigare nämnda variabler i Todo, se figur 3.



| | Id | Title | Description | Category | Priority | Status | DueDate |
|---|--------|-------------|----------------------------|-----------|----------|--------|---------------------|
| | Filter | Filter | Filter | Filter | Filter | Filter | Filter |
| 1 | 2 | Gymma | 30min träning | Personlig | 2 | 0 | 2024-10-24 17:00:00 |
| 2 | 8 | Grupparbete | Gå igenom användartester | Studier | 1 | 1 | 2024-10-21 13:55:00 |
| 3 | 16 | 2 | Test ny | Arbete | 2 | 0 | 2024-10-25 21:00:00 |
| 4 | 17 | Test 3 | Testar status till boolean | Annat | 2 | 1 | 2024-10-28 12:30:00 |
| 5 | 21 | dfgdgh | esrhgrth | Hem | 3 | 0 | 2024-10-30 15:20:00 |

Figur 8 – DB Browser som visar Todos tabell med testdata

6.6 Blazor

För att kunna hämta alla To-do poster, lägga till ny, uppdatera, radera och markera avklarade To-do poster, skapades ett formulär som kopplades till C# funktionaliteter. Det gjordes med @bind och @onclick metoder för att utföra olika funktionaliteter enligt flödesschema för varje uppgift, se bilaga

B. **Home.razor.cs** komponent innehåller all funktionalitet och är kopplad till Home.razor som innehåller all HTML kod. Kommunikationen skedde med hjälp av "bindings".

För att hämta data och ändra befintlig data, skapas en ny DbContext instans enligt figur 9.

```
dataContext ??= await TodoDataContextFactory.CreateDbContextAsync();
```

Figur 9 – Ny koppling till databasen

Därefter hämtades datan till readTodoList som är Todo av typen List enligt figur 10.

```
private List<Todo>? readTodoList;
```

Figur 10 – Data från databasen hämtas som Todo av typen List

Data hämtas och sorteras i ordning efter status, prioritet och titel enligt figur 11.

```
if (dataContext is not null)
{
    ReadTodoList = await dataContext.Todos
        .OrderBy(todo => todo.Status)
        .ThenBy(todo => todo.Priority)
        .ThenBy(todo => todo.Title)
        .ToListAsync();
}
```

Figur 11 – Datahämtning och sortering

För att skriva ut To-do posts vid start körs en foreach i Home.razor komponenten enligt figur 12.

```
<!--För varje todo post skriv ut-->
@foreach (var todo in @ReadTodoList)
```

Figur 12 – Foreach metod för att skriva ut varje To-do post som är lagrad i databasen

Med hjälp av Entity Framework användes inbyggda metoder i [Home.razor.cs](#) för att hantera databasoperationer byggt på CRUD såsom lägga till, uppdatera, radera och markera To-do post som avklarad enligt figur 13. Metoderna utförde operationerna och funktionaliteten likt logiken bakom de skapade flödesscheman, se bilaga B.

```
dataContext.Todos.Add(NewTodo);
dataContext.Todos.Update(TodoToUpdate);
dataContext.Todos.Remove(todoPost);
```

Figur 13 – Inbyggda metoder för att skicka ändringar till databasen.

Med hjälp av knappar kunde information skickas med @onclick till metoder i C# koden enligt figur 14.. För att dynamiskt ändra status på en To-do post som avklarad/icke avklarad behövdes även en @bind. Bindningen gjorde att statusen på To-do posten ändrades med toggle och sen skickades ändringar till databasen med metoden Update enligt figur 15.

```
<input type="checkbox" name="check-todo" class="check-todo"
  @onclick="() => MarkTodoStatus(todo)" @bind="@todo.Status">
<div class="btn-todo-post-wrap">
  <button name="button" type="button" class="btn btn-primary"
    @onclick="() => ShowEditTodoForm(todo)">Ändra</button>
  <button name="button" type="button" class="btn btn-secondary"
    @onclick="() => DeleteTodoPost(todo)">Radera</button>
</div>
```

Figur 14 – Knapparna är bundna till metoder för att utföra ändringar

```
todo.Status = !todo.Status;
//Uppdaterar ändring
dataContext.Todos.Update(todo);
```

Figur 15 – Statusen på To-do post togglas och ändringen skickas till databasen

6.6.1 Felhantering

Projektet använde ett enklare felhanteringssystem som kontrollerade att alla fält förutom tid och status var ifyllda i både formuläret och uppdatering av To-do poster. Med hjälp av en valideringsmetod kunde både formuläret och uppdatering kontrolleras enligt figur 16. Om inputfälten inte var ifyllda visades ett felmeddelande enligt figur 17.

```
private bool IsFormValid(Todo todoToValidate)
{
    return todoToValidate is not null && todoToValidate.Priority != 0
    && todoToValidate.Description is not null &&
    todoToValidate.Description != "" && todoToValidate.Title is not null
    && todoToValidate.Title != "" && todoToValidate.Category != "";
}
```

Figur 16 – Metod som kontrollerar att inga fält är null eller tomma

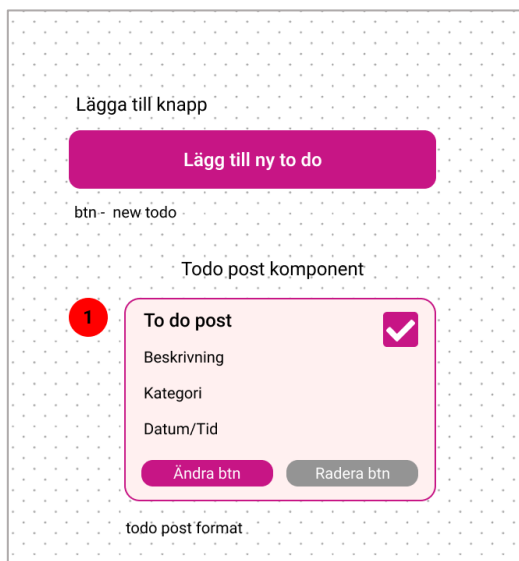
```
@if (@HasError)
{
```

```
<div class="error-msg">
    <p>Alla fält måste vara ifyllda!</p>
</div>
}
```

Figur 17 – Skriver ut felmeddelande

6.6.2 Styling

Stylingen av webbsidan har utgick ifrån wireframes i Bilaga C och den grafiska profil i Bilaga D. CSS på startsidan har lagts till i [Home.razor.css](#). Den kapslade in Home.razor komponent och all CSS styling som skapades i filen ändrade endast komponentens element. Följande figur 18 visar temat på MyPlanner applikationen



Figur 18 – Grafiska profil på applikationen

7 Resultat

Utifrån de uppsatta punkterna i problemmotiveringen så har projektet MyPlanner nått följande mål:

- Wireframes har skapats för applikationens layout.
- Passande favicon och logotyp finns i applikationen.
- ER-diagram över vad som ska lagras i To-do listan, se bilaga A.
- Skapat Flödesscheman, se bilaga B.
- Skapat SQLite databas med tabell Todos, se bilaga E.
- Skapat nytt Blazor projekt MyPlanner.
- Kopplat Blazor till databasen mha Entity Framework Core.
- Utfört CRUD operationer mha Entity Framework Core.
- Skapat html struktur för sidan med formulär, knappar och To-do poster.
- Bundit html element med C# funktionaliteter mha @bind och @onclick.
- Lagt till felhantering för tomma input.
- Stylat element med CSS enligt wireframes och grafiska profilen.

Mål som inte har följts:

- Designskisser har inte skapats eftersom applikationen är av mindre omfattning.

8 Slutsatser

Utveckling av applikationen har inte varit linjär. Jag började först med att utveckla ett Memory spel med Windows Forms. Men efter en veckas frustration valde jag att skrota idén och satsa på att skapa en webbapplikation som lagrar Att-göra lista kombinerad med SQLite databas. Det fanns gott om information och guider om Blazor och koppling till databaser vilket gjorde projektet smidigare att genomföra.

Själva applikationen MyPlanner är enkel byggt, utmaningen har snarare legat i att förstå Blazor, dess funktionaliteter och möjligheter. Det finns mycket mer att utforska vidare.

Den svåraste och mest tidskrävande delen var att sätta upp strukturen och kopplingen till databasen. Trots att det fanns guider att följa så blev det lite svårt att få till strukturen rätt. Jag behövde därmed göra flera försök fram och tillbaka för att få rätt struktur.

Valideringen i MyPlanner bygger på en enkel felhantering som endast validerar om alla fält är fyllda. Här hade jag kunnat vidareutveckla ett valideringssystem som validerar separata fält med hjälp av Editform i Blazor. Editform hade gett mer funktionalitet, kontroll och valideringsmöjligheter. Dock var tiden begränsat för att utforska Blazor och därmed valde jag använda en enkel felhantering.

Jag hade även kunnat tillämpa Machine learning för att automatisera prioritering och kategorier av To-do poster. Men återigen så hade det varit tidskrävande och därmed fick jag utesluta ML i projektet.

Jag har under utvecklingens gång lärt mig mycket om kopplingar till databas, hur Entity Framework kan förenkla arbetet, hur element binds till C# kod för att utföra olika funktionalitet. Det har varit givande att skapa ett projekt i Blazor och ser fram emot att vidareutveckla mina kunskaper och förståelse i Blazor.

Källförteckning

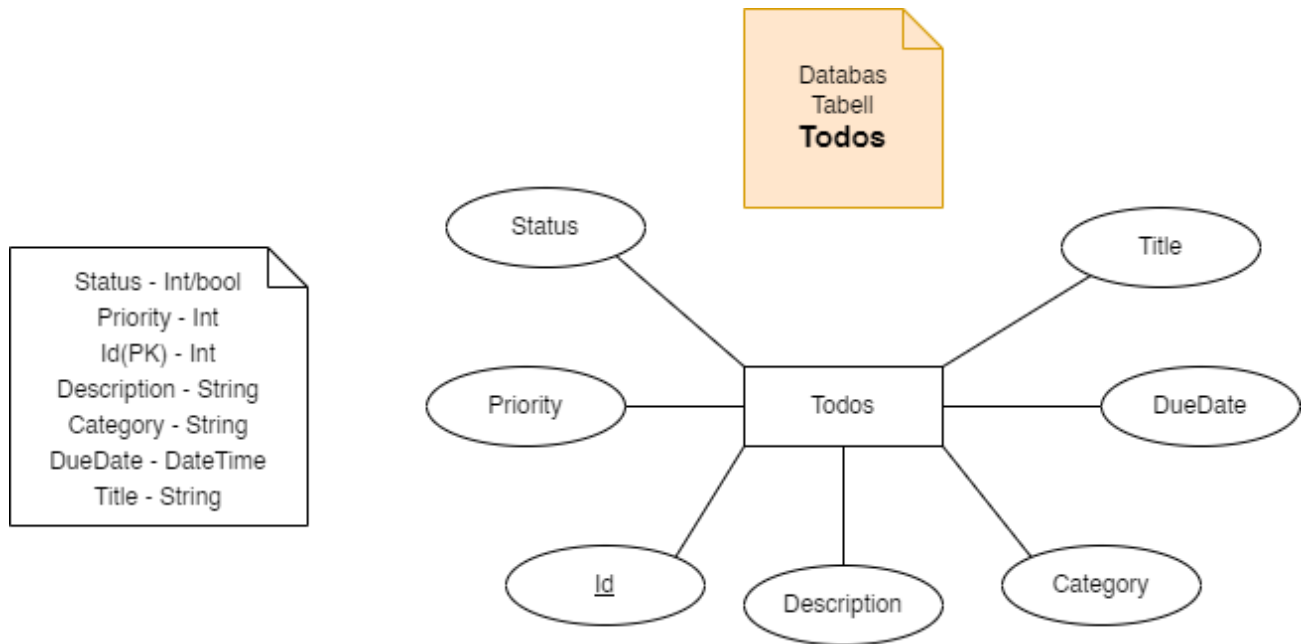
- [1] Hasselmalm, Mikael. (2024-09-10). "Programmeringsspråket C#". [Föreläsning]. Mittuniversitet. Tillgänglig från: https://play.miun.se/media/C%20HT24/0_34j6fzoi.
- [2] Microsoft Learn, "Objektorienterad programmering (C#)", <https://learn.microsoft.com/sv-se/dotnet/csharp/fundamentals/tutorials/oop>. Publicerad 2023-06-08. Hämtad 2024-10-26.
- [3] Hasselmalm, Mikael. (2024-09-24). "Objektorienterad C#". [Föreläsning]. Mittuniversitet. Tillgänglig från: https://play.miun.se/media/C%20OOP%20HT24/0_1cr7dd4t.
- [4] Microsoft Learn, "Async return types (C#)", <https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/async-return-types>. Publicerad 2023-08-04. Hämtad 2024-10-26.
- [5] Microsoft Learn, "Introduktion till klasser", <https://learn.microsoft.com/sv-se/dotnet/csharp/fundamentals/types/classes>. Publicerad 2023-08-24. Hämtad 2024-10-26.
- [6] M J. Price, *C#12 and .NET 8 Modern Cross-Platform Development Fundamentals*. 8 uppl. Birmingham: Packt Publishing Ltd. 2023.
- [7] Csharp.progdocs.se, "Entity Framework Core", <https://csharp.progdocs.se/annat/databaser/entity-framework-core>. Publicerad Okänt. Hämtad 2024-10-29.
- [8] Morgan J., "Blazing Fast CRUD: Using Entity Framework Core and SQLite in Blazor Part 3". ALL HANDS ON TECH, <https://www.allhandsontech.com/programming/blazor/how-to-sqlite-blazor-3/>. Publicerad Okänt. Hämtad 2024-10-31.
- [9] Morgan J., "Blazing Fast CRUD: Using Entity Framework Core and SQLite in Blazor Part 4". ALL HANDS ON TECH, <https://www.allhandsontech.com/programming/blazor/how-to-sqlite-blazor-4/>. Publicerad Okänt. Hämtad 2024-10-31.

- [10] Microsoft Learn, "ASP.NET Core Blazor", <https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-8.0#build-a-full-stack-web-app-with-blazor>. Publicerad 2024-09-02. Hämtad 2024-10-31.

- [11] Microsoft Learn, "ASP.NET Core Blazor data binding", <https://learn.microsoft.com/en-us/aspnet/core/blazor/components/data-binding?view=aspnetcore-8.0>. Publicerad 2024-06-05. Hämtad 2024-10-31.

Bilaga A: ER-diagram

SQLite – todos

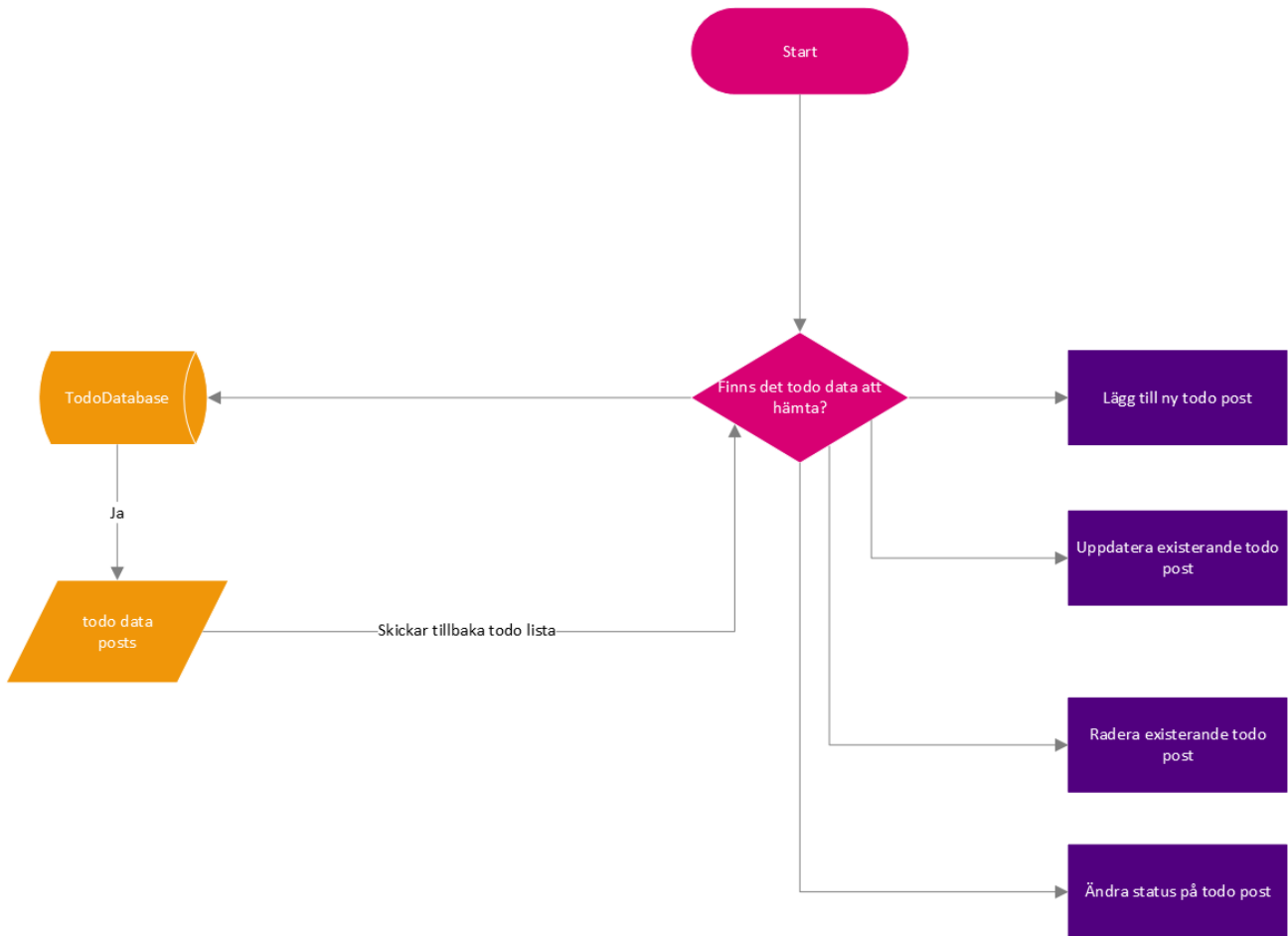


Figur A – ER-diagram över tabellen Todos och datatyp för varje entitet.

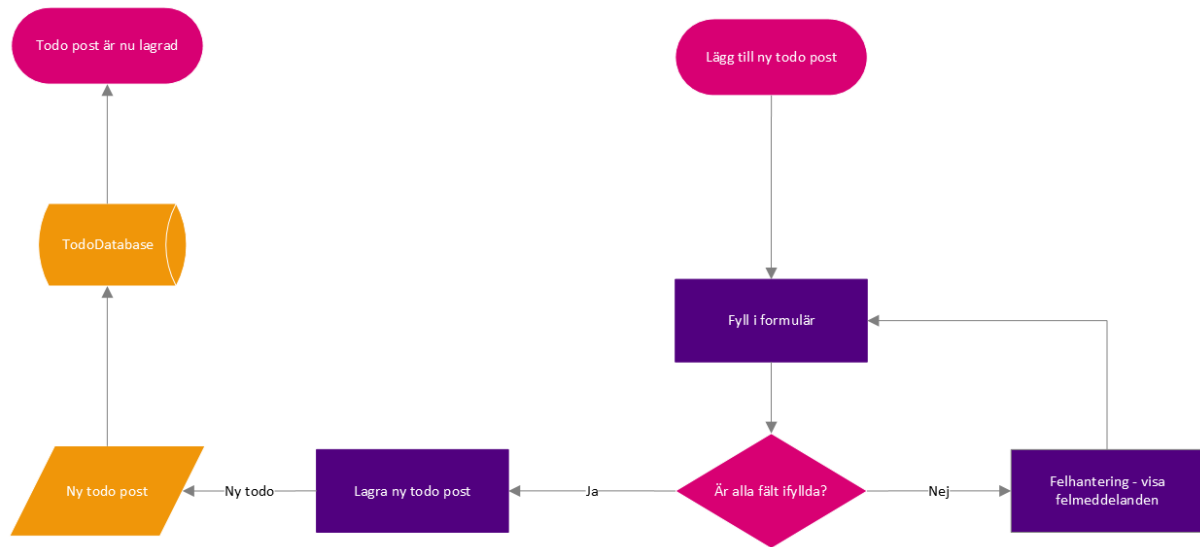
Vissa datatyper översätts till motsvarande i SQLite: String blir Text, Status blir Integer och DueDate blir Text.

Bilaga B: Flödesscheman

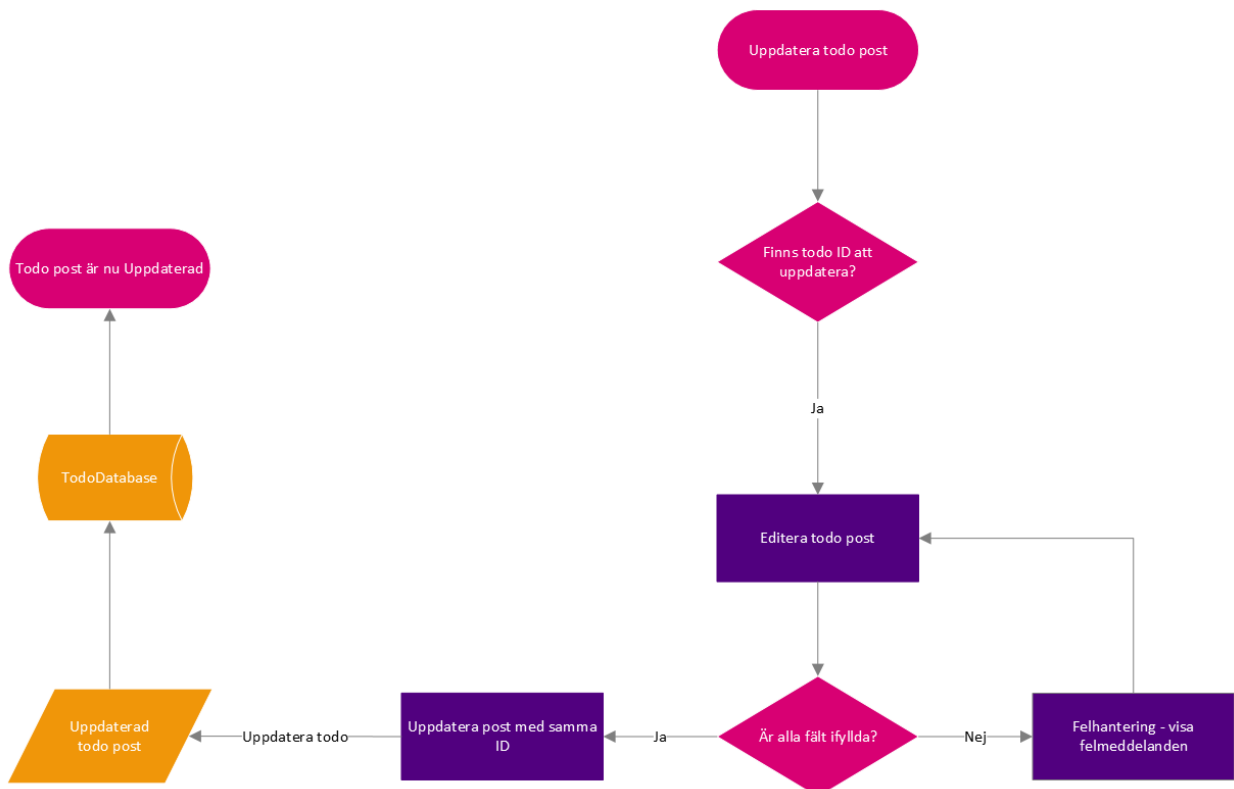
Logiken bakom funktionaliteten



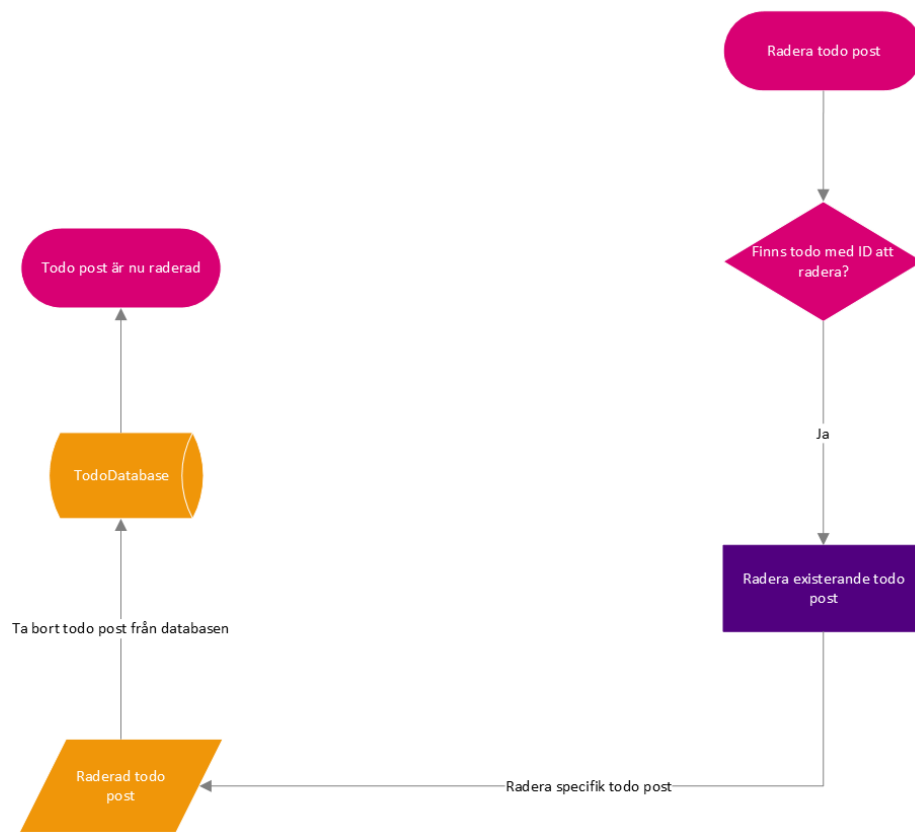
Figur B – Överblick över alla funktionaliteter



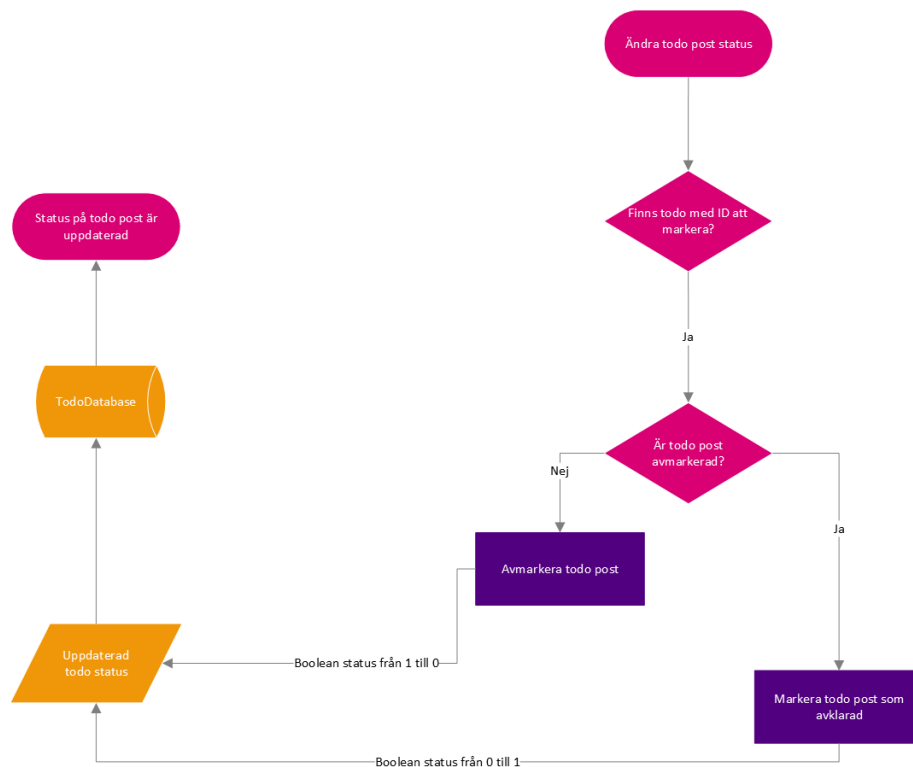
Figur B2 – Flödesschema: Lägg till ny todo post



Figur B3 – Flödesschema: Uppdatera todo post



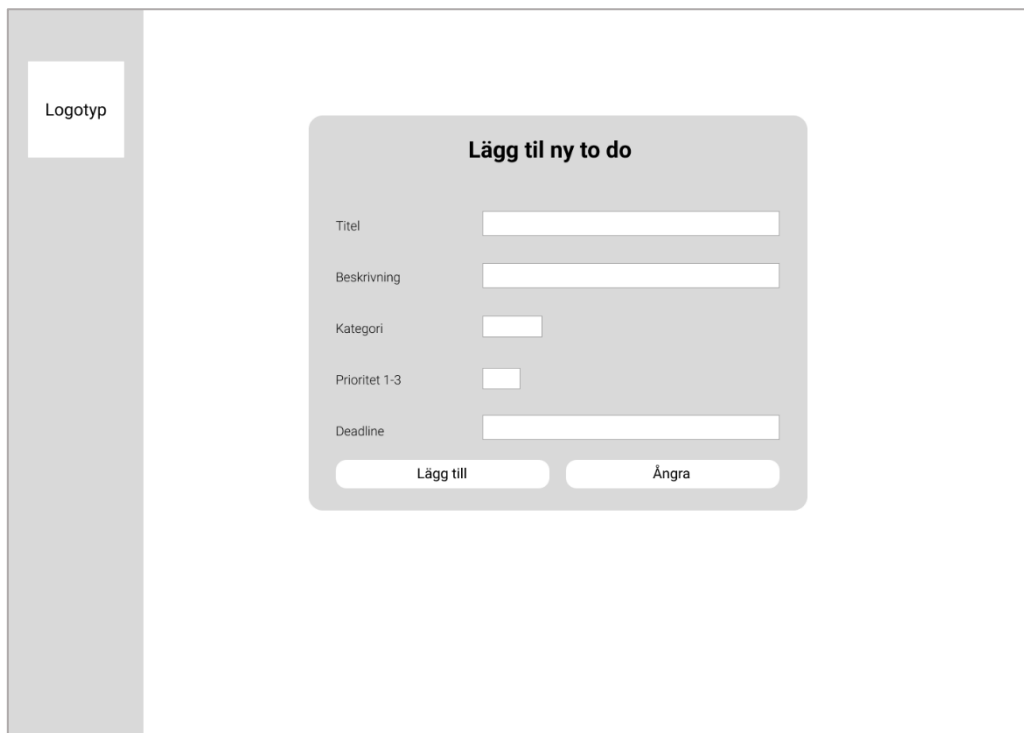
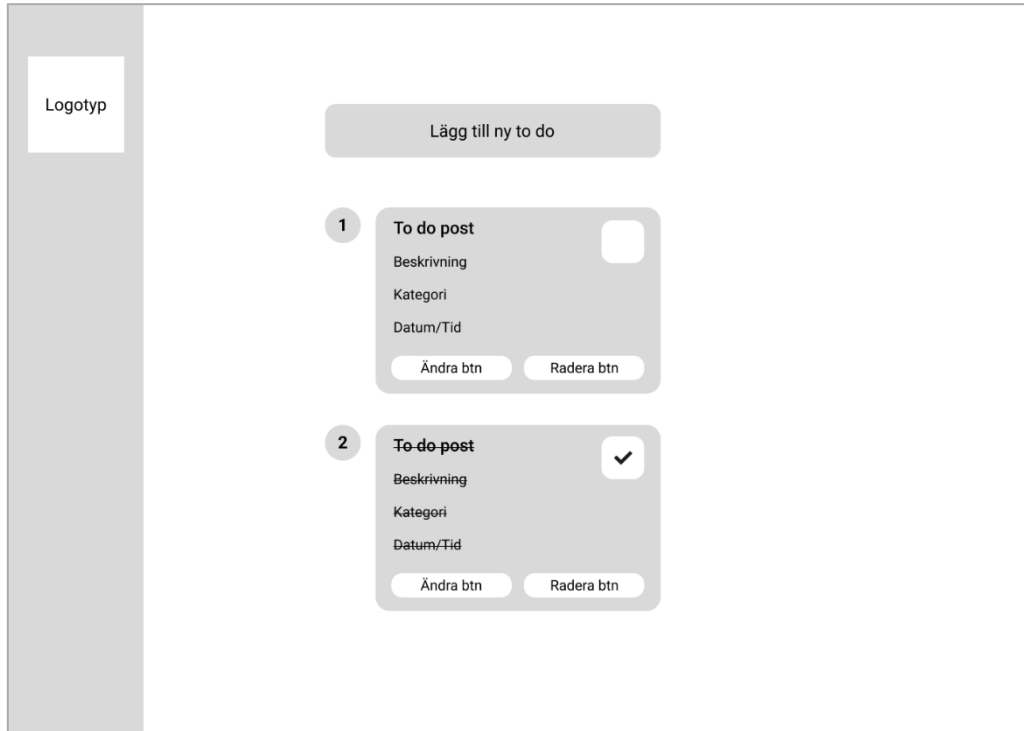
Figur B4 – Flödesschema: Radera todo post



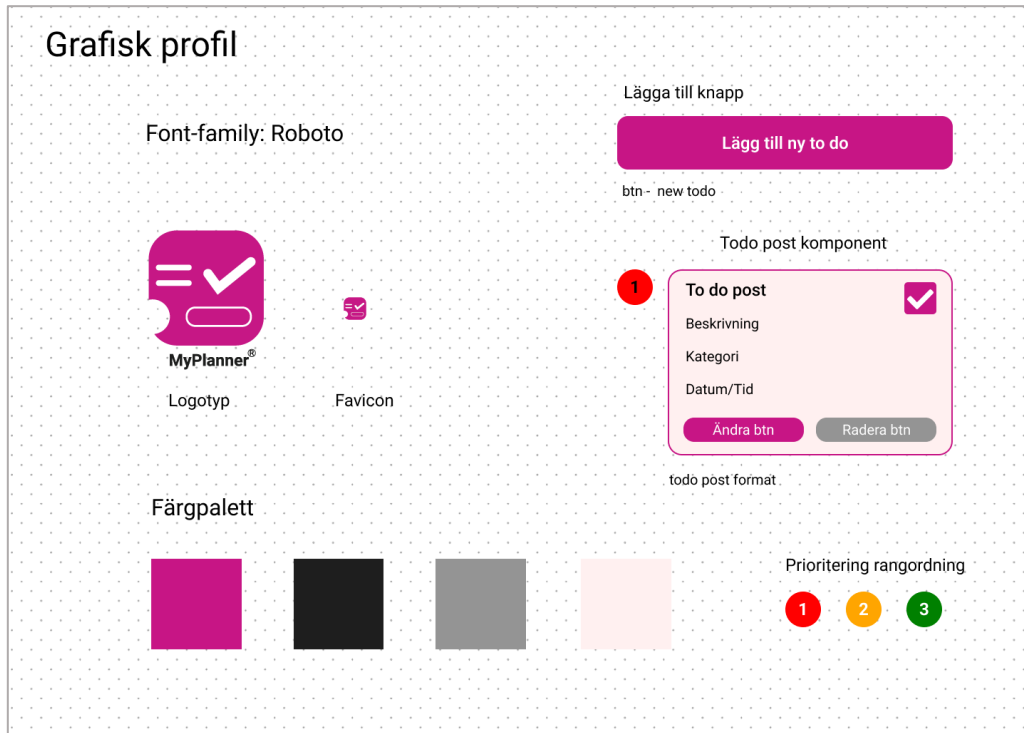
Figur B5 – Flödesschema: Uppdatera status på todo post

Bilaga C: Wireframes

Layout och struktur



Bilaga D: Grafisk profil



Bilaga E: Databas SQLite

| Database Structure | | |
|---|---------|---|
| Browse Data Edit Pragma's Execute SQL | | |
| Create Table Create Index Modify Table Delete Table Print | | |
| Name | Type | Schema |
| ▼ Tables (3) | | |
| ▼ Todos | | CREATE TABLE "Todos" ("Id" INTEGER NOT NULL CONSTR |
| Id | INTEGER | "Id" INTEGER NOT NULL |
| Title | TEXT | "Title" TEXT |
| Description | TEXT | "Description" TEXT |
| Category | TEXT | "Category" TEXT |
| Priority | INTEGER | "Priority" INTEGER NOT NULL |
| Status | INTEGER | "Status" INTEGER NOT NULL |
| DueDate | TEXT | "DueDate" TEXT NOT NULL |
| > __EFMigrationsHistory | | CREATE TABLE "__EFMigrationsHistory" ("MigrationId" TE |
| > sqlite_sequence | | CREATE TABLE sqlite_sequence(name,seq) |
| Indices (0) | | |
| Views (0) | | |
| Triggers (0) | | |

Figur E – Todos tabell struktur i databasen