

MVC-Applikation

DATABASKONSTRUKTION (IT354G)

WILLIAM LINDHOLM

Innehållsförteckning

Innehåll

Innehållsförteckning	1
Formulär (G)	2
Listinmatning (G)	4
Sökning (G)	7
Förändring av data (G)	8
Exekvering av procedur (VG?)	11
Visning av tabeller (G)	12
Automatisk generering av Tabellhuvud (VG)	12
Generering av länkar (G)	13
Användning av dolda fält (VG)	14
Diskussion	16
Bilagor	17
Bilaga A:	17

Formulär (G)

För att lägga till data i en databas via ett webbgränssnitt används ofta formulär. Med tanke på att en databas kan innehålla flera tabeller där användare önskar lägga till eller redigera data, är det effektivt att ha en generisk funktion för datainmatning. I detta sammanhang introduceras en klass kallad "ModalBuilder". Tillsammans med en "partial" vid namn "_ModalPartial" skapar denna klass automatiskt inmatningsrutor. Se genererad HTML kod i figur 1.

```
<form method="post" action="/Agents/Create">
  <div class="modal-body">
    <div class="form-group">
      <label for="CodeName">CodeName</label>
      <input name="CodeName" type="text" class="form-control"
        placeholder="CodeName">
    </div>
    <!-- ... Andra form-group element här ... -->
  </div>
  <div class="modal-footer">
    <button type="submit" class="btn btn-primary">Save changes</button>
  </div>
</form>
```

Figur 1: HTML formulär genererad av "ModalBuilder" och "_ModalPartial".

För att kunna ansluta och lägga till data i en SQL-databas behöver vissa parametrar specificeras. Dessa innefattar databasens adress (IP/domännamn), portnummer, databasens namn, samt inloggningsuppgifterna. Inom ASP.NET Core kan denna känsliga information förvaras i en säkerhetsfil, kallad "secrets" fil. Informationen i denna fil kan sedan läsas via "IConfiguration"-interfacet, vilket är en integrerad del av ASP.NET Core.

När anslutningen är etablerad, behöver man bestämma vilken tabell i databasen man vill interagera med. Användaren kan sedan specificera vilka kolumner som ska ändras eller läggas till, samt de tillhörande värdena.

För att hantera inkommande data från de genererade formulären har klassen "DatabaseRepository" implementerats. Denna klass tar emot information om vilken databastabell som data ska läggas till i, samt datans värden organiserade i form av nyckelvärdepar. Denna datastruktur underlättar kopplingen mellan varje värde och den tillhörande kolumnen, vilket innebär att värdena inte behöver anges i någon särskild ordning – en betydande fördel. Se funktionen 'CreateRow' i figur 2 för implementation.

```
public void CreateRow(string table, List<KeyValuePair<string, object>> data)
{
    if (string.IsNullOrEmpty(table) || data == null || data.Count == 0)
    {
        throw new ArgumentException("Invalid table name or data to insert.");
    }

    using MySqlConnection dbcon = new MySqlConnection(_connectionString);
    dbcon.Open();

    string columns = string.Join(", ", data.Select(d => d.Key));
    string values = string.Join(", ", data.Select(d => $"@{d.Key}"));
    string query = $"INSERT INTO {table} ({columns}) VALUES ({values});";

    using MySqlCommand cmd = new MySqlCommand(query, dbcon);
    foreach (var item in data)
    {
        cmd.Parameters.AddWithValue($"@{item.Key}", item.Value);
    }

    cmd.ExecuteNonQuery();
}
```

Figur 2: Implementation av inmatningsfunktion "CreateRow()".

I implementationen ovan i figur 2 finns funktionen "CreateRow", som syftar till att skapa en ny rad i en angiven databastabell baserat på de värden som tillhandahålls. Funktionen tar emot två parametrar: namnet på databastabellen (table) och en lista med nyckel-värde-par (data) där nycklarna representerar kolumnnamnen och värdena representerar de data som ska införas i dessa kolumner.

Först kontrolleras att både tabellnamn- och den data som matats in är giltig, annars kastas ett undantag. Därefter inleds en uppkoppling till databasen med hjälp av en "MySqlConnection", och en "INSERT" SQL-fråga byggs dynamiskt baserat på den data som angivits. Varje kolumn och värde som ska läggas till formas som parametrar i frågan för att förhindra SQL-injektionsattacker. Efter att frågan har byggts läggs värdena till kommandots parametrar, och sedan körs SQL-kommandot för att utföra insättningen i databasen.

Till skillnad från PHP, krävs här ingen specifik funktion för att ladda om den sida som data modifierats i, då kontrolldelen i MVC-mönstret sköter detta mer eller mindre automatiskt genom att en sida eller vy alltid måste vara returparameter.

Till sist implementerades felmeddelanden genom JS-biblioteket "ToastR", där felmeddelande enkelt kan skapas med hjälp av funktionen "toast.error()" där ett felmeddelande matas in som parameter. Denna kod befinner sig i huvudlayouten av projektet och läser av "TempData"-variabeln, ett lagringsmedium i ASP.NET som lagras per användarsession, där data försvinner efter att den lästs. Innehåller "TempData" ett felmeddelande skrivs denna alltså då ut genom "javascript" funktionen "toast.error()".

Med alla dessa delar i åtanke kommer kedjan för anrop till IIS (Internet Information Services, den webbserverprogramvara som ofta används av ASP.NET Core) servern som "hostar" denna sida bli följande.

1. Klienten som hämtat sidan gör ett http-anrop med den data som matats in i formuläret, eller existerar i formuläret (dolda fält).
2. Den "controller" motsvarande den action som lagts in i formuläret läser av den data som skickats. (Här fångas även eventuella fel som kan ske vid anslutningen till databasen för att kunna ge felmeddelanden som visas på sidan.)
3. Controllern översätter indata till ett format som kan läsas av de eventuella modeller som kommer tillkallas.
4. Controllern kallar eventuellt på en "Model" för att hämta nödvändiga data för att visa sidan eller informationen användaren efterfrågat.
5. "Model" anropar eventuellt "DatabaseRepository" för att kunna läsa eller skriva data från databasen.
6. "DatabaseRepository" läser av anslutningsparametrarna befunda i en "secrets"-fil för att påbörja en anslutning till "MySql"-databasen.

Listinmatning (G)

Vid inmatning av data krävs ibland inmatning av främmande nycklar, dessa är fördefinierade utifrån de primärnycklar som redan finns i den främmande tabellen. Databasen kommer därför svara med ett fel om dessa inte är inmatade korrekt. Därför bör det finnas ett sätt att välja bland de främmande nycklar som existerar.

Listboxar kan användas i detta fall, då de enkelt låter användaren välja ett av de redan existerande värdena för att undvika "foreign key constraint errors". Detta hanteras genom att automatiskt generera listboxar för dessa. Se C#-kod för generering av listinmatningsdata i figur 3.

```
public List<SelectListItem> GetColumnAsDropdown(string query)
{
    var table = GetTable(query);
    var dropdown = new List<SelectListItem>();

    foreach (DataRow row in table.Rows)
    {
        string value = string.Join(", ", row.ItemArray.Select(item =>
            item.ToString()));

        dropdown.Add(new SelectListItem
        {
            Value = value,
            Text = value
        });
    }

    return dropdown;
}
```

Figur 3: Generering av "dropdown" inmatning.

I figur 3, finns koden som automatisk hämtar en, eller flera kolumner, som sedan placeras i en listinmatning. Hämtas fler än en kolumn, separeras de hämtade värdena med kommatecken, detta används för att kunna mata in främmande nycklar bestående av fler än en primärnyckel. Här användes

datatypen "List<SelectListItem>", "SelectListItem" består av flera olika parametrar bland annat "Text" och "Value", vilket gör den perfekt för lagring av värden som ska matas in i en "dropdown".

Koden hanterar i figur 3 hanterar inte databaskopplingen själv utan, förlitar sig istället på funktionen "GetTable"; en annan funktion i denna klass som används för att hämta hela tabeller från databasen. Antingen med hjälp av ett SQL-kommando, eller en sträng endast innehållande namnet på en tabell eller vy. Detta bidrar till mer återanvänd kod, och mindre upprepad kod. Se funktionen "GetTable" i figur 4.

```
public DataTable GetTable(string query)
{
    using (MySqlConnection dbcon = new MySqlConnection(_connectionString))
    {
        if (!query.Contains(" "))
        {
            query = $"SELECT * FROM {query}";
        }

        dbcon.Open();
        MySqlDataAdapter adapter = new MySqlDataAdapter(query, dbcon);
        DataSet ds = new DataSet();
        adapter.Fill(ds, "result");
        return ds.Tables["result"];
    }
}
```

Figur 4: Hämtning av tabeller från databas.

I figur 4 hämtas tabeller från databasen med hjälp av en "MySqlConnection" innanför ett "using" block. Detta ser till att, om saker går fel, stängs och rensas kopplingen direkt, istället för att potentiellt inte komma till den rad som stänger kopplingen genom "dbcon.close()". Den data som hämtas placeras i ett DataTable, som sedan enkelt kan läsas för att hämta värden, tillsammans med deras korresponderande kolumnnamn.

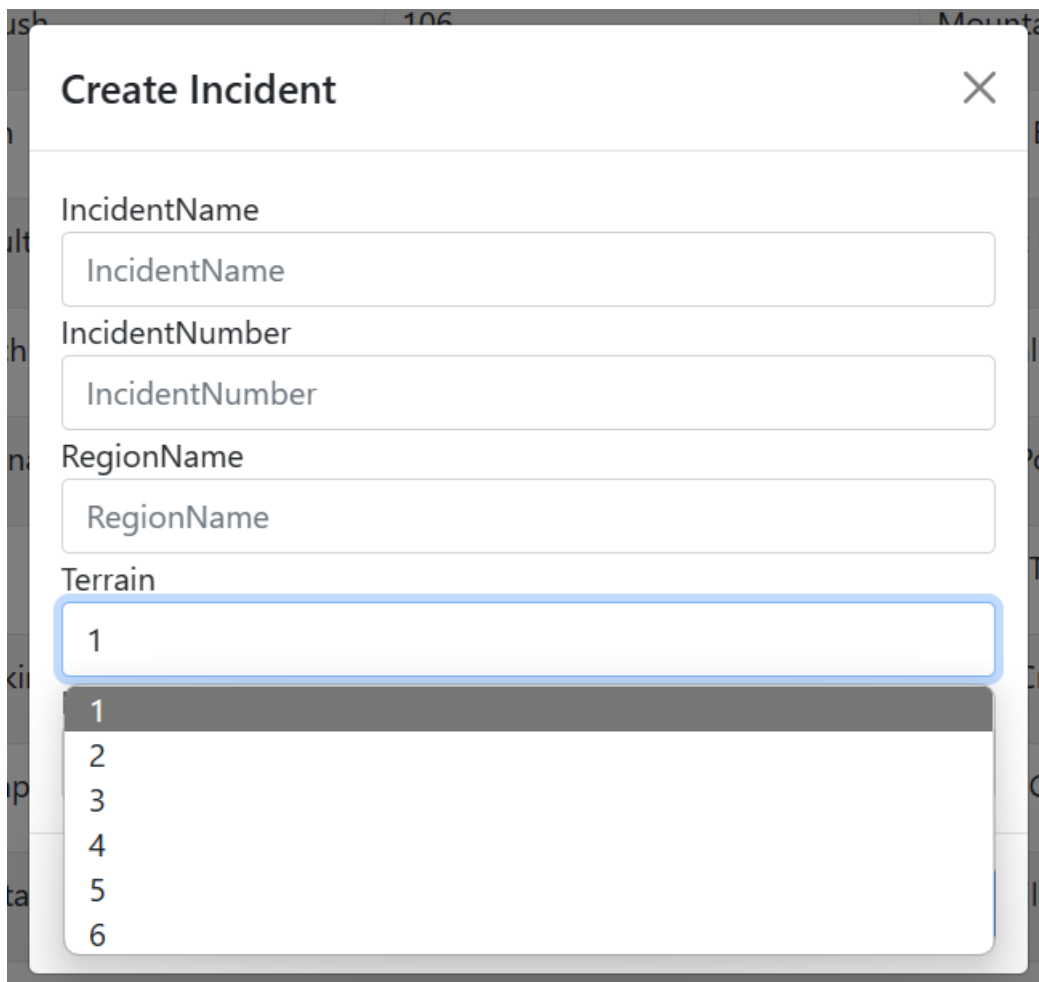
Vid generering av HTML-koden för listboxar finns i ASP.NET mer strukturerade sätt att göra detta på än den "råa" strängkonkateneringen som ofta används i PHP-applikationer. Istället används en del i "_ModalPartial" som läser av listan av "SelectListItem":s enligt koden i figur 5. Se resulterande HTML i figur 6.

```
case "dropdown":
    <div class="form-group">
        <label for="@input.Id">@input.Label</label>
        <select name="@input.Id" class="form-control" id="@input.Id">
            @foreach (var option in input.DropdownItems)
            {
                <option value="@option.Value">@option.Text</option>
            }
        </select>
    </div>
    break;
```

Figur 5: Skapande av HTML-dropdown.

```
<div class="form-group">
  <label for="Terrain">Terrain</label>
  <select name="Terrain" class="form-control" id="Terrain">
    <option value="1">1</option>
    <option value="2">2</option>
    <option value="3">3</option>
    <!-- ... Fler värden här ... -->
  </select>
</div>
```

Figur 6: resulterande HTML-kod för dropdown.



The image shows a web form titled "Create Incident" with a close button (X) in the top right corner. The form contains four input fields: "IncidentName", "IncidentNumber", "RegionName", and "Terrain". The "Terrain" field is a dropdown menu that is currently open, showing a list of numbers from 1 to 6. The number "1" is selected and highlighted in the dropdown list. The background of the form is a light gray, and the dropdown menu has a white background with a blue border.

Figur 7: Resultat av skapande av dropdown.

Sökning (G)

För att genomföra sökningar efter värden i databasen används i detta fall teckenmatchning med "LIKE" i SQL. Där rader kan filtreras utifrån en sträng inmatad av användaren.

```
//Controller
public IActionResult Index(string IncidentName = "")
{
    ViewBag.Table = _incidentsModel.GetIncidents(IncidentName);
    ViewBag.Modal = _incidentsModel.CreateIncidentModal();

    return View();
}

//Model
public TableObject GetIncidents(string searchQuery)
{
    return _tableBuilder
        .SetDataTable(_databaseRepository.GetTable($"SELECT * FROM Incident
        WHERE IncidentName LIKE '{searchQuery}%'"))
        .SetRedirect("Details")
        .Build();
}
```

Figur 8: Hantering av sökning (Controller & Model)

I figur 8, tar Index-funktionen emot den sökparameter som potentiellt matats in, därefter skickas denna med vid skapande av tabellen på sidan. Vid användning av "LIKE" följt av ett strängvärde omringat av (%) procenttecken, kommer alla värden visas om ingen söksträng matas in eftersom detta resulterar i "LIKE '%" vilket kommer matcha allt. Det finns därför ingen separat hantering om användaren inte genomför en sökning. Se hur denna sökning och SQL-fråga konstruerar motsvarande tabell i figur 4.

Förändring av data (G)

För varje rad i tabellerna som genereras, kan även "ActionLinks" som raderar data genereras. För att detta ska kunna ske på ett sätt som resulterar i så mycket återanvändbar kod som möjligt implementerades en funktion som tar emot den tabell data ska raderas från, samt nyckelvärdepar som kan användas för att identifiera den rad som ska raderas. Se implementation av funktionen "DeleteRow()" i figur 9.

```
public void DeleteRow(string table, List<KeyValuePair<string, object>>
primaryKeys)
{
    string whereClause = "WHERE ";
    for (var i = 0; i < primaryKeys.Count; i++)
    {
        whereClause += $"{primaryKeys[i].Key} = '{primaryKeys[i].Value}'";
        if (i < primaryKeys.Count - 1)
        {
            whereClause += " AND ";
        }
    }

    using (MySQLConnection dbcon = new MySQLConnection(_connectionString))
    {
        dbcon.Open();
        MySqlCommand cmd = new MySqlCommand($"DELETE FROM {table}
{whereClause};", dbcon);
        cmd.ExecuteNonQuery();
    }
}
```

Figur 9: Generell funktion för borttagning av data.

Funktionen "DeleteRow" i figur 9 tar emot de värden som ska raderas genom en lista av nyckelvärdepar. Denna metod möjliggör en enkel hämtning av kolumnvärdet eller värdet i en kolumn. Därefter skapas en selektion som filtrerar ut rätt data som ska raderas genom att använda alla primärnycklar som tillhandahålls i funktionen. Slutligen utförs SQL-koden med "ExecuteNonQuery()" eftersom den inte förväntar sig någon data som returneras när data raderas. För att generera de länkar som raderar en rad används koden i figur 10.

```
@if (Model.PrimaryKeys?.Count > 0)
{
    foreach (var key in Model.PrimaryKeys)
    {
        urlQuery[key] = row[key];
    }
    deleteQuery = new RouteValueDictionary(urlQuery);
    deleteQuery["Table"] = Model.DeleteTable;
}

@if (!string.IsNullOrEmpty(Model.DeleteTable))
{
    <td class="col-1">
        @Html.ActionLink("Delete", "Delete", Model.DeletionController,
            deleteQuery, new { @class = "btn btn-block btn-danger m-0 w-100" })
    </td>
}
```

Figur 10: Generering av ActionLinks som raderar data.

Koden i figur 10 befinner sig i den tidigare nämnda "_TablePartial" modulen, som genom denna kod först hämtar de primärnycklar som definierats för en tabell, och därefter skapar ett "RouteValueDictionary" av dessa. Denna datatyp används för att enkelt kunna skicka de nödvändiga parametrarna som ett GET-anrop i URL:en då "routeValueDictionaries" kan matas in direkt i en "ActionLink" för att skicka med strängparametrar. Till sist tar en funktion kallad "Delete" emot denna data, se figur 11.

```
public IActionResult Delete(string table, string IncidentName, string
IncidentNumber, string OperationName, DateTime StartDate)
{
    try
    {
        _operationsModel.DeleteOperation(table, IncidentName, IncidentNumber,
            OperationName, StartDate);
    }
    catch (MySqlException e)
    {
        switch (e.Number)
        {
            case 1451:
                TempData["ErrorMessage"] = "Foreign Key Constraint Failed!";
                break;
            case 1452:
                TempData["ErrorMessage"] = "Foreign Key Constraint Failed!";
                break;
            default:
                TempData["ErrorMessage"] = $"Something went wrong:
                    {e.Number}";
                break;
        }
    }

    var referrer = Request.Headers["Referer"];
    if (!string.IsNullOrEmpty(referrer))
    {
        return Redirect(referrer);
    }

    return RedirectToAction("Index");
}
```

Figur 11: Kontroller ansvarig för radering från tabellen "Operation".

I figur 11 börjar koden med att ta emot en "HTTP-request" från en "ActionLink". Den här begäran innehåller information om en tabells primärnycklar och tabellnamn. Sedan anropas en raderingsfunktion som finns i modellen. Inom modellen är huvudfunktionen att anropa den generella raderingsfunktionen "DeleteRow()" med de tillhörande data som har skickats in, omvandlade till en lista av nyckelvärdepar. Det är viktigt att notera att själva raderingsprocessen omringas av ett "try-catch" block för att hantera eventuella fel som kan uppstå under raderingen av data. Eventuella felmeddelanden skickas sedan till användaren med hjälp av "toastr".

Avslutningsvis försöker koden att hämta URL:en för den tidigare sidan som användaren besökte genom att undersöka de HTTP-headers som skickades med begäran. Om det finns en så kallad "referrer" (en länk till den föregående sidan), så omdirigeras användaren tillbaka dit efter att raderingen har slutförts. Detta möjliggör återanvändning av samma raderingsfunktion för flera sidor, och användaren kommer alltid tillbaka till den sida där raderingsåtgärden ursprungligen initierades. Om ingen tidigare sida hittas, omdirigeras användaren tillbaka till indexsidan.

Exekvering av procedur (VG?)

I den databas som används finns proceduren "GetOperationsInRange" som returnerar ett antal operationer som faller innanför de datum som användaren valt att filtrera efter. Denna procedur använder då inparametrarna startdatum och slutdatum som användaren väljer genom formuläret i figur 12.

```
<form action="@Url.Action("FilterOperations")" method="get" class="mt-1 d-
inline-flex">
  <div class="me-3">
    <label for="startDateInput">Start Date:</label>
    <br />
    <input id="startDateInput" type="date" name="StartDate" class="form-
control d-inline-block" style="vertical-align: middle; width: auto;"
/>
  </div>
  <div>
    <label for="endDateInput">End Date:</label>
    <br />
    <input id="endDateInput" type="date" name="EndDate" class="form-
control d-inline-block" style="vertical-align: middle; width: auto;"
/>
    <button type="submit" class="btn btn-primary ml-2">Filter</button>
  </div>
</form>
```

Figur 12: Formulär för filtrering av operationer.

I HTML-koden i figur 12 skapas ett formulär som kallar på funktionen "FilterOperations", som då läser av vad användaren matat in för värde i "StartDate" och "EndDate" när knappen klickas på. "Controller":n tar sedan emot denna data och skickar kallar på funktionen "FilterOperations()" i modellen, se denna kod i figur 13.

```
public TableObject FilterOperations(DateTime StartDate, DateTime EndDate)
{
    return _tableBuilder
        .SetDataTable(_databaseRepository.GetTable($"CALL
        GetOperationsInRange('{StartDate}', '{EndDate}'))
        .Build();
}
```

Figur 13: Exekvering av procedur för filtrering av operationer.

Eftersom denna procedur, precis som en vanlig "SELECT"-fråga returnerar en vanlig tabell, kallas SQL-frågan för att filtrera operationerna genom funktionen "GetTable()", se figur 4.

Visning av tabeller (G)

För att visa tabeller implementerades en generell lösning, där tabellobjekt enligt strukturen i figur 14 konstrueras av en "TableBuilder"-klass. Denna datatype läses sedan av en "_TablePartial" som skriver ut tabellen.

```
public struct TableObject
{
    public DataTable DataSet;
    public string ControllerName;
    public string? DeleteTable;
    public string? DeletionController;
    public List<string>? PrimaryKeys;
    public string? Redirect;
}
```

Figur 14: Struktur för lagring av data för byggande av tabell.

I koden i figur 14 lagras först och främst ett DataTable, som innehåller all data som kommer skrivas ut på sidan, därefter innehåller den diverse information för att kunna rikta användaren mot funktioner som kan visa mer information eller radera en viss rad. Se bilaga A för vyn/partialen som läser av denna data.

En del av koden i bilaga A ansvarar för att rendera tabellen genom att iterera över DataTable.Rows, som innehåller alla rader av data. Varje datapunkt i varje rad omges av en <td>-tag enligt HTML-standarderna för att strukturera tabellrader korrekt. För att skapa alla rader, går vyn igenom varje rad i första steget och sedan, inuti varje rad, itererar igenom varje kolumn. Detta sker mellan rad 14 och 19.

Automatisk generering av Tabellhuvud (VG)

I koden bilaga A genereras även tabellhuvudet automatiskt, de namn som följer med för varje kolumn hämtas automatiskt från databasen genom att läsa "DataTable.Columns" på rad 14. Vidare, inom den efterföljande foreach-loopen, plockas varje kolumnsnamn ut och representeras av column-objektet. Detta tillvägagångssätt tillåter koden att iterera över varje kolumn från den hämtade databastabellen. För att sedan presentera namnet på varje kolumn, används "razor"-syntax i på rad 17, där @column.ColumnName hämtar och skriver ut namnet på den aktuella kolumnen från databasen. Därmed, kan koden i bilaga A dynamiskt generera varje kolumnnamn i tabellen baserat på den tabellen som hämtas från databasen.

Generering av länkar (G)

I bilaga A genereras dynamiska länkar som antingen tar bort data eller leder användaren till en sida med mer information om en rad baserat på de parametrar som skickas med från en rad. På rad 6 och 7, initieras två "RouteValueDictionary"-instanser kallade "urlQuery" och "deleteQuery". Dessa objekt används för att skapa de strängparametrar som hamnar i URL:en när en länk klickas på.

När det gäller själva skapandet av de länkar som riktar mot en detaljsida (alltså länkar som leder till mer information om en rad) kontrollerar koden först om en URL har angivits genom att inspektera "Model.Redirect" på rad 20 och 53. Om det finns en sådan URL skapas en särskild kolumn i tabellen mellan rad 22 och 25. För varje rad i tabellen samlas nyckelvärdet, eller värdena, och lagras i "urlQuery"-objektet. Därav kan en "Details"-länk på rad 56 med "Razor"-funktionen "@Html.ActionLink" genereras, som använder sig av "urlQuery" för att bestämma vilken information som ska skickas med när länken klickas på.

För raderingslänkarna kontrollerar koden först om det finns en särskild tabell att radera genom att kontrollera att "Model.DeleteTable" har ett värde på rad 26 och 59. Om ett tabellnamn anges introduceras en ny kolumn mellan rad 28 och 31. Denna kolumn fylls sedan med knappar som raderar den raden de befinner sig på genom att omdirigera till raderingsfunktionen "Delete" som sedan riktar data mot den generella implementationen funnen i figur 9. Efter att ha konfigurerat "urlQuery" baserat på den aktuella radens data, läggs namnet av den tabell som data ska raderas från till med nyckeln ["Table"]. Genom detta skapas en "Delete"-länk på rad 62, vilken utnyttjar "deleteQuery" för att avgöra vilken databasrad som ska raderas när användaren klickar på länken.

Användning av dolda fält (VG)

Varpå användaren klickat på en detaljlänk, som tar användaren vidare till en sida som exempelvis visar alla operationer som befinner sig under in incident, finns redan den incident som en ny operation skulle tillhört specificerad. Det är därför onödigt att användaren ska behöva mata in incidentens namn och nummer en extra gång vid skapande av en operation här. Därav används dolda fält för att automatiskt skicka med vilken incident en ny operation ska tillhöra om denna skapas inifrån en incidents detaljsida.

Dessa dolda fält skapas genom att vid modalens konstruktion, skapa fält där typen sätts till hidden, se kod i figur 15. Sedan kommer de paramtetrar som matats in i klassen "ModalBuilder" att överättas till HTML genom den kod befunnen i figur 16.

```
var modalBuilder = new ModalBuilder()
    .SetTitle("Create Operation")
    .SetIdentifier("createOperationModal")
    .SetAction("CreateOperation", "Incidents")
    .AddInput("OperationName", "OperationName", "normal", "OperationName")
    //fler AddInput rader här...
    .AddInput("IncidentName", "IncidentName", "hidden", IncidentName)
    .AddInput("IncidentNumber", "IncidentNumber", "hidden",
IncidentNumber.ToString());
```

Figur 15: Skapande av formulär med dolda fält.

```
case "hidden":
    <input type="hidden" name="@input.Id" id="@input.Id"
        value="@input.Placeholder">
    break;
```

Figur 16: Generering av dolda fält i HTML.

I detta fall har "IncidentName" och "IncidentNumber" matats in i denna funktion med parametrar, och används därav för de dolda fälten. Detta eftersom användaren klickat på den incident vid namn "Breach" som har nummer "105" Se resulterande HTML i figur 17.

```
<form method="post" action="/Incidents/CreateOperation">
    <input type="hidden" name="IncidentName" id="IncidentName" value="Breach">
    <input type="hidden" name="IncidentNumber" id="IncidentNumber"
        value="105">

    <button type="submit" class="btn btn-primary">Save changes</button>
</form>
```

Figur 17: Resulterande HTML formulär (Onödig kod bortskalad för enklare läsning).

Resultatet av detta alltså i detta fall att en användare kan klicka på en incident, sedan klicka på skapa operation inuti denna incident, och då inte behöva mata in incidentens namn eller nummer. Se bild på resultat i figur 18. Notera att inmatning för incident inte syns på bilden.

The image shows a 'Create Operation' dialog box with a close button (X) in the top right corner. It contains five input fields and two action buttons at the bottom.

Field Label	Current Value / Placeholder	Special Features
OperationName	OperationName	Text input
StartDate	åååå - mm - dd	Calendar icon
EndDate	åååå - mm - dd	Calendar icon
Operation Result	Success	Text input
GroupLeader	A1	Text input

Buttons: Close (grey), Save changes (blue)

Figur 18: Bild på formulär med dola fält.

Diskussion

Som alltid, vid skapande av mjukvara finns det ett oändligt antal sätt att implementera de funktioner som efterfrågas. I denna implementation finns det många saker jag hade gjort annorlunda, om jag hade startat om. Bland annat fanns inte den fulla förståelsen för hur MVC-mönstret implementeras som bäst. Därmed skapades mycket onödig "boiler Plate" kod, då flera funktioner implementerades på nytt för varje "Controller". Mer specifik hade de modeller som skapades för de olika delarna av databasen ha återanvänts i flera "Controllers", så länge de läser och skriver till samma tabell såklart.

Vidare kunde en mer generell funktion skapats för hantering av det dataflöde som sker mellan formulär och databasinsättningar. Exempelvis genom att använda ASP.NET:s inbyggda funktionalitet "IFormCollection" där all data som skickas i ett HTTP-anrop kan fångas. Sedan hade denna data kunnat itereras över för att hämta all data för en insättning. Istället byggs den lista som krävs för insättning i funktionen "CreateRow" i figur 2 upp manuellt för varje insättning. Likaså vid radering av data. En så kallad ViewModel hade även kunnat användas här för att mer generalisera och strukturera hur indata ser ut för diverse anrop.

Slutligen hade databasen kunnat skyddas bättre mot SQL-injektion genom att använda parameteriserade frågor överallt, alltså frågor där den SQL-kod som ska köras är förbestämd, och användaren inte kan ta sig ut utanför de citattecken som omringar den text som matas in.

Bilagor

Bilaga A:

```
#1 @using MVC_databaskonstruktion.Utils
#2 @model TableObject
#3
#4 @{
#5     var dataTable = Model.DataSet;
#6     var urlQuery = new RouteValueDictionary();
#7     var deleteQuery = new RouteValueDictionary();
#8 }
#9
#10 <div class="overflow-auto w-100">
#11     <table class="table table-striped table-bordered">
#12         <thead class="table-dark">
#13             <tr>
#14                 @foreach (System.Data.DataColumn column in dataTable.Columns)
#15                 {
#16                     <th scope="col">
#17                         @column.ColumnName
#18                     </th>
#19                 }
#20                 @if (!string.IsNullOrEmpty(Model.Redirect))
#21                 {
#22                     <th scope="col" class="col-1">
#23                         Details
#24                     </th>
#25                 }
#26                 @if (!string.IsNullOrEmpty(Model.DeleteTable))
#27                 {
#28                     <th scope="col" class="col-1">
#29                         Delete
#30                     </th>
#31                 }
#32             </tr>
#33         </thead>
#34         <tbody>
#35             @foreach (System.Data.DataRow row in dataTable.Rows)
#36             {
#37                 <tr>
#38                     @for (int col = 0; col < dataTable.Columns.Count; col++)
#39                     {
#40                         <td>
#41                             @row[col]
#42                         </td>
#43                     }
#44                     @if (Model.PrimaryKeys?.Count > 0)
#45                     {
#46                         foreach (var key in Model.PrimaryKeys)
#47                         {
#48                             urlQuery[key] = row[key];
#49                         }
#50                         deleteQuery = new RouteValueDictionary(urlQuery);
#51                         deleteQuery["Table"] = Model.DeleteTable;
#52                     }
#53                     @if (!string.IsNullOrEmpty(Model.Redirect))
#54                     {
#55                         <td class="col-1">
#56                             @Html.ActionLink("Details", Model.Redirect, Model.ControllerName,
#57                                 urlQuery, new { @class = "btn btn-block btn-info m-0 w-100" })
#58                         </td>
#59                     }
#60                     @if (!string.IsNullOrEmpty(Model.DeleteTable))
#61                     {
#62                         <td class="col-1">
#63                             @Html.ActionLink("Delete", "Delete", Model.DeletionController,
#64                                 deleteQuery, new { @class = "btn btn-block btn-danger m-0 w-100" })
#65                         </td>
#66                     }
#67                 </tr>
#68             }
#69         </tbody>
#70     </table>
#71 </div>
```